Memory Dependence Prediction

by

Andreas Ioannis Moshovos

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1998

Abstract

As the existing techniques that empower the modern high-performance processors are being refined and as the underlying technology trade-offs change, new bottlenecks are exposed and new challenges are raised. This thesis introduces a new tool, *Memory Dependence Prediction* that can be useful in combating these bottlenecks and meeting the new challenges. Memory dependence prediction is a technique to guess whether a load or a store will experience a dependence. Memory dependence prediction exploits regularity in the memory dependence stream of ordinary programs, a phenomenon which is also identified in this thesis. To demonstrate the utility of memory dependence prediction this thesis also presents the following three novel microarchitectural techniques:

1. *Dynamic Speculation/Synchronization of Memory Dependences:* this thesis demonstrates that to exploit parallelism over larger regions of code waiting to *determine* the dependences a load has is not the best performing option. Higher performance is possible if memory dependence speculation is used especially if memory dependence prediction is used to guide this speculation.
2. *Speculative Memory Cloaking and Bypassing:* this thesis approaches memory as either an *inter-operation communication* or as a *data-sharing* mechanism. In the first case, memory is used to communicate values among instructions. In the second case, memory is used to hold values that are read repeatedly. Memory dependence prediction can be used to explicitly express either action so that loads can obtain a speculative value long before they can even access memory. Moreover, this thesis presents a technique to further reduce memory latency by linking directly the actual producer of a value with the actual consumers, taking loads and stores off the access path.
3. *Transient Value Cache (TVC):* Supporting highly-parallel execution requires the ability to perform multiple, simultaneous memory accesses. The TVC uses a small data cache to provide this support for a large fraction of loads while avoiding an increase in the latency of all other loads. This is achieved by using memory dependence prediction to selectively place the small data cache either in-series or in-parallel to the L1 cache.

Acknowledgments

Vasiliki has been a constant source of support, advice, devotion and above all optimism for as long as we have known each other. I am grateful for her patience and willingness to make sacrifices just so that I can continue working towards my doctorate. I am also grateful to her mother Naysika and her sister Dora for their love, support and forbearance. My parents, Koula and Yannis, and my brother Nikos have always been there for me providing encouragement and unconditional support. They made everything in their power to provide me with the best education, even if that meant we had to be separated. I only wish that my grandparents, Elpida and Andreas, who in many ways made me who I am, were still with us.

I want to express my appreciation to my advisor, Guri Sohi, for taking me as his student and for his support throughout my doctorate studies. He heavily influenced my way of thinking about technical issues and ideas. Above all, I am most grateful for that he forced me to develop my own ideas and learn how to best defend them. It has been a privilege working with him.

Over the past five and a half years I had the privilege of interacting with Jim Goodman, Mark Hill, Jim Smith and David Wood. Most if not all I know about Computer Architecture I learned from them and Guri. I am also grateful to all of them for serving in my committee. Mark Hill and Jim Smith also served in my prelim committee, while David Wood and Jim Smith had the dreadful task of reading this thesis. David's, Guri's and Jim's efforts made this thesis more readable and accurate. Jim Smith made life occasionally more tasteful with his paella. I am also grateful to Manolis Katevenis who introduced me to Computer Architecture and taught me how to make presentations.

My thanks to the members of the Multiscalar and Kestrel research groups, both past and present. Scott Breach and T. N. Vijaykumar have been close collaborators in the dependence speculation and synchronization work, and sounding boards for new ideas. I am grateful to Scott for writing the first simulation model for the memory dependence synchronization mechanism and for suggesting the merged MDST/MDPT organization. Amir Roth has been a collaborator in recent work on prefetching and control flow prediction. It has been a pleasure working with them. My thanks to Sridhar Gopal, Andy Glew, Avinash Sodani, Craig Zilles, Todd Austin, Steve Bennett, Harit Modi, Eric Rotenberg, Yannos Sazeides, Quinn Jacobson, Subramanya Sastry and Timothy Heil for the numerous discussions. My thanks to Doug Burger, Babak Falsafi, Alain Kagi and Subbarao Palacharla for many discussions about technical and not so technical issues. I most grateful to both Babak and Vijay for they were always willing to talk and offer advice. Babak also taught me how to say the most important things in Farsi. I am certainly going to miss the basketball games with Babak, Alain, Tia Newhall and Steve Seitz.

My stay in Madison was made more enjoyable and interesting by new friends: Dionisios Pnevmatikatos and Natalia Francis, Minos Garofalakis, Yannis Ioannidis, Yannis Schoinas and Leah Parks, Yannis Christou, Kyriakos and Sarah Kutulakos, Anastasia Ailamaki, Dimitris Komilis, Stefanos Kaxiras, Aggeliki Baltoyianni, Isidoros Sarinopoulos, Eirini Xagoraraki, Leonidas Galanis and Alkis Polyzotis.

Many thanks to Lorene Webber and to Debra Diewald. They have been always willing to help no matter how hectic their day was.

Contents

Chapter 1

Introduction

Program execution may initially seem as an inherently sequential process where the following steps are to be performed repeatedly, one after the other: (1) fetch an instruction, (2) read a set of input data, perform a calculation, if necessary (3) store the results for future reference, and finally, (4) decide which instruction to fetch next. From this perspective, it would seem that our only hope for faster processing would be to simply rely on advances in the underlying semiconductor device technologies; faster circuits may make performing each of the steps faster, and as result they reduce the time required to execute a whole program. Modern high-performance computing systems however, employ techniques that allow them to be quite a bit smarter about program execution in effect making better use of what solid-state technology has to offer at any given point of time. A plethora of techniques that empower these computing systems rely on empirical observations about program behavior to be effective. The motivation underlying these techniques is that programs, for the most part, do not behave randomly. Rather, they exhibit several idiosyncrasies, or in other words, they tend to exhibit regularity in how the operate and in what they produce. Which is to say that while in principle it is possible to design a program that would deem any such technique ineffective, such programs rarely have any other practical use. All this is best understood if we consider two prevalently used techniques of this kind: *caching* and *branch prediction*. Both techniques are implemented in virtually all modern high-performance processing systems.

Caching aims at better approximating the ideally large and fast memory device which corresponds to the data storage model typical programming languages present to their users. It has been long known that programs have a tendency to either access the same memory location repeatedly (temporal locality) or to access neighboring memory locations (spatial locality), both phenomena appearing close in time. Caches exploit this empirical observation by placing a set of recently accessed memory locations in a small and fast storage structure, a *cache*. As a result, many if not most of the memory accesses are serviced in the cache, benefiting from its low latency (how fast it responds) and high bandwidth (how much data it can respond with) characteristics. Had programs not exhibited temporal and spatial locality in their memory reference stream, caching would not have been an effective technique; data placed in the cache would rarely get referenced before evicted. Another example of a technique that exploits programs behavior is branch prediction. Branch prediction facilitates fetching and, often, executing instructions without having to wait to determine whether these instructions should be executed (normally, we would have to wait until the current instruction completes execution to determine which instruction should we execute next). Branch prediction exploits the regularity found in the control flow paths programs tend to follow (this is not intended to be a formal, exhaustive definition of the underlying phenomena branch prediction exploits). Had programs not exhibited regularity in their control flow, branch prediction and the performance benefits it produces, would not have been possible.

As the preceding discussion serves to demonstrate, in our efforts to build even faster or better suited to our purposes computing systems, we may seek to understand how ordinary programs behave and develop techniques that exploit this behavior to better utilize the resources offered by the underlying semiconductor device technologies. In this context, this thesis introduces a form of regularity exhibited by ordinary, sequential programs along with a number of micro-architectural techniques that exploit this regularity to improve performance. Specifically, we have identified that high levels of regularity is there to be found in the relationships formed when loads (memory read instructions) and stores (memory write instructions) access memory. These relationships are commonly referred to as memory dependences, hence the name we use for this phenomenon: *memory dependence locality.* Informally, memory dependence locality suggests that if at some point a particular load or store experiences a memory dependence, chances are that the same memory dependence will be experienced again the next time the same instruction is encountered (a formal definition of memory dependences and of memory dependence locality can be found in chapter 2).

Identifying a particular regularity in program behavior in itself is useful only in indicating a potential for exploiting this regularity. Actual techniques that make use of this opportunity for some practical purpose are required. To this respect, in this thesis we present *memory dependence prediction,* a technique facilitated by memory dependence locality and three micro-architectural techniques that are based on memory dependence prediction. In memory dependence prediction, information about the memory dependences exhibited during program execution is collected on-the-fly (i.e., while the program is running) and is used to make educated guesses on the forthcoming dependence behavior of the program. In our proposal these actions take place in a program and programmer transparent way via the use of architecturally invisible structures.

The three micro-architectural techniques we introduce are: (1) *memory dependence speculation and synchronization,* (2) *memory cloaking and bypassing,* and (3) *transient value cache.* A brief introduction of each of the aforementioned techniques and of their goals is given in the next section where we enumerate the contributions of this thesis.

1.1 Contributions

The contributions of this thesis are: (1) we identify that programs exhibit locality in their memory dependence stream, (2) we present memory dependence prediction, a technique that exploits memory dependence locality to guess whether a instruction will experience a dependence and if necessary which this dependence exactly is, and (3) we present three micro-architectural techniques that utilize memory dependence prediction to improve performance. In Section 1.1.1 we discuss memory dependence locality and memory dependence prediction. In sections 1.1.2 through 1.1.3 we discuss the three micro-architectural techniques we propose.

1.1.1 Memory Dependence Locality and Prediction

In this thesis we introduce *memory dependence localit*y, a characteristic exhibited by ordinary, sequential programs. Memory dependence locality refers to the regularity that exists in the relationships formed when loads and stores access memory (a formal definition of memory dependences is given in chapter 2). We consider two forms of memory dependence locality: (1) *memory dependence status locality* and (2) *memory dependence set locality* or, simply, *memory dependence locality.* Informally, memory dependence status locality suggests that, if at some point during execution a load or a store experiences a memory dependence of a particular type, it is highly probable that when encountered again, the same load or store will experience a memory dependence of the same type. In this case, which are the exact dependences experienced are not important, only whether such dependences exist is. Memory dependence locality is a specialization of memory dependence status locality. Memory dependence locality suggests that if at some point during execution a load or a store experience a particular dependence (or a set of dependences), it is highly probable that when encountered again, the same load or store will experience the same dependence (or dependences). In this case, not only whether dependences of a particular type exist is important, but also which exactly these dependences are.

Either form of memory dependence locality facilitates history-based prediction of the corresponding events. Specifically, in this thesis we present two memory dependence prediction techniques: (1) memory dependence status prediction, and (2) memory dependence set prediction, or simply, memory dependence prediction. *Memory dependence status prediction* is a technique that allows us to guess with high accuracy whether a load or a store has a memory dependence of a given type. *Memory dependence prediction* is a technique that allows us guess not only whether a given load or store has dependence but also which exactly this dependence (or dependences) is. Both techniques operate by (1) observing memory dependences as they occur through memory, (2) associating memory dependence information with the static instructions that are involved (e.g., with the PC of stores or loads), and (3) using the recorded information to make educated guesses on what dependences instructions will experience the next time they are encountered.

To support the efficacy of memory dependence prediction, in chapter 2 we study the memory dependence behavior of programs and demonstrate that: (1) sufficient regularity exist in both the memory dependence status and the memory dependence stream of programs, and (2) the working set of instructions with dependences is relatively small. The first property—as we explain in chapter 2—is a requirement of history-based prediction as it indicates that past dependence behavior is a good indicator of forthcoming memory dependence behavior. The second property suggests that we can collect and record dependence information for prediction purposes using finite structures of reasonable size. This is required if memory dependence prediction is to be of practical use.

In itself, memory dependence prediction serves just to provide highly accurate information about memory dependences. Techniques are required to make use of this information for some practical purpose. We present three such techniques in this thesis: (1) *dynamic speculation and synchronization of memory dependences, (2) speculative memory cloaking and bypassing,* and (3) *transient value cache*. A description of the goals and operation of each technique is given in the following three sections.

1.1.2 Dynamic Speculation and Synchronization of Memory Dependences

Techniques to exploit instruction-level parallelism (ILP) are an integral part of virtually all modern high-performance processors. With these techniques, instructions do not necessarily execute one after the other and in the order they appear in the program. Rather, instructions are executed in any order convenient provided however, that program semantics are maintained (i.e., the same results are produced). This ability is useful in reducing execution time by executing instructions in-parallel (many at the same time) and by avoiding stalling execution while an instruction takes its time to execute (e.g., it performs a relatively time consuming calculation or accesses a relatively slow storage device for its operands). An arena where ILP techniques are particularly useful is that of tolerating memory latency where these techniques are used to send loads requests as early as possible, overlapping memory processing time with other useful computation.

One approach to exploiting ILP is to first make sure that executing an instruction will not violate program semantics *before* the instruction is allowed to execute. In the case of a load, this action amounts to waiting to determine if a preceding, yet unexecuted store writes to the same memory location, that is whether a *true dependence* with a preceding store exists. However, and as we explain in detail in chapter 3, waiting to determine whether a preceding store writes to the same memory location is not the best option. Higher performance is possible if *memory dependence speculation* is used, that is, if a load is allowed to execute speculatively before a preceding store on which it *may* be data dependent. Later on, and after the preceding store has calculated its address, we can check whether program semantics were violated. If no true memory dependence is violated in the resulting execution order, speculation was successful. Otherwise, speculation was erroneous and corrective action is necessary to undo the effects of erroneous execution. A penalty is typically incurred in the latter case.

In this work we focus on dynamic memory dependence speculation techniques and study how existing methods of applying memory dependence speculation will scale for future generation processors. Prior to this work, memory dependence speculation was used whenever the opportunity to execute a load existed. We will refer to this technique as *naive memory dependence speculation*. In this work we demonstrate that as processors attempt to extract higher-levels of ILP by establishing larger instruction windows: (1) memory dependence speculation becomes increasingly important, and (2) the net penalty of memory dependence mispeculation can become significant. The latter observation suggests that further performance improvements are possible if mispeculations could be avoided. Specifically, we demonstrate that further performance improvements are possible under the following two execution models: (1) a centralized, continuous window ILP processor, and (2) in a distributed, split-window ILP processor. In the centralized, continuous window processor, the net penalty of mispeculation becomes significant when loads cannot inspect the addresses of preceding stores either because a mechanism is not provided (to simplify the design) or because of the latency required to inspect store addresses. In the distributed, spit-window processor mispeculations are problematic independently on whether loads can inspect preceding store addresses. Moreover, we demonstrate that the potential benefits increase as the size of the instruction window also increases in either processor environment.

Motivated by the aforementioned observations we study the trade-offs involved in memory dependence speculations and propose techniques to improve the accuracy of memory dependence speculation. Specifically, we propose techniques to: (1) identify via memory dependence prediction those loads and stores that would otherwise be mispeculated, and (2) delay load execution *only as long as it is necessary* to avoid a memory dependence mispeculation. The best performing technique we propose is *memory dependence speculation and synchronization*, or *speculation/synchronization*. With this technique, initially loads are speculated whenever the opportunity exists (as it is common today). However, when mispeculations are encountered, information about the violated dependence is recorded in a memory dependence prediction structure. This information is subsequently used to predict whether the immediate execution of a load will result in a memory dependence violation, and (2) if so, which is the store this load should wait for.

Timing simulations show that for a distributed, split-window processor (i.e., Multiscalar [26,14,82,27,40,92,13]), our technique can improve performance by 28% for integer codes and 15% for floating point codes on the average. More importantly, the performance obtained through the use of our techniques is very close to that possible with perfect, in advance knowledge of all memory dependences (31% and 17% respectively), suggesting that our method is successful in attaining both goals of memory dependence speculation: (1) avoiding mispeculations, and (2) delaying load execution only as long as it is necessary.

We also study memory dependence speculation under a centralized, continuous window processor (typical current superscalar) that utilizes fetch and execution units of equal bandwidth, and a program order priority scheduler (i.e., when there are many instructions ready to execute, the older ones in program order are given precedence). For this processor model we make two observations. The first is that using an address-based load/store scheduler (i.e., a structure where loads can inspect preceding store addresses to decide whether memory dependences exist) coupled with naive memory dependence speculation offers performance very close to that possible with perfect, in advance knowledge of all memory dependences, provided that going through the address-based scheduler does not increase load latency. The second is that if building an address-based load/store scheduler is not an option (clock cycle) or not a desirable option (complexity), naive memory dependence speculation can still offer most of the performance benefits possible by exploiting load/store parallelism. However, under this set of constraints the net penalty of mispeculation is significant suggesting that our memory dependence speculation and synchronization technique might be useful in improving performance. Specifically, timing simulations show that an implementation of our techniques results in performance improvements of 19.7% (integer) and 19.1% (floating-point) which are very close to those ideally possible: 20.9% (integer) and 20.4% (floating-point).

1.1.3 Speculative Memory Cloaking and Bypassing

Faster execution requires faster methods of reading and writing memory values. The memory dependence speculation and synchronization technique we described in the previous section helps in reducing the observed memory latency by allowing loads to access memory earlier. However, even with this technique, the interface used to read and write memory values and the mechanisms implementing memory remain the same: a load or store has to first calculate an address and then use this address to access memory. Yet, memory can be viewed as an interface through which programs synthesize a desired, possibly elaborate action. Which is to say that while from the perspective of a single instruction memory appears as a storage device, from the perspective of the program as a whole, reading or writing a memory value is really a single step in a series of actions which combined produce a desired effect. Just because we have decided to use an address-based memory interface in expressing what the program does, this does not imply that we have to implement this action in that way also. This perspective hints to another direction of improving memory performance, that of: (1) first identifying what purpose memory serves at any given point, (2) then, studying whether the address-based interface introduces any inefficiencies or overheads, and (3) if so, designing mechanisms that can perform the desired action faster.

In this thesis we identify two common uses of memory: (1) *inter-operation communication*, and (2) *data-sharing.* In inter-operation communication a store is used to write a memory value so that loads may later read it. In data-sharing, a memory location is used as a place holder for values that are read repeatedly (i.e., by many loads) in the program. As we explain in detail in chapter 4, the address-based memory interface introduces both overheads and inconveniences in performing these actions. For example, for inter-operation communication to take place, both the store and the load have to calculate their address and then locate each other using that address. These actions take time and more importantly the value being communicated may be available long before these actions complete (a detailed discussion of the overheads and inconveniences introduced by the address-based memory interface is delayed until chapter 4).

We also observe that an explicit specification of either action has potential advantages over the current address-based specification. In an explicit specification of inter-operation communication both the store and the load that ought to communicate, not only are given knowledge of the desired action but also can locate each other directly without having to first calculate an address. Similarly, in an explicit specification of data-sharing, a load that will access a previously accessed memory location, is not only given knowledge of this fact, but is also provided with a mechanism to identify the earlier load that accessed the same memory location without having to first calculate an address.

Motivated by the aforementioned observations we present *speculative memory cloaking*, or *cloaking*, a technique that: (1) transparently converts the address-based specification of inter-operation communication and of data-sharing into an explicit form, and (2) uses this explicit representation to allow loads to obtain their memory value by just naming an earlier store or load that wrote or accessed it. In cloaking, memory dependence prediction is used to identify those loads and stores that communicate and to identify those loads that access a common memory location. This information is used to create direct, albeit speculative links between these instructions so that values can flow directly, as soon as they become available. In particular, the mechanism we present operate by associating memory dependence information with the PC of the static loads and stores. This permits loads and stores to pass values via cloaking using information derived using their PCs.

Aiming at further reducing memory value access latency, we observe that loads and stores typically do not change the data they write or read. These instructions are really used as agents retrieving or placing memory data that other instructions produce or consume; the value written by a store is produced by another instruction, while the value read by a load is passed to other instructions. Motivated by this observation we propose *speculative memory bypassing*, a straightforward extension to cloaking, that speculatively bypasses loads and stores, linking producing and consuming instructions directly. Specifically, when memory is used as an inter-operation mechanism, speculative memory bypassing converts DEF-STORE-LOAD-USE chains into DEF-USE ones. When memory is used to hold data that is read repeatedly, this technique consolidates a series of LOAD_1-USE_1...LOAD_N-USE_N chains into a single LOAD_1-USE_1...USE_N producer-consumer graph.

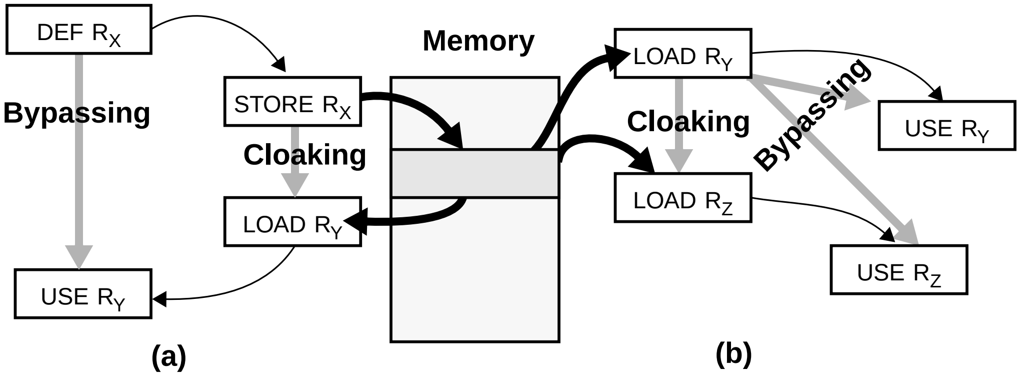


Figure 1.1: *Speculative Memory Cloaking and Bypassing. (a) Inter-operation communication: exploiting read-after-write dependences. (b) Data sharing: exploiting read-after-read dependences.*

The effect achieved via cloaking and bypassing is illustrated in Figure 1.1. Cloaking allows values to flows between loads and stores, while bypassing takes loads and stores off the communication path, allowing values to flow directly from the actual producer to the actual consuming instructions.

Since we make use of memory dependence prediction, the proposed techniques are speculative and so are the values obtained through their use. Accordingly, value verification through the traditional memory name space is necessary. However, this verification can take place while the speculative values are used for further processing. Provided that speculation is successful, the observed memory latency is reduced as instructions that would otherwise wait for the traditional memory hierarchy to provide a value may now execute earlier, possible even before the memory access is initiated.

Trace driven simulations show that a straightforward implementation of a combined cloaking/bypassing mechanism can supply correct values for about 60% and 50% of all loads for the integer and the floating point programs respectively. Timing simulations show that for a fairly aggressive and highly optimized 8-way superscalar with an 128-entry window and that uses memory dependence speculation, cloaking/bypassing can improve performance by 4% and 9%.

1.1.4 Transient Value Cache

Highly parallel execution can benefit from both low memory latency and from the ability to perform multiple memory accesses in parallel. Both speculation/synchronization and cloaking/bypassing aim at reducing memory latency. The final technique we present in this work aims at supporting multiple memory accesses per cycle. This technique is motivated by the relatively large fraction of loads that read a value that was either recently written by a store or recently read by another load. Specifically, we found that roughly 70% (integer programs) and 43% (floating-point programs) of all loads read a memory location that is within the last 128 memory locations accessed by preceding stores or loads. This result suggests that a relatively small data cache capable of storing just 128 memory locations could potentially service all these loads. Such a small data cache could help in supporting multiple memory requests for the following two reasons: (1) it could be easier to multiport than a reasonably sized L1 data cache, and (2) loads that would get serviced in this small data cache will not have be exposed to the L1, freeing up L1 port resources to be used for other loads. Unfortunately, placing such a data cache in between the processor and the L1 would increase the latency of all dynamic loads that would not hit in it (30% and 57% respectively). Placing it in series with the L1 data would not be of much use either as all loads will have to be exposed to both the L1 and the small data cache. What is needed is a method to place this small data cache in series with the L1 for those loads that will hit in it and in parallel to the L1 for all other loads. This is exactly what the *Transient Value Cache* (TVC) tries to do. The *Transient Value Cache*, or TVC is a novel memory hierarchy component that combines a memory dependence status predictor and a relatively small, narrow data cache. The basic approach is illustrated in Figure 1.2. The TVC records in its data cache component the *n*th most recent accessed memory locations as a traditional data cache would. However, contrary to what is done in a traditional memory hierarchy, the data cache component does not always appear in series with the rest of the memory hierarchy. Instead, its placement is decided using a memory dependence prediction as follows: When a load is ready to access memory, a prediction is made on whether the memory location it will access is resident in the TVC, or viewed differently whether the load has a dependence with a recent store or load respectively. If so, the load is send only to the TVC, in which case the TVC appears in *series* with the L1 data cache. Otherwise the load is send to both the TVC and the rest of the memory hierarchy, in which case the TVC appears in *parallel* with the L1 data cache. Provided that prediction accuracy is high, the potential benefits of the TVC approach are: (1) the loads that hit in the TVC are hidden from the rest of the memory hierarchy, freeing up L1 data cache ports to be used by other loads, (2) the latency of loads that are unlikely to hit in the TVC remains unchanged. High prediction accuracy is essential as the TVC may result into increased load latency when prediction incorrectly indicates that a load will find its data in the TVC. We also present a possible store-to-store dependence (WAW) status prediction extension, were the TVC is also used to hide from the rest of the memory hierarchy those store accesses that are likely to be overwritten quickly precluding potential problems with writeback traffic contention while reducing the L1 port requirements.

Trace driven simulation shows that a TVC comprised of a counter-based memory dependence status predictor and a 128-word (512 bytes) cache component can service 66.8% and 40.4% of all loads on the average and for the integer and floating point codes respectively. More importantly, only 3.3% and 1.1% of all loads would observe a latency increase as the result of erroneous memory dependence status prediction. In contract, a traditional memory organization that would place the 128-word data cache in series with the L1, would result in 70.1% (integer) and 43.1% (floating-point) reduction of loads at the L1 interface while increasing the latency of 29.7% (integer) and 43.7% (floating-point) of all loads. While the aforementioned results provide an indication of the potential of the TVC approach, further investigation is required to determine its performance impact.

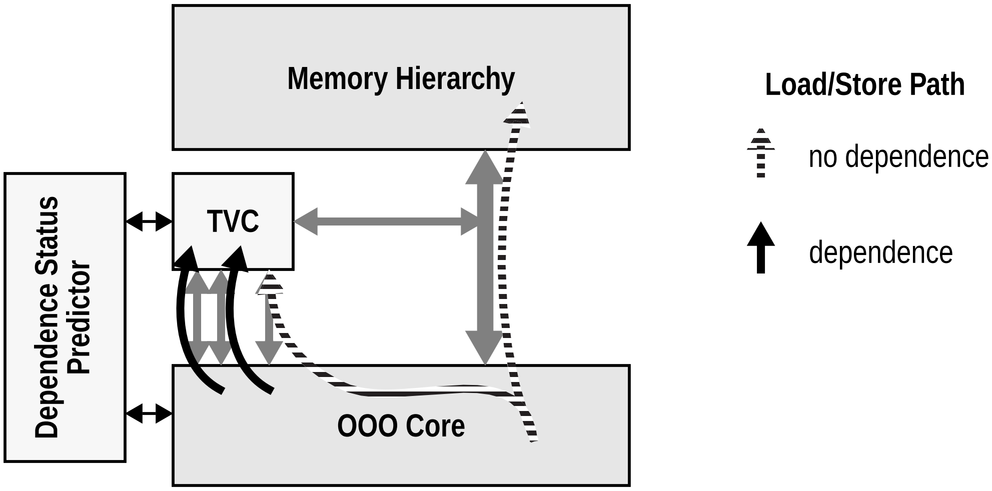


Figure 1.2: *The Transient Value Cache*

1.2 Thesis Organization

The remainder of this thesis is organized as follows: in Section 1.3 we detail the experimental framework used to evaluate the ideas and techniques we propose. In chapter 2, we provide a formal definition of memory dependences, a short introduction to the principles of operation of history-based memory dependence predictors and also present an analysis of the memory dependence behavior of the programs studied. This analysis focuses on those aspects of memory dependence behavior that are interesting from the perspective of history-based memory dependence prediction. It is here that we provide experimental evidence in support of the efficacy of memory dependence prediction. In chapter 3, we study memory dependence speculation and present memory dependence speculation and synchronization. In chapter 4, we present speculative memory cloaking and bypassing. In chapter 5, we present the transient value cache approach. Finally, we summarize our findings in chapter 6 and offer suggestions on how this work can be extended. In the appendix we present additional measurements that relate to memory dependence prediction and to its applications we present in this thesis.

1.3 Experimental Framework

In this section we detail the experimental framework we used for the experiments reported in this thesis. In Section 1.3.1 we provide details on the programs analyzed and on the compiler infrastructure we used. In Section 1.3.2 we discuss our simulation methodology,

1.3.1 Programs and Compiler Infrastructure

In all experiments reported in this thesis we used the SPEC95 benchmark suite [86]. We used two input data sets. The exact parameters used per input data set are given in Table . Unless otherwise noted, the first input data set is used. All programs were compiled using a modified version of the GNU *gcc* compiler version 2.7.2. The modifications allow the compiler to also generate binaries for the Multiscalar architecture [26, 82] and only affects binaries compiled for that architecture. The base instruction set architecture is the MIPS-I [42] but with no architectural delay slots of any kind. The modifications done to gcc along with a description of the additional instructions introduced to support the Multiscalar architecture are detailed in [92]. Fortran sources were compiled by first converting them to C using AT&T's *f2c* compiler. Unfortunately, we have no way of determining how the use of a FORTRAN-TO-C translator impacts the quality of the produced code. However, it is to be expected that an actual FORTRAN compiler would improve the quality of the produced code mainly as the result of better static disambiguation and scheduling. We note that such optimizations may affect the results of the techniques we propose either way. For example, while better disambiguation may help in exposing some of the load/store parallelism in floating point programs, hence reducing the potential of a dynamic approach, it will also reduce the distance between dependent loads and stores increasing the probability of mispeculation, hence increasing the potential of the speculation/synchronization technique presented in chapter 3. All programs were compiled using the -O2 optimization level and with loop unrolling and function inlining enabled.

Two binaries were generated per program: (1) a MIPS-I binary, and (2) a Multiscalar binary. The instruction counts, along with the fraction of loads and stores executed per binary and input data set are given in Table 1.1.

1.3.2 Simulation Methodology

We employ two simulation techniques: (1) trace-driven simulation and (2) detailed, execution-driven timing simulation. Traces are generated using two functional simulators, one for MIPS-I ISA and one for the Multiscalar architecture. The functional simulators execute *all* user-level instructions. System calls are serviced by the OS of the host machine. The functional simulators we used are derivatives of the Multiscalar functional simulator [13]. In all experiments that utilize functional simulation we included all user-level instructions in our measurements.

We also make use of detailed, execution-driven timing simulation. For this purpose we utilized two simulators, one that simulates a dynamically scheduled superscalar processor and one that simulated a Multiscalar processor. Both simulators are derivatives of the Multiscalar timing simulator [13]. The out-of-order core simulator was rewritten to facilitate simulation of the techniques we examine and to better approximate the processor models we study. Moreover, mechanisms to collect advance memory dependence informations were incorporated. While these modifications were required for our experimentation that also decreased simulation speed considerably. The simulators execute all user-level instructions including those on control speculative paths. Systems calls are redirected to the OS of the host machine.

The default superscalar configuration we used is detailed in Table 1.2. We used a 32K data cache to compensate for the relatively small memory working sets of the SPEC95 programs. For some experiments we use a 64-entry reorder buffer model. That model, has 4 copies of all functional units, a 2-port load/store queue and memory system, and can fetch up to 4 instructions per cycle.

The default Multiscalar configuration we used is detailed in Table 1.3. For some experiments we used models of processing units with 4 copies of all functional units and all 4 load/store ports. For those experiments the number of banks and miss handlers for the I-cache, D-cache and the ARB was partitioned accordingly to provide four times the bandwidth of the configuration shown in Table 1.3.

Finally, to attain reasonable simulation times we utilized sampling for the timing simulations. In this technique which was also employed, for example, in [96,67,13], the simulator switches between functional and timing simulation. The mode of simulation is changed once a predefined number of instructions have been simulated. In all sampling simulations the observation size is 50,000 instructions. We chose sampling ratios that resulted in rougly 100M instructions being simulated in timing mode (i.e., sample size). We did not use sampling for *099.go, 107.mgrid, 132.ijpeg* and *141.apsi*. We used a 1:1 timing to functional simulation ratio

Program	**Input Data Set 1**	**Input Data Set 2**
SPECint’95		
099.go	play level = 9, board size = 9	train input set: play level = 9, board size = 50
124.m88ksim	modified test input: 370 iterations of Dhrystone Benchmark	train input set
126.gcc	reference input file recog.i	test input: file cccp.i
129.compress	modified train input: maximum file size increased to 50,000	modified train input: maximum file size increased to 100,000
130.li	modified test input: (queens 7)	train input set
132.ijpeg	test input	train input set
134.perl	modified train input: jumple.pl with dictionary reduced by retaining every other 15th word	train input set
147.vortex	modified train input: persons.250k database, PART_COUNT 250, LOOKUPS 20, DELETES 20, STUFF_PARTS 100, PCT_NEWPARTS 10, PCT_LOOKUPS 10, PCT_DELETES 10, PCT_STUFFPARTS 10	train input set
SPECfp’95		
101.tomcatv	modified train input: N = 41	train input set
102.swim	modified train input: X = 256, Y = 256	test input set
103.su2cor	modified test input: LSIZE = 4 4 4 8 16	test input set
104.hydro2d	modified test input: MPROW = 200	test input set
107.mgrid	modified test input: LMI = 4	test input set
110.applu	modified train input: itmax = 25, nx = 10, ny = 10, nz = 10	train input set
125.turb3d	modified train input: nsteps = 1, itest = 0	test input set
141.apsi	modified train input: grid points x = 32, grid points z= 8, time steps = 130	test input set
145.fpppp	modified reference input: natoms = 4	train input set
146.wave5	modified train input: particle distribution 1000 20, grid size 625x20	train input set

Benchmark input parameters.

	Non-Multiscalar						*Multiscalar*		
	Input Data Set 1			*Input Data Set 2*			*Input Data Set 1*		
Program	*IC*	*Loads*	*Stores*	*IC*	*Loads*	*Stores*	*IC*	*Loads*	*Stores*
SPECint'95									
099.go	133.8	20.9%	7.3%	553.7	21.3%	7.9%	141.2	22.7%	7.8%
124.m88ksim	196.3	18.8%	9.6%	141.5	18.5%	13.3%	213.3	17.3%	8.7%
126.gcc	316.9	24.3%	17.5%	1,496.5	23.4%	19.4%	333.9	23.9%	17.3%
129.compress	153.8	21.7%	13.5%	296.4	21.7%	12.9%	153.8	21.7%	13.5%
130.li	206.5	29.6%	17.6%	182.9	25.4%	16.1%	229.7	26.6%	15.7%
132.ijpeg	129.6	17.7%	8.7%	1,478.2	17.6%	8.4%	139.2	18.6%	9.0%
134.perl	176.8	25.6%	16.6%	> 2,200.0	25.5%	16.4%	176.8	25.6%	16.6%
147.vortex	376.9	26.3%	27.3%	> 2,200.0	28.6%	24.8%	390.7	21.3%	30.6%
SPECfp'95									
101.tomcatv	329.1	31.9%	8.8%	> 2,200.0	31.5%	8.9%	333.7	30.6%	8.7%
102.swim	188.8	27.0%	6.6%	753.1	27.0%	6.6%	191.8	27.0%	6.5%
103.su2cor	279.9	33.8%	10.1%	1,099.9	34.0%	10.0%	283.7	33.5%	10.0%
104.hydro2d	1,128.9	29.7%	8.2%	1,130.1	29.7%	8.2%	1,162.6	29.4%	8.0%
107.mgrid	95.0	46.6%	3.0%	> 2,200.0	49.3%	2.1%	100.6	46.1%	5.2%
110.applu	168.9	31.4%	7.9%	649.3	31.5%	7.9%	171.8	31.9%	8.0%
125.turb3d	1,666.6	21.3%	14.6%	> 2,200.0	21.1%	14.1%	1,701.7	20.6%	14.4%
141.apsi	125.9	31.4%	13.4%	>2,200.0	30.8%	12.2%	129.5	31.0%	13.5%
145.fpppp	214.2	48.8%	17.5%	469.6	48.9%	17.2%	202.5	49.4%	11.0%
146.wave5	290.8M	30.2%	13.0%	>2,200.0	32.0%	12.7%	299.3	30.2%	12.9%

Table 1.1: *Benchmark Execution Characteristics. Instruction counts ("IC" columns) are in millions.*

(i.e., once 50000 instructions are simulated in timing mode, we switch to functional mode and simulate 50000 instructions before switching back to timing mode, and so on) for: *110.applu, 124.m88ksim, 130.li, 134.perl* and *145.fpppp*. We used a 1:2 timing to functional simulation ratio (i.e, once 50000 instructions are simulated in timing mode, we switch to functional mode and execute 100000 instructions before switching back to timing mode, and so on) for: 1*01.tomcatv, 102.swim, 126.gcc, 129.compress, 146.wave5* and *147.vortex*. We used a 1:3 timing to functional simulation ratio for *103.su2cor.* And finally, we used a 1:10 timing to functional simulation ratio for: *104.hydro2d* and *125.turb3d*. During the functional portion of the

Fetch Interface	Up to 8 instructions can be fetched per cycle. Up to 4 fetch requests can be active at any time. Combining of up to 4 non-continuous blocks.
Branch Predictor	64K-entry combined predictor [58]. Selector uses 2-bit counters. 1st predictor: 2bit counter based. 2nd predictor: *Gselect* with 5-bit global history. 4 branches can be resolved per cycle. 64-entry call stack. 2K BTB. Up to 4 predictions per cycle.
Instruction Cache	64K, 2-way set associative, 8 banks, block interleaved, 256 sets per bank, 32 bytes per block, 2 cycles hit, 10 cycle miss to unified, 50 cycle miss to main memory. Lockup free, 2 primary misses per bank, 1 secondary miss per primary. LRU replacement.
OOO core	128-entry reorder buffer, up to 8 operations per cycle, 128-entry combined load/store queue, with 4 input and 4 output ports. Loads can execute as soon as their address becomes available. Stores check for memory dependence violations by comparing addresses and data. It takes a combined 4 cycles for an instruction to be fetched and placed into the reorder buffer.
Architected Registers	64 integer, 64 floating point, HI, LO and FSR.
Functional Units	8 copies of all functional units. All are fully-pipelined. 4 memory ports.
Functional Unit Latencies	Integer: 1 cycle latency except for: multiplication 4 cycles, division 12 cycles,. Floating point: 2 cycles for addition/subtraction and comparison (single and double precision or SP/DP). 4 cycles SP multiplication, 5 cycles DP multiplication, 12 cycles SP division, 15 cycles DP division.
Store Buffer	128-entry. Does not combine store requests to memory. Combines store requests for load forwarding.
Data Cache	32K, 2-way set associative, 4 banks, 256 sets per bank, 32 bytes per block, 2 cycle hit, 10 cycle miss to unified, 50 cycle miss to main memory. Lockup-free, 8 primary miss per bank, 8 secondary miss per primary. LRU replacement.
Unified Cache	4M-byte, 2-way set associative, 4 banks, 128-byte block, 8 cycle + # 4 word transfer * 1 cycle hit, 50 cycles miss to main memory. Lockup-free, 4 primary miss per bank, 3 secondary per primary.
Main Memory	Infinite, 34 cycle + #4 word transfer * 2 cycles access.

Table 1.2: *Default configuration for superscalar timing simulations*

simulation the following structures were simulated: I-cache, D-cache, and branch prediction. Table 1.4 presents data that are useful in quantifying the error that is introduced by the use of sampling. In this experiments we report the relative difference in IPC (instructions per cycle) reported with sampling simulation over full timing simulation of the default superscalar configuration.

Processing Units	4 or 8, single task, 16-entry reorder buffer with 8-entry scheduler. **Functional Units:** 2 copies of all functional units, except for load/store units that has 1 port. Latencies same as in TableSection 1.2. **Load/Store unit:** 16 entry load/store queue, 16 entry store buffer non-combining to memory, combining for local load requests. Loads may execute after all preceding local stores have calculated their address. **Fetch interface:** 2 instructions per cycle, one branch prediction, 16-entry call-stack, 1K BTB. **Control Predictor:** global-pattern based, 16-bit pattern register, 64K-entry, 2-bit counters, 2 targets.
Inter-Task Predictor	Path-based DOLC=7,3,6,8 path register 64K-entry, 2-bit counters, 4 targets [13]. 64-entry call-stack.
Task Cache	1K-entry, 2-way set associative, 64-byte task descriptor, LRU replacement. 1 cycle hit, 12 cycle miss to unified, 50 cycle miss to main memory. 1 bank, bus, lockup.
Instruction Cache	64K, 2-way set associative, #PU banks, block interleaved, 32 bytes per block, 1 cycle hit, 10 cycle miss to unified, 50 cycle miss to main memory. Lockup free, 8 primary misses per bank, 8 secondary miss per primary. LRU replacement. Crossbar with one port per PU and per bank.
Register File	4 registers per cycle, 2 cycle latency between adjacent units
Address Resolution Buffer	#PU banks, 32-way set associative, 128 entries per bank, byte disambiguation granularity. 2 cycle hit.
Data Cache	32K, 2-way set associative, #PU banks, 32 bytes per block, 2 cycle hit, 10 cycle miss to unified, 50 cycle miss to main memory. Lockup-free, 8 primary miss per bank, 2 secondary miss per primary. LRU replacement. Crossbar with one port per PU and per bank. Same block access combining for crossbar.
Unified Cache	4M-byte, 2-way set associative, 4 banks, 128-byte block, 8 cycle + # 4 word transfer * 1 cycle hit, 50 cycles miss to main memory. Lockup-free, 4 primary miss per bank, 3 secondary per primary.
Main Memory	Infinite, 34 cycle + #4 word transfer * 2 cycles access.

Table 1.3: *Default configuration for Multiscalar timing simulations.*

	IPC Full	Relative Difference w/ Sampling
099	1.81	0.0% (N/A)
124	3.54	-1.142%
126	2.56	-1.171%
129	2.32	0.0%
130	2.38	0.0%
132	4.16	0.0% (N/A)
134	2.77	-1.818%
147	4.66	-0.858%
101	3.06	+0.653%
102	2.39	+3.347%
103	3.51	-0.854%
104	3.48	-0.574%
107	5.11	0.0% (N/A)
110	4.29	-0.233%
125	4.63	0.0%
141	3.25	0.0% (N/A)
145	3.89	-0.514%
146	3.85	-0.239%

Table 1.4: *Error introduced by the use of sampling in a timing simulation. "IPC Full" columns report the instructions per cycle execution rate when no sampling is used. Also reported is the relative difference in IPC introduced by the use of sampling.*

Chapter 2

Memory Dependence Behavior Analysis

Before we embark into describing possible applications of memory dependence prediction it is best if we develop an understanding of the memory dependence behavior of programs. Accordingly, in this chapter we present a characterization of memory dependence behavior. This information will aid us during both the motivation and the design process of the applications described in the chapters that follow. Hopefully, this information will also help stimulate other applications of memory dependence prediction. We should note that there are certainly many more attributes of memory dependence behavior than those we consider in this study. Moreover, there might be types of memory dependence information that might be interesting other than those we consider in this work (a description can be found in Section 2.3). Rather than performing an exhaustive analysis, we focus on those attributes that seem most relevant to a specific class of memory dependence predictors[1] (discussed in Section 2.2).

An overview of the specific results presented in this chapter along with a justification of why we include them is delayed until Section 2.3.1. This is necessary to motivate the relevance of our metrics we need to first formally define memory dependences and review the principles underlying the operation of the class of memory dependence predictors we consider. However, in a nutshell, the results of this chapter are: (1) most of loads and stores experience dependences, (2) relatively small structures (e.g., 4K entries) can be used to capture a large fraction of that dependence activity, (3) the working set of memory dependences is relatively small, (4) memory dependences exhibit fairly regular behavior. As will become apparent by the discussion

[1] There is an inherent difficulty in making generally applicable observations about the "predictability" —which informally can be defined as the ability to design automata capable of guessing the relevant information— of memory dependences or other program related information. After all, the program itself is an automaton that may be used to predict its own actions (in Section 6.2.5, we will discuss such a possibility). The interested reader can refer to any description of algorithmic information content and of complex adaptive systems, e.g., [29] chapters 2 through 4. For this reason, in this work we focus on a specific class of predictors and measure those aspects of memory dependence behavior that seem relevant for those predictors.

of this chapter, these results constitute strong indications that memory dependences may be amenable to history-based prediction.

The rest of this chapter is organized as follows. Before we proceed into the details of the memory dependence behavior analysis, we briefly define memory dependences and their types (Section 2.1) and present an abstract description of a class of history-based memory dependence predictors (Section 2.2). The material presented in these two sections allows us to motivate the relevancy of the metrics presented in Section 2.3 and aid in their interpretation. An overview of the metrics presented is given in Section 2.3.1. A summary of our findings is given in Section 2.4. As we will discuss in Section 2.1, there are four possible types of memory dependences. Much of the discussion in this chapter focuses on those two types (read-after-write and read-after-read) which we extensively use in the applications we present later on.

2.1 Memory Dependence Types

In this section we review what memory dependences are, present the various types of memory dependences and explain what implications each of these types has on interpreting program semantics. We also define static and dynamic dependences and discuss what possible shape the dependences of an instruction may take (i.e., whether they are one-to-one or many-to-one).

A memory dependence is a relationship between two instructions that access memory. We could define memory dependences simply as the relationships formed when instructions access a common memory location (address). However, such a definition will encompass many more relationships than those that are of interest for our purposes. For example, such a definition will allow a dependence among a memory read (load) and all preceding memory writes (stores) to the same memory location. To define memory dependences precisely, we first need to define the concept of *memory location versions*. Throughout the course of execution, a memory location may be used to hold many different values. Every time a value is written to a memory location by a memory write (store), a new *version* of that memory location is created. As programs are written with an implied, total order, memory location versions can also be ordered according to the program implied order. With this definition in hand we can now proceed to define memory dependences, restricting our attention to the relationships formed among instructions starting from one that creates a new version of a memory location and ending with the one that creates the immediately succeeding in program order version of the same memory location.

```
1: sw M(100), r1                                  1: sb 101, r1
2: lw  M(100), r1       for i = 0 to 99           2: sb 100, r1
3: lw  M(100), r2           2: sw a[i], r1        3: lh  100, r2
4: sw M(100), r3            3: lw a[i], r2        4: sh 100, r3
5: lw M(100), r4

      (a)                     (b)                      (c)
```

Figure 2.1: *Memory dependence examples. "lw" ("sw") stands for "load word" ("store word"), where a "word" is four bytes long. "lh" ("sh") stands for "load half-word" ("store half-word") where a "half-word" is two bytes long. Finally, "lb" ("sb") stands for "load byte" ("store byte").*

A memory dependence is a relationship between two memory accessing instructions that either create or read the same version of a memory location or that create the immediately succeeding in program order version of the same memory location. Since there are two types of memory accessing instructions, loads and stores, that read and write memory data respectively, there are four possible types of memory dependences: *read-after-write (RAW), read-after-read (RAR), write-after-write (WAW)* and *write-after-read (WAR)*. A RAW dependence is formed when a load reads the memory location version written by a preceding store. A RAR dependence is formed when two loads read the same version of a memory location. A WAW dependence is formed in between a store that creates a new version of a memory location and a preceding store that created the most recent in program order version of the same memory location. Finally, a WAR dependence is formed between a store that creates a new version of a memory location and any preceding load that reads the immediately preceding in program order version of the same memory location. The example of Figure 2.1, part (a) is useful in illustrating the various memory dependence types. A sequence of five memory instructions is shown, numbered and in program order. There are three RAW dependences: (1,2), (1,3) and (4,5). There is one RAR dependence (2,3). There are two WAR dependences: (2,4) and (3,4). Finally, there is one WAW dependence: (1,4). As per our definition no dependence exists among load 5 and any instruction before store 4 as the latter creates a new version of memory location 100.

Memory dependences contain ordering information as all are of the form X-*after*-Y. This order is derived from the order in which the instructions appear in the original sequential program order. However, it should be noted that to interpret program semantics, RAR dependences do not impose any ordering restrictions as two loads that are connected via a RAR dependence can execute in any order with respect to each other. (For this reason, the term “RAR relationship” might have been a more appropriate term. However, for uniformity we will use the term RAR dependence.) All other dependences however, have implications on what is required to maintain sequential semantics. A load that has a RAW dependence with a preceding store must read the value written by that store. A store that has a WAR dependence with a preceding load must not overwrite the data read by the load before the latter had a chance to read it. Finally, WAW dependences dictate that in the program order, once both stores are encountered the correct memory state is defined by the value written by the latter store.

At times it is useful to differentiate between *static dependences* and their *dynamic instances*. A static dependence identifies the pair of static instructions that are connected via the dependence at some point during execution. For example, a static dependence can be a (store PC, load PC) pair, where by PC we denote the program address where the corresponding load or store resides. A dynamic instance of a static dependence, in addition to the dependent static instructions, also identifies their specific dynamic instances. Typically, a static dependence will have multiple dynamic instances. The example of Figure 2.1 part (b) illustrates the difference between a static dependence and its dynamic instances. A loop is shown, whose iterations first write to the *i*th element of array `a[]` (store at line 2) and then read the value written (load at line 3). A static dependence exists between the store at line 2 and the load at line 3. When this code executes, multiple dynamic instances of this static dependence will be encountered, one per iteration of the loop.

Another interesting characteristic of memory dependences is their *shape*, which for the purposes of this study we define as the number of distinct memory dependences a load or store experiences (a precise definition is given in Section 2.3.3). We can measure shape on each dynamic load or store instance separately (e.g., how may loads read the value written by a particular instance of a given static store) or we can measure shape over all dynamic instances of a given static store or load (e.g., after the program has executed how many unique loads accessed values written by instances of a given store). A given load or store may have multiple static dependences of all possible types (for example, as the result of control flow). The same is true for each dynamic instance of loads and stores: Since a memory version can be read by multiple loads before it is overwritten, multiple RAW (store at line 1 in Figure 2.1, part (a)) and WAR (store at line 4 of the same figure) dependences are possible for stores. For the same reason multiple RAR dependences are possible per load (this would be the case in part (a) of Figure 2.1, if another load was reading memory location 100 before the store at line 4 and after the store at line 1). A load may also have multiple RAW and WAR dependences while a store may also have multiple WAW dependences. This is a result of the plurality of memory data types supported by a typical ISA (for example, in the MIPS-I ISA [42], a load or store may access anywhere from 1 to 4 consecutive bytes in memory). An example is shown in part (b) of Figure 2.1. Load 3, that reads two bytes starting from location 100, has two RAW dependences, (1,3) and (2,3), since stores 1 and 2 only write a single byte. It can also be seen that store 4 has two WAW dependences: (1,4) and (2,4).

Finally, for clarity it is useful to use the terms *source* and *sink* to refer to instructions that are dependent. Given a dependence (A, B), where A and B are instances of a load or store, we define the sink and the source based on the order in which instructions A and B appear in the program defined sequential execution order. We will refer to the oldest in program order (encountered first) instruction as the *source*, while we will refer to the youngest in program order (encountered last) instruction as the *sink*. It should be understood that while for RAW dependences the terms source and sink also appropriately describe the flow of memory values, this is not so for the other dependence types. However, we will uniformly use these terms for all dependence types for clarity.

2.2 A Class of History-Based Memory Dependence Predictors

In this work we are interested in two types of memory dependence information which informally are (formal definitions are given in sections 2.3.6 and 2.3.7): (1) whether a load or a store has a dependence of a given type (i.e., the exact dependence is not important in this case, only its existence is), and (2) which exactly are the memory dependences a load or a store instance has (i.e., the exact load or store with which the dependences exist is important). As the discussion of the previous section serves to imply, the memory dependence information that interests us can be derived by inspecting the memory address stream of a program. However, and as it will become apparent during the description of the techniques presented in chapters 3 through 5, using address-based information to derive memory dependence information is not always an option or sometimes, it is not a desirable option. For example, we may wish to know whether a version created by a store will be read by a subsequent load (i.e., whether the store has a RAW dependence). However, at the time this information is required we may have not even seen a subsequent load, let alone a load that accesses the same memory location.

Instead of waiting to derive memory dependence information by address stream inspection, we may opt for a method that allows us to guess, preferably with high accuracy, the memory dependence information we require. This is exactly the function of a memory dependence predictor. A simple memory dependence predictor is one that always makes a predefined, hard-wired prediction. In our preceding paragraph example, such a predictor could for example, always predict that a store will experience a RAW dependence. Examples of such predictors for other types of program related information abound. For example, caches implicitly use such a predictor as all memory addresses accessed are placed into a typical data cache in hope that they will be soon referenced again. Static branch predictors are another example. Finally, naive memory dependence speculation (discussed in Section 3.2) implicitly uses such a predictor that always predicts that a load will not experience a RAW dependence with a preceding store within the current instruction window.

However, in this work we focus on a different class of predictors. There are two reasons: (1) often the prediction accuracy possible with a predictor that always responds with a predefined, hard-wired answer is not sufficient (most of the material of Chapter 3 is motivated by this observation about naive memory dependence speculation), and (2) in some cases we are interested in information for which there does not seem to be a reasonable method of providing a predefined prediction. The latter point is best understood if, for example, we consider applications that require prediction of the exact RAW memory dependences (i.e., load-store pairs).

In this work we focus on a class of history-based predictors which attempt to learn how the program behaves and use that information to make guesses about the desired memory dependence information. The example of Figure 2.2 is helpful in illustrating how a straightforward history-based memory dependence predictor of this kind might operate. Here we assume that we are interested in predicting whether a store has a RAW dependence with a subsequent load and which load that is. The first time a store is encountered our predictor has no information and for this reason cannot make an educated guess on whether a RAW dependence exists. However, what our predictor does in this case is wait to observe via the address space whether a dependence is experienced, and if so, which dependence that is. The actions that lead to the detection of the RAW dependence are as follows: first the store is encountered. Later on, when the store accesses memory, a record is made of the address it wrote to (action 1). This record is associated with the store instruction somehow (e.g., marked by the PC of the store). When later on, after the load has been encountered and once it accesses memory, its address is used to locate the record left by the store (action 2). At this point our predictor has successfully detected that the store instruction experienced a RAW dependence and which load this dependence was with. This information is recorded in a prediction structure (action 3). Later on, when another instance of the same store instruction is encountered, the prediction structure is inspected (action 4) and since a record is found for the particular store, our predictor can now guess that the same RAW dependence will be experienced again. If so desired, our predictor can later validate whether its guess was correct and change its prediction for subsequent instances of the same store. Provided that stores tend to experience the same memory dependence most of the time, the accuracy of our predictor will be high.

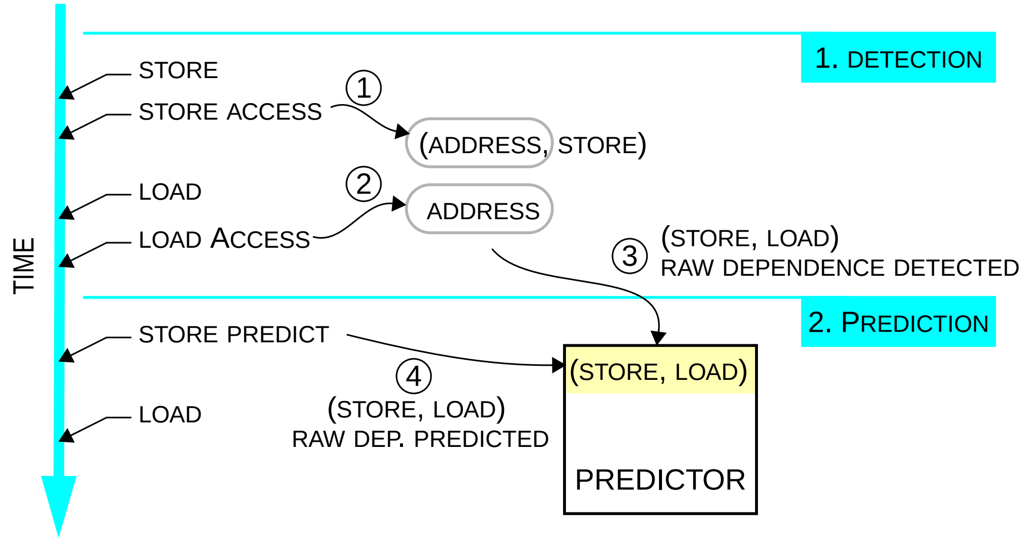
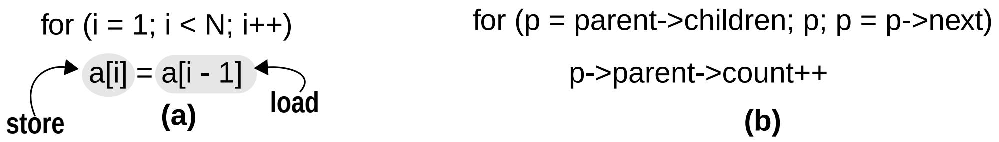


Figure 2.2: *Example illustrating the operation of history-based memory dependence predictors.*

Generally, history-based predictors rely on the assumption that past behavior is a strong indicator of future behavior. If this property does not hold then predictors of this kind will fail. However, as the results presented later in chapter serve to demonstrate typical programs do exhibit this kind of regularity in their memory dependence stream. The predictor we have presented in the previous paragraph is of the most straightforward type as it simply predicts that what happened last time will happen the next time around. Provided that events repeat with high probability, even such a simple predictor will be quite accurate. If higher prediction accuracy is required, we may opt for more sophisticated predictors that attempt to discover repeating patterns of events (e.g., [12, 99]).

Example code sequences whose memory dependences are a good match for a simple history-based predictor are shown in Figure 2.3. Each iteration of the loop of part (a), contains a load that reads a[i - 1] and a store that writes to a[i]. A RAW dependence exists between the load and the store of two consecutive iterations.

Similarly, every iteration of the loop of part (b) contains a load and a store which are used to read, increment and update the count variable. RAW dependences exist between each store and the load of the next iteration. Also WAW dependences exist between stores of successive iterations. The aforementioned RAW and WAW dependences are amenable to history based prediction as once observed they occur every time the corresponding instructions are encountered.



```
for (i = 1; i < N; i++)
    a[i] = a[i - 1]
```

```
for (p = parent->children; p; p = p->next)
    p->parent->count++
```

Figure 2.3: *Example code sequences that are amenable to history-based memory dependence prediction.*

For history-based memory dependence prediction to be possible and of practical use the following characteristics are desirable: (1) we should be able to build history, that is detect memory dependences, (2) we should be able to record the collected history using structures of reasonable size, and (3) the memory dependence attribute we wish to predict should exhibit sufficient regularity so that past behavior is a good indication of future behavior. A multitude of options exists on how to go about designing the specifics of a history-based memory dependence predictor. Weighting the appropriateness of each option is an exercise most meaningful given a target application. Accordingly, in this chapter, we restrict our attention to aspects of memory dependence behavior that seem relevant for most, if not all predictors of this type or that have implications on the sophistication required of the prediction mechanism. A description of these attributes along with a justification of why we include them is given at the beginning of the next section.

2.3 Memory Dependence Behavior Analysis

With a high-level understanding of what is involved in predicting memory dependence information we now proceed to characterize the memory dependence behavior of the SPEC95 programs. In this analysis we focus on those characteristics of memory dependence behavior that are more relevant from the perspective of the class of history-based predictors we described in the previous section and of the memory dependence information that we utilize in the techniques presented in chapters 3 through 5. In this section we first describe the types of memory dependence information that interest us and then proceed to list the attributes of memory dependence behavior we studied.

While there are certainly many different types of memory dependence information that might be useful, in this work we are interested in predicting two types of dependence information. Informally, these are: (1) whether a particular instruction has a dependence of a particular type or the *memory dependence status* of an instruction (a formal definition is given in Section 2.3.6), and (2) the set of dependences an instruction has, or the *memory dependence set* of an instruction (a formal definition is given in Section 2.3.7).

As we have seen, there are four types of memory dependences. For most of the analysis that follows we focus on RAW and RAR dependences. We do so, as these are the dependence types that we use for the bulk of the techniques we present in chapters 3 to 5. However, we do measure some of the WAR and WAW dependence characteristics of programs. Before presenting our findings in sections 2.3.2 through 2.3.7, we first list the metrics we used along with a justification of why we include them in Section 2.3.1.

2.3.1 Metrics and Justification

For history-based memory dependence prediction to be possible the following characteristics are desirable: (1) we should be able to build history, that is detect memory dependences, (2) we should be able to record the collected history, and (3) memory dependences should exhibit sufficient regularity so that past behavior is a good indication of future behavior. In this context, the following characteristics of memory dependence behavior are relevant:

1. What fraction of loads and stores experience what memory dependence types? This metric provides an indication of the potential coverage of any memory dependence prediction based technique. These measurements we present in Section 2.3.2.
2. Do we have to predict a single dependence or multiple dependences per store or load? This characteristic of memory dependences has ramifications on how a prediction mechanism will have to represent dependences and on the sophistication required of it. For example, if dependences are mostly one-on-one (i.e., each load or store instance has a single dependence), a direct representation of dependences may be practical. If however, multiple dependences have to be predicted per dynamic instance, other representations may be necessary. These measurements are presented in Section 2.3.3.
3. How large is the working set of loads and stores with dependences? Or, for how many loads and stores we have to record dependence history information in order to be able to make predictions for a desired fraction of loads and stores? If high coverage is desired, a prediction mechanism will have to record information for those *instructions* that experience dependences. The working set size of those instructions provides an indication of the amount of resources that will be necessary to record the relevant information. These measurements we present in Section 2.3.4.
4. What size structures are required to detect a desired level of memory dependence activity? Detecting memory dependences is required to build the history necessary for prediction purposes. Many options exist in how to go about detecting dependences. For the purposes of this study we consider a straightforward, yet effective way which amounts to keeping a record of the last *n data addresses* touched by the program. A detailed description of this metric is given in Section 2.3.5 along with the measurements. (Note that item 3 is a property of the instructions, while this metric is a property of the memory addresses accessed.)
5. Whether sufficient regularity exists in the dynamic behavior of the attribute of memory dependences we want to predict. Since we are interested in *memory dependence status* and in *memory dependence set* prediction (definitions are given in sections 2.3.6 and 2.3.7) we restrict our attention to these two attributes and present locality measurements in sections 2.3.6 and 2.3.7 respectively. Informally, locality refers to the likelihood that the same memory dependence status or memory dependences are encountered in two consecutive executions of the same static instruction. A formal definition is given at the beginning of each section.

2.3.2 Memory Dependence Characterization

We start our analysis by measuring the frequency of memory dependences. We also characterize dependences by how they are distributed in memory segment terms (data, heap, stack) and by how far apart are the dependent instructions in the execution stream.

First, we measure the percentage of executed loads and stores that experience RAW, RAR, WAR and RAW, WAR, WAW dependences respectively. These results are shown in Figures 2.4 and 2.5 for loads and stores respectively. For loads we also show the fraction that experience both RAR and RAW dependences at the same time as in the methods we present in Chapter 4 preference will be given to the RAW dependences in such cases. Before we comment on these results it is important to note that accounting correctly for RAR and WAW dependences requires distinguishing which side of a dependence (sink or source) a load or a store appears at. The reason is that a particular load (store) may have a RAR (WAW) dependence with loads (stores) the precede or that follow it. Ultimately, we could distinguish between backward and forward dependences by accounting for the corresponding dependences separately. However, we choose to measure those loads that have a dependence with a preceding load, and those stores that overwrite a memory location written by a preceding store (i.e., we look only at backward RAR and WAW dependences). It is this definition of RAR dependences that is useful for the techniques we present in Chapter 4. With this definition of RAR dependences we do not count a RAR dependence on the first in program order load that accesses a memory location that other subsequent loads also access. Similarly, we do not count a WAW dependence on the first store that writes to a particular memory location that later gets overwritten.

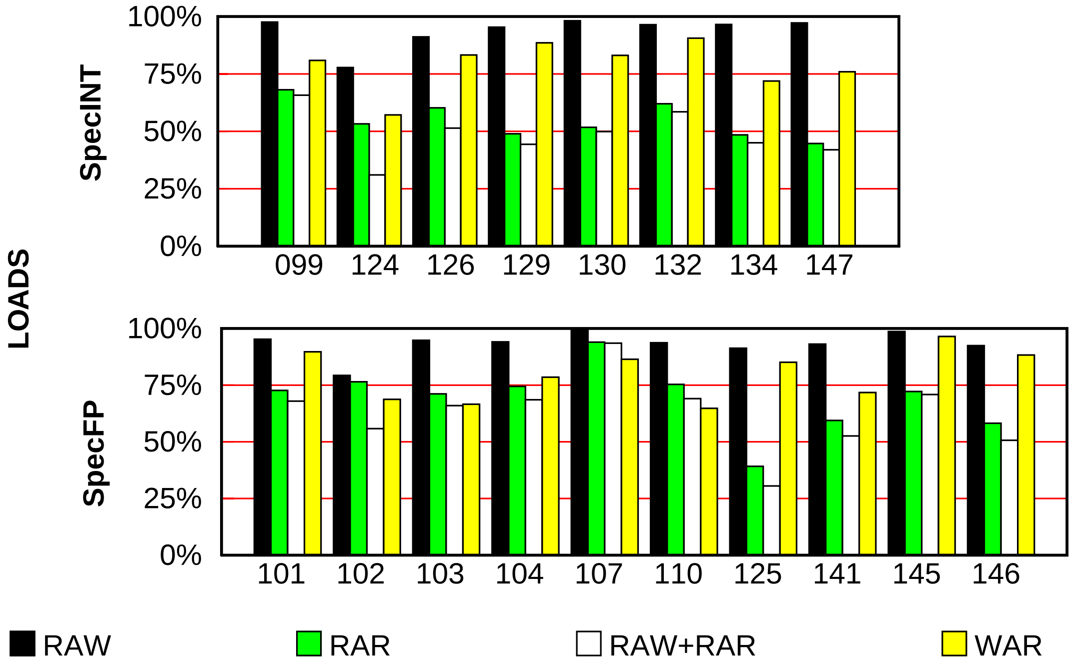


Figure 2.4: *Memory Dependence Breakdown - Loads. Shown is the fraction (Y-axis) of all executed loads that experience a memory dependence of the given type.*

Focusing on Figure 2.4 and specifically on RAW dependences we can observe that the vast majority of executed loads read a value written by a preceding store through a RAW dependence. The rest of loads access a value that was produced outside of the scope of the program. This data was either loaded in memory before the program execution was initiated or was the result of a system call. Many loads have also WAR dependences (above 70% for all programs, except 124.m88ksim). In conjunction with the high percentage of RAW dependences, this observation suggests that memory is often used to hold values that are written, read and later overwritten.

Greater variation is exhibited by RAR dependences. For most programs around half of all loads read a value previously read by a preceding load. At the two extremes are 107.mgrid with about 93% of loads having a RAR dependence and 125.turb3d with 40% of load having a RAR dependence. Most of these loads and in some cases all of them, read a value that was also written by a store as it can be seen by comparing the RAR and RAW+RAR bars. This observation suggests that some stored values are read at least twice. In those cases where that RAR bar is above 50%, some memory values are read more than two times (since we do not include the first load in our RAR measurement).

Figure 2.5 shows a breakdown of the dependences experienced by stores. For most programs nearly all stores experience RAW dependences suggesting that these values might be used for further computations within the program itself (it might also be that this traffic corresponds to callee-saved registers that contain dead values). However, some stored values are never read by a load. This phenomenon is most acute in the case of 126.gcc and 147.vortex where nearly half of all stored values are never used. Three possible causes of this phenomenon are: (1) Some of these stored values constitute program output. (2) Others are the result of ambiguous dependences. In these cases, a value resides both in a register and in memory. As a result every time the register is updated its copy in memory is also updated, however the memory copy is read by a load only when undetectable statically memory dependences exist. (3) However, in some cases this phenomenon is the result of the algorithms used; stored values are simply overwritten by another store before a load has had a chance to read them. (For example gcc uses its own dynamic memory allocator, implementing multiple stacks of objects. This allocator uses several global flags that relate to the current stack. These flags are always updated when the current stack changes. However, these flags are not always accessed before the next stack change occurs.) The latter observation (point 3) is supported by the frequency of WAW dependences, which are typically more frequent than RAW dependences. Finally, it should be noted that most stores also overwrite a value previously read by a load (WAR dependences). This result confirms our previous observation that a large fraction of memory values are written, read and overwritten repeatedly.

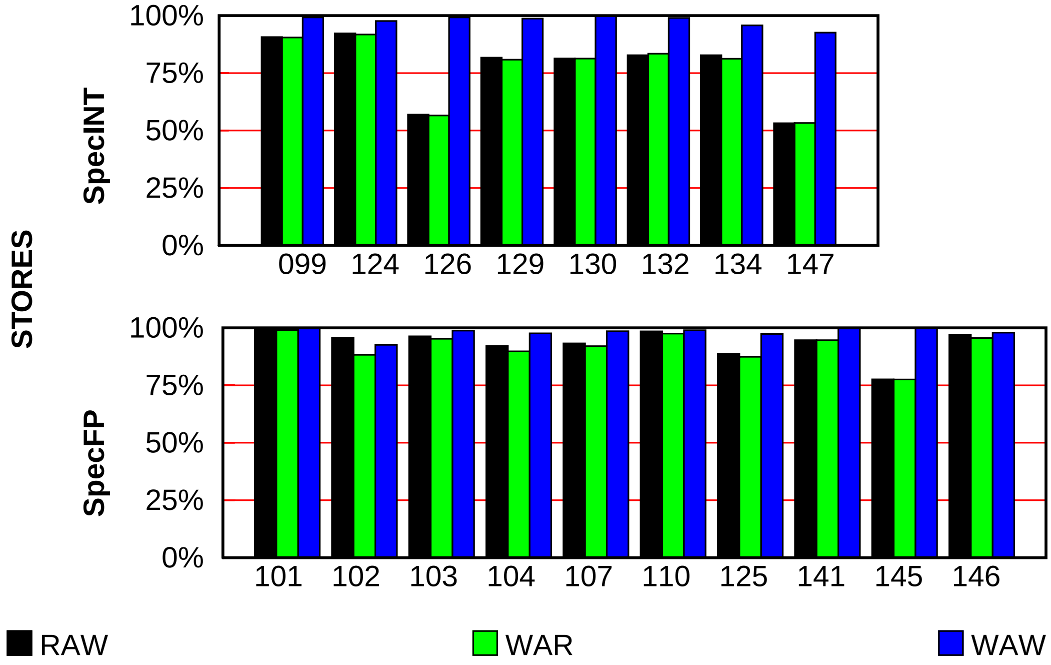


***Figure 2.5:** Memory Dependence Breakdown - Stores. Shown is the fraction (Y-axis) of all executed stores that experience a memory dependence of the given type.*

2.3.2.1 Address Space Distribution of Memory Dependences

In this section we present a breakdown of memory dependences in terms of the address space through which they occur. Specifically, we classify dependences into those that occur through the data, the heap and the stack segments. These measurements are interesting primarily as reference material and secondarily in providing indications on: (1) whether much of the dependence activity results from programming conventions (stack), and (2) whether most of the activity results from the data and the stack segments which are typically easier to disambiguate at compile time. Table 2.1 reports these measurements. The percentages shown are over all loads (or stores) that experience the particular type of dependence. Note that since we use rounding, the sum of all three segments per dependence type does not necessarily add up to 100%. It can be seen that there is no clear trend in how memory dependences are distributed. While a significant fraction of dependences occurs via the stack, much of the dependence activity, and in some case most of the activity, takes place through either the data or the heap segment. We should note that the occasional appearance of heap memory dependences in floating programs is a side-effect of the use of the FORTRAN to C translator and its implementation of FORTRAN built-in functions.

		RAW			RAR			WAR		
		DATA	HEAP	STACK	DATA	HEAP	STACK	DATA	HEAP	STACK
LOADS	***099***	72.5%	0.0%	27.5%	89.0%	0.0%	11.0%	66.8%	0.0%	33.2%
	124	60.6%	13.6%	25.9%	81.0%	18.9%	0.1%	63.7%	1.2%	35.2%
	126	11.6%	33.6%	54.8%	28.7%	47.4%	23.9%	9.1%	31.0%	60.0%
	129	92.9%	0.0%	7.1%	100.0%	0.0%	0.0%	92.4%	0.0%	7.6%
	130	18.8%	45.2%	36.1%	24.5%	70.9%	4.6%	14.3%	43.4%	42.2%
	132	1.1%	71.5%	27.4%	6.4%	84.1%	9.5%	0.7%	70.2%	29.1%
	134	11.6%	51.1%	37.4%	22.6%	70.6%	6.8%	11.2%	38.7%	50.1%
	147	7.1%	31.4%	61.5%	19.8%	45.5%	34.7%	2.9%	18.3%	78.8%
	101	0.6%	0.1%	99.2%	6.9%	0.1%	93.0%	0.6%	0.1%	99.2%
	102	91.7%	0.0%	8.3%	100.0%	0.0%	0.0%	90.4%	0.0%	9.6%
	103	38.3%	1.2%	60.5%	54.8%	0.8%	44.3%	15.9%	1.8%	82.3%
	104	90.0%	0.9%	9.1%	93.9%	0.5%	5.6%	88.0%	1.0%	11.0%
	107	87.6%	0.0%	12.4%	87.8%	0.0%	12.2%	86.5%	0.0%	13.5%
	110	72.3%	0.0%	27.7%	79.1%	0.0%	20.9%	59.9%	0.0%	40.1%
	125	21.7%	0.0%	78.3%	46.2%	0.0%	53.8%	16.1%	0.0%	83.9%
	141	66.2%	0.1%	33.6%	71.8%	0.0%	28.2%	60.8%	0.2%	39.0%
	145	47.5%	0.0%	52.5%	57.7%	0.0%	42.3%	46.9%	0.0%	53.1%
	146	88.5%	0.0%	11.5%	93.0%	0.0%	7.0%	87.9%	0.0%	12.1%

Table 2.1: *Address space distribution of memory dependences.*

		RAW			WAW			WAR		
		DATA	HEAP	STACK	DATA	HEAP	STACK	DATA	HEAP	STACK
STORES	*099*	39.2%	0.0%	60.8%	39.3%	0.0%	60.7%	39.1%	0.0%	60.9%
	124	55.6%	1.6%	42.7%	56.8%	1.9%	41.3%	55.9%	1.1%	43.0%
	126	5.3%	6.0%	88.7%	11.8%	5.2%	83.0%	5.2%	5.1%	89.7%
	129	86.8%	0.0%	13.2%	89.0%	0.0%	11.0%	86.6%	0.0%	13.4%
	130	15.7%	15.8%	68.5%	14.2%	21.2%	64.6%	15.7%	15.8%	68.5%
	132	1.0%	47.8%	51.2%	0.9%	55.1%	44.1%	0.9%	48.1%	50.9%
	134	6.7%	32.0%	61.3%	11.0%	34.2%	54.8%	6.8%	30.6%	62.6%
	147	0.8%	19.1%	80.1%	9.4%	22.5%	68.0%	0.8%	16.6%	82.6%
	101	0.8%	0.5%	98.7%	1.1%	0.5%	98.4%	0.8%	0.5%	98.7%
	102	72.0%	0.0%	28.0%	66.5%	0.0%	33.5%	69.6%	0.0%	30.4%
	103	6.5%	4.1%	89.5%	6.9%	4.0%	89.1%	6.2%	4.1%	89.6%
	104	79.5%	3.2%	17.3%	80.4%	3.0%	16.6%	79.2%	3.3%	17.5%
	107	84.6%	0.0%	15.4%	85.2%	0.1%	14.7%	84.5%	0.0%	15.5%
	110	58.5%	0.0%	41.5%	57.5%	0.0%	42.5%	58.1%	0.0%	41.9%
	125	17.1%	0.0%	82.9%	14.4%	0.0%	85.6%	15.9%	0.0%	84.1%
	141	63.7%	0.3%	36.0%	62.4%	0.3%	37.3%	63.7%	0.3%	36.0%
	145	23.4%	0.0%	76.6%	38.0%	0.0%	61.9%	23.4%	0.0%	76.6%
	146	84.2%	0.0%	15.8%	83.4%	0.0%	16.6%	84.0%	0.0%	16.0%

Table 2.1: *Address space distribution of memory dependences.*

For reference we also present a breakdown of all load and stores accesses in terms of the address space being accessed. This data is presented in Table 2.2. Note that in these results we measure fractions over all executed loads and stores (in Table 2.1, we measured fractions over all executed load or stores that experienced a dependence of a given type). To correlate the results of Table 2.2 with those of Table 2.1, the frequency of the corresponding memory dependences (as reported in figures 2.4 and 2.5) must be taken into account.

2.3.2.2 Dynamic Instruction Distance Distribution

We have seen that the majority of memory instructions experience memory dependences and that the distribution of these dependences over the stack, heap and data segments exhibits great variation. In this section we measure how far apart are the instructions connected with dependences. We use *dynamic instruction distance* as our metric. We define the dynamic instruction distance of a dependence to be the number of instructions executed between the two dependent instructions. This metric provides an indication of the

	LOADS			STORES		
	DATA	HEAP	STACK	DATA	HEAP	STACK
099.go	73.2%	0.0%	26.8%	39.8%	0.0%	60.2%
124.m88ksim	69.3%	10.6%	20.1%	55.5%	4.2%	40.3%
126.gcc	19.3%	20.7%	50.0%	11.70%	5.8%	82.5%
129.compress	93.2%	0.0%	6.7%	89.2%	0.0%	10.8%
130.li	20.2%	44.3%	35.4%	14.2%	21.3%	64.5%
132.ijpeg	4.4%	69.1%	26.5%	0.9%	55.5%	43.6%
134.perl	14.4%	49.6%	36.1%	10.5%	37.0%	52.4%
147.vortex	9.2%	40.0%	59.8%	8.7%	28.2%	63.0%
101.tomcatv	5.2%	0.2%	94.5%	1.1%	0.5%	98.4%
102.swim	93.4%	0.0%	6.6%	69.0%	0.0%	31.0%
103.su2cor	40.9%	1.8%	57.3%	7.2%	3.9%	88.8%
104.hydro2d	90.2%	1.2%	8.6%	80.8%	3.0%	16.2%
107.mgrid	87.6%	0.0%	12.4%	85.4%	0.1%	14.5%
110.applu	74.0%	0.0%	26.0%	57.9%	0.0%	42.1%
125.turb3d	28.5%	0.0%	71.5%	16.7%	1.5%	90.0%
141.apsi	68.5%	0.2%	31.3%	62.3%	0.4%	37.2%
145.fpppp	48.2%	0.0%	51.8%	38.1%	0.0%	61.9%
146.wave5	89.3%	0.0%	10.7%	83.7%	0.0%	86.1%

Table 2.2: *Address space distribution of load and store accesses. Fractions are reported over all executed loads or stores.*

number of dependences that are going to be observed from within the instruction window of an out-of-order processor. We will use this information during Chapter 3 where we will be concerned with methods of extracting and exploiting load-store parallelism. These measurements are also useful in estimating the fraction of memory dependence activity that can be detected using existing load/store execution mechanisms found in most modern, dynamically scheduled ILP processors (i.e., load/store queues).

Figure 2.1 shows the cumulative distribution of dynamic instruction distances for RAW and RAR dependences. Samples are taken at the following distances: 8, 64, 512, 4K and 32K. In all four cases, we measure distances at the point of the dependent load (sink). The percentages shown are over all loads that experience dependences of the corresponding type. As we have discussed, a given dynamic load instance may experience multiple RAW and RAR dependences. We account for multiple dependences by including distance measurements for the closest and the furthest away source instructions (a store for RAW dependences, a load for RAR dependences). In the case of RAW dependences, virtually no difference was observed suggesting that rarely loads experience multiple RAW dependences or that whenever they do the source stores are clustered very close to each other in the execution stream. For this reason we only account for the closest source store in our measurements. Loads with RAR dependences however, were sensitive to the selection of their source load. For this reason we do include measurements for both the closest and the furthest away possible source loads.

It can be observed that a noticeable fraction of loads experiences a RAW dependence with a store that is more than 32K instructions away. This phenomenon is more pronounced for the floating point programs where the common case is that very few loads have RAW dependences that are visible within a window of even 32K instructions. However, floating point programs also exhibit great variation. At one extreme are

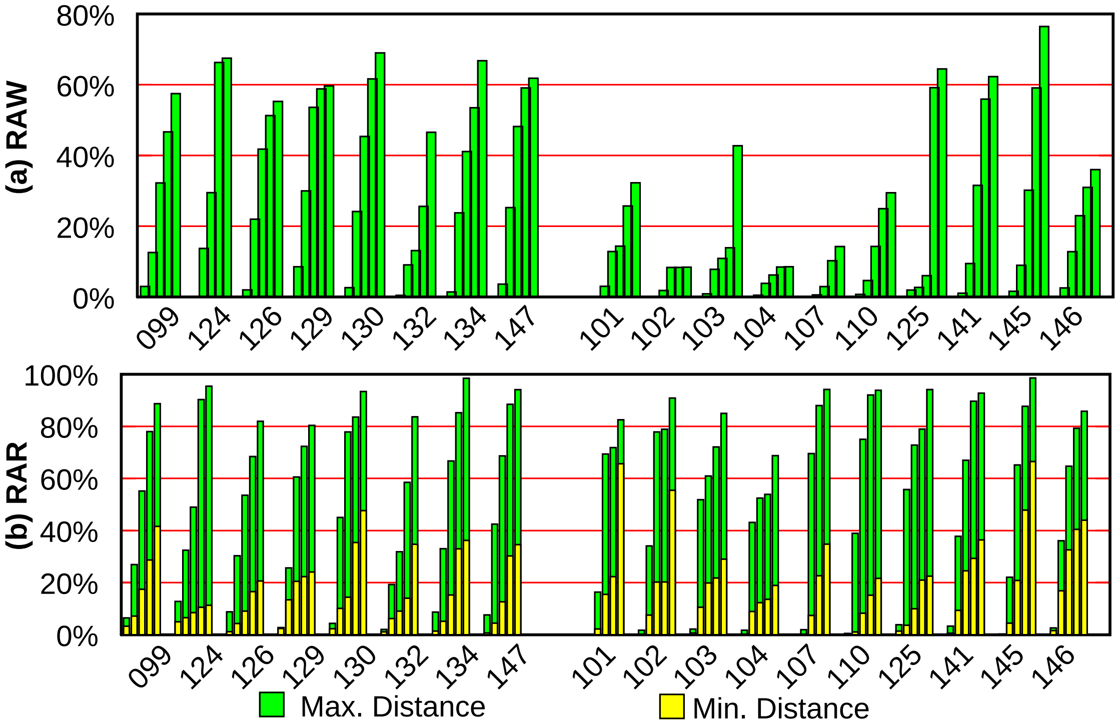


Figure 2.1: *Cumulative dependence dynamic instruction distance distribution. Samples are taken at 8, 64, 512, 4K and 32K distances. (a) RAW dependences. (b) RAR dependences.*

programs like 102.swim and 104.hydro2d were roughly, only 6% and 8% respectively of loads with RAW dependences experience a dependence within a 32K instruction window. At the other extreme are programs like 145.fpppp and 141.apsi that demonstrate a steady increase of RAW dependences as longer distances are considered. We take a closer look at those programs in order to better understand their behavior. In the case of 102.swim the RAW dependences that are not visible in their vast majority are accesses to array elements. 102.swim repeats a series of loops. Each of these loops reads values from one set of arrays, producing values for a different set of arrays. No recurrences exist in these loops. As a result, the RAW dependences that are observed are mostly across different loops. Given that these loops iterate many times, these RAW dependences are spread over many instructions. For 102.swim, the few RAW dependences that are visible within the instruction window range shown, are mostly the result of accesses to induction or other global variables that were not register allocated. 104.hydro2d also experiences very few RAW dependences within the measured range. It is similar to 102.swim in that its computation proceeds in a series of loops that read a different set of arrays than they update. This program however has loops with recurrences. However, these recurrences are on non-array variables and are mostly register allocated. The loops of this program also access loop-invariant values that are calculated just before the loop starts. Some of these variables are not register allocated or are occasionally spilled to the stack, giving rise to the majority of the shorter distance RAW dependences. 145.fpppp on the other hand spends most of its time in loop whose iterations consists of a series of dependent calculations. These calculations use either stack allocated variables (often of aggregate data types) or global data segment allocated ones. Because these values are used quickly after they are updated, a significant number of RAW dependences are observed even in short distances.

It is also interesting to observe that in the range of modern instruction windows (less than 64 instructions for most processors) most loads do not experience RAW dependences. However, for most programs there is a rapid increase in RAW dependences as we move to larger distances. These results suggest that as processors become able to establish larger windows, the probability of having to access a value written by a preceding store within the same window will increase. We will make use of this result in Chapter 3, where we will be interested in exploiting load/store parallelism.

Focusing on RAR dependences we can observe that great variation is exhibited in the distances of the closest and the furthest away possible source loads. When the furthest possible source loads are considered, results are very similar to those for RAW dependences. However, when the closest possible loads are considered we observe significantly higher levels of RAR dependences even in short distances. However, even then there is a large fraction of RAR dependences that are not visible from within the range of a typical, current generation instruction windows. We will use this result in Chapter 4, to motivate the use of distant RAR dependences in providing faster access to memory values.

2.3.3 Memory Dependence Shape Characterization

We next consider the shape memory dependences take. For the purposes of this study we define *shape* to be the number of dependences of a given type experienced by a load or a store. We use two metrics: (1) *instance dependence set size* which is the number of dependences of a given type experienced by a dynamic instance of a load or a store., and (2) *aggregate dependence set size*, which is the number of distinct static dependences of a given type that are experienced by a static load or a store when all dependences observed by all dynamic instances of the particular instruction are taken into account. Informally, the instance dependence set size of a load or a store provides an indication of the number of dependences we will have to predict if in our prediction we care about the exact dependences loads and stores have. The aggregate dependence set size measurements do not directly pertain to memory dependence prediction and is presented only to provide additional insight on the memory dependence behavior of programs.

Figure 2.2, part (b) shows an example instruction sequence that is helpful in clarifying the difference between the two metrics (this sequence can be generated by an execution of the loop of part (a)). The instance of store st1 at line 1 has a RAW instance dependence set size of 3 as it experiences three RAW dependences with the loads of lines 2, 3 and 4. Each of these loads has a RAW instance dependence set of 1. The instance of store st1 at line 5 has a RAW instance dependence set size of 1 as it experiences a single RAW dependence with the load of line 6. The RAW aggregate dependence set size of st1 is 4 as its dynamic instances (shown at lines 1 and 5) overall experienced 4 distinct static RAW dependences each with one of the 4 static loads shown (ld1 to ld4). Note that the aggregate dependence set size represents the number of distinct static dependences experienced dynamically by a given load or store for a particular run of the program. It is not the number of all memory dependences that might appear if all possible execution paths are taken into account.

The shape memory dependences take has also implications on their representation. This is most relevant for memory dependence predictors that attempt to predict the exact dependences an instruction has. For example, if a store instance experiences multiple RAW dependences (i.e., multiple loads are reading the value written by the store, e.g., store at line 1 of Figure 2.2, part (b)) then a memory dependence predictor will have to somehow predict multiple dependences for that store. In general, we have noted that there are two reasons why a load or a store may experience multiple dependences at any given instance: (1) a memory value may be read multiple times after it has been written or before it is overwritten by a store, and (2) loads and stores may be manipulating different data types.

In the sections that follow we first present measurements on the instance dependence set size (Section 2.3.3.1) and then on the aggregate dependence set size (Section 2.3.3.2).

```
for i = 100 to 101
    st1 M(i)
      if (i == 100)
             ld1 M(i)
             ld2 M(i)
             ld3 M(i)
       else
             ld4 M(i)
```

(a)

```
1: st1 M(100)
2: ld1  M(100)
3: ld2  M(100)
4: ld3 M(100)
5: st1 M(101)
6: ld4 M(101)
```

(b)

Figure 2.2: *Example illustrating the difference between static dependence set size and instance dependence set size.*

2.3.3.1 Instance Dependence Set Size

In this section we present measurements of the instance dependence set of instructions. We show results for both the source and sink instructions of each dependence type. The results of these experiments are shown in Figure 2.3 where we report the cumulative distribution of the dynamic instance dependence set per dependence type and separately for the source and the sink instructions. Fractions are reported over all loads or stores (depending on the dependence type and on whether the measurements apply to the source or the sink instructions) that experienced a memory dependence of the given type. For example, in part (a) we report the instance dependence set size distribution as a fraction of all stores (source) and of all loads (sink) that experienced a RAW dependence.

Focusing first on RAW dependences (part (a)), it can be seen that virtually all loads (sink) observe a single producing store for most programs. Only 126.gcc, 132.ijpeg, 134.perl and 147.vortex exhibit loads that experience multiple producing stores. Even so, these loads represent a very small fraction of all loads with RAW dependences (less then 4% in most cases). As expected, loads cannot experience more than 4 producing stores per instance (the largest data type is 4 bytes while the smallest is one byte). Stores with RAW dependences (source) exhibit slightly different behavior. For the integer codes and for the most part, stores see a single load consumer. However, a noticeable fraction sees more than one consuming load, a result suggesting that stored values are often read more than once. This latter phenomenon is more pronounced for the floating point programs. These result suggest, that a mechanism that represents RAW dependences explicitly (i.e., by enumerating all consuming loads) will have to represent multiple dependences per instance of many stores.

The sink loads of RAR dependences for the most part observe a single source load. Only for 124.m88ksim, 126.gcc and 132.ijpeg is there a noticeable fraction of loads that see more than one source loads as a result of accessing different data types. The source loads of RAR dependences on the other side see more than one sink loads, suggesting that after a memory value is read once, it is often read again, multiple times. As it was the case with RAW dependences, a mechanism that represents RAR dependences explicitly will have to represent multiple RAR dependences per instance of source loads.

The dynamic shape of WAR dependences follows closely that of RAW dependences. Source loads see, for the most part, a single sink store that overwrites them, while sink stores typically overwrite more than one load. Finally, WAW dependences are for the most part one-on-one. While in some programs there are either source or sink stores that experience multiple WAW dependences, these represent a very small fraction of all stores with WAW dependences (less than 2% in all cases).

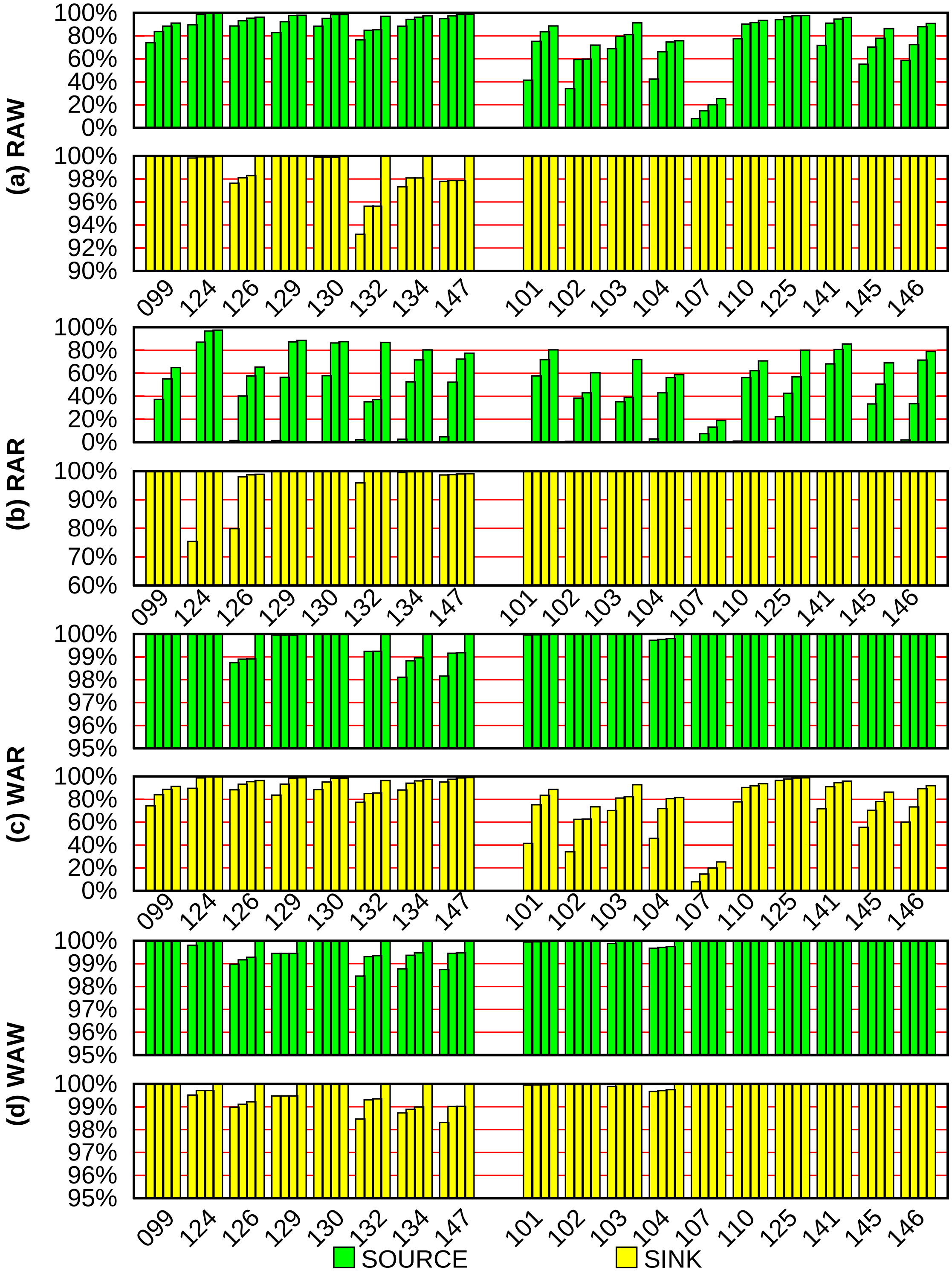


Figure 2.3: *Shape of memory dependences: cumulative dynamic instance dependence set distribution.*

2.3.3.2 Aggregate Dependence Set Size

In this section we consider the aggregate dependence set size of loads and stores. While these measurements are not pertinent to dynamic history-based memory dependence prediction, they provide indications on what would be necessary had we attempted to represent and convey dependence set information statically. For this purpose we measure the *aggregate dependence set size* of loads and stores which we defined earlier. Figure 2.4 shows the cumulative distribution of aggregate dependence set size per dependence type. We consider both source and sink instructions in this experiment and focus on RAW and RAR dependences. The dependence set size range shown is 1 to 4 dependences. The Y axis reports the fraction of dynamic instructions that experience the same dependence type that would have been covered if we considered dynamic instances of only those static instructions that have static dependence set sizes less than or equal to the value of the X-axis.

We can observe that a relatively large fraction of sink loads observe a single static dependence. However, in most cases the majority of sink loads have more than one static dependence. In some cases a significant fraction of loads have more than 4 static dependences. Aggregate memory dependence sets are relatively higher for the source loads and stores. These results may seem discouraging for memory dependence prediction. However, we note that as we will later demonstrate, the working set of memory dependences observed per static load or store is relatively small. This phenomenon will allows us to predict memory dependences without having to record all static dependences per load or store.

2.3.4 Working Set of Memory Dependences

History-based memory dependence prediction requires associating dependence history information with the relevant instructions. In this context, an important consideration is whether we could expect to store this information using structures of reasonable size. While determining the exact size of these structures is possible only given a specific prediction mechanism, it is desirable to define a metric that provides a rough estimate of the amount of resources that will be required. For this purpose, we provide measurements of the working set size of instructions with dependences of a given type. We do so by measuring the probability that a load or a store that experiences a dependence of a given type is among the last *n* loads or stores that experienced a dependence of the same type (another way of viewing this metric is as the hit rate of a fully-associative load/store cache with LRU replacement). A relatively high probability provides an indication that the working set is less than *n*. The results of these experiments are shown in Figure 2.5 and for the following values of *n*: 16, 256, 1K and 4K.

It can be seen that even when we consider just the last 16 instructions, we can capture a significant fraction of memory dependence activity. Coverage increases sharply as we consider larger values of *n* and approaches 100% when *n* is 4K. This result is especially encouraging as it applies even to those programs that have relatively large instruction working sets (e.g., 126.gcc and 147.vortex). The only exception is 145.fpppp whose working set of RAW dependences is large enough to demonstrate a probability of approximately 0.7 (70%) even when *n* is 4K. 145.fpppp spends most of its time in loop iterations whose static size is relatively large (several thousands of instructions). Most of these instructions are floating point loads and stores which are used to read and modify a rather large number of stack allocated variables. The code used to update these variables consists of a different load-store pair per variable. Since in most cases each of these variables is updated once per iteration, a large number of dependences is encountered until the next dynamic instance of the same static dependence.

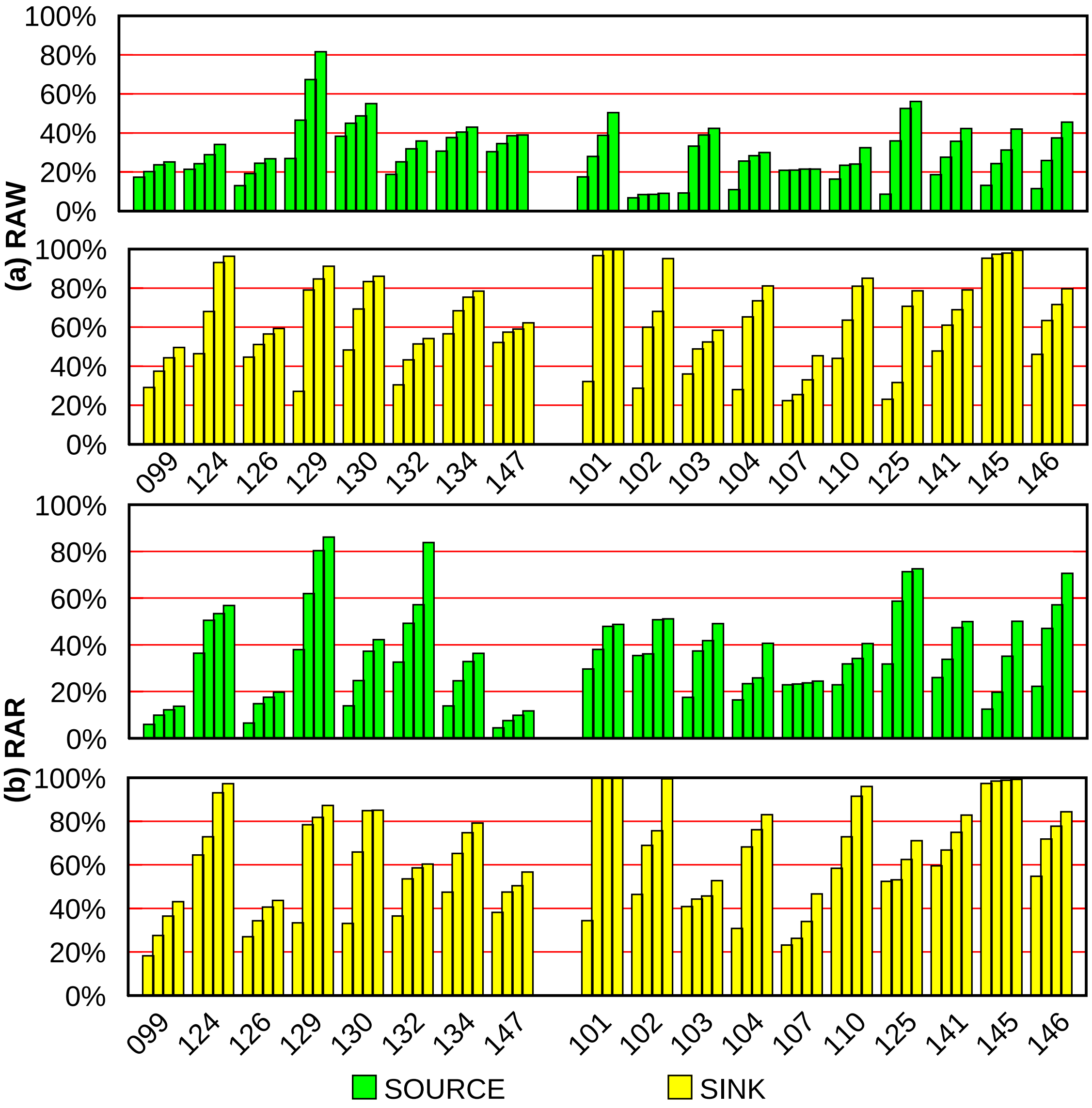


Figure 2.4: *Static memory dependence set size cumulative distribution. (a) Read-after-Write dependences. (b) Read-after-Read dependences. Range shown is 1 to 4 dependences. Y axis reports fractions over all executed instructions that experience the particular dependence type.*

The results of these experiments suggest that prediction structures with a reasonable number of entries should be sufficient in covering most of the memory dependence activity (i.e., recording memory dependence information for the majority of loads and stores with memory dependences).

2.3.5 Capturing Memory Dependence Activity

As we discussed in Section 2.2, a history-based memory dependence predictor requires a memory dependence detection mechanism. Such a mechanism records a number of recent accesses and if necessary, the identities of the instructions that performed these accesses. The number of accesses that can be recorded

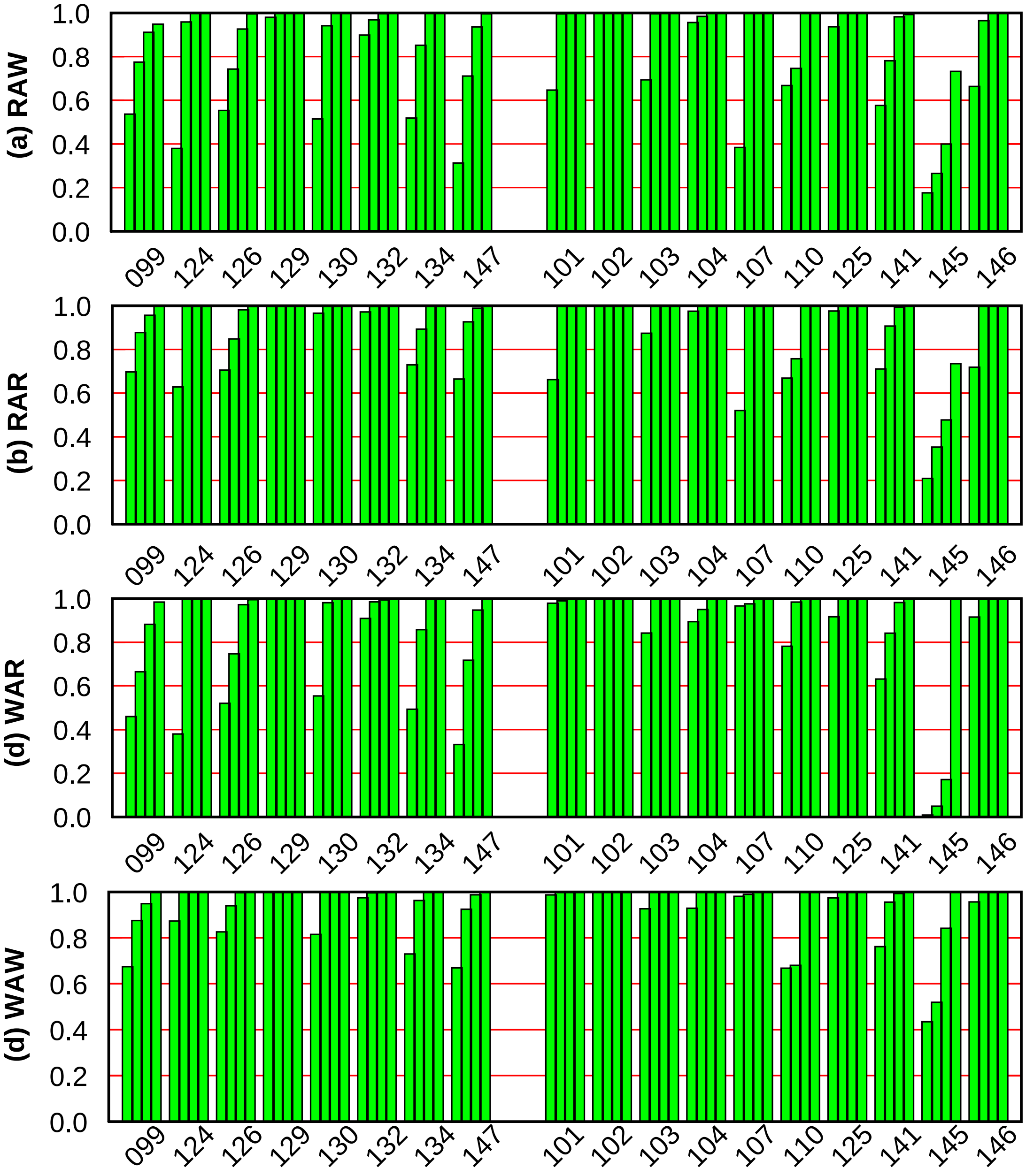


Figure 2.5: *Probability that a store or a load instruction that has a dependence has been among the* ***n*** *most recent instructions that experienced a dependence of the same type.* ***n*** *values shown are: 16, 256, 1K and 4K (left to right).*

sets a rough upper bound on the fraction of memory dependence activity that is visible to the predictor. We have seen that relying on mechanisms bound by the number of instruction visible from within the instruction windows of modern ILP processors does not allow us to capture much of the RAW and RAR memory dependence activity (Section 2.3.2.2). To quantify the amount of information that is necessary to capture a desired level of memory dependence activity, in this section we measure the *address distance* of memory dependences. We define *address distance* as the number of unique addresses (word granularity) accessed in between two dependent instructions. Recording information about these addresses should allow us to collect the desired dependence information. This metric is occasionally pessimistic since we do not need to record all memory accesses to detect certain kinds of dependences. For example, to detect RAW dependences we need to record only store accesses. This metric can also be optimistic when multiple dependences per instance of a source or of a sink instruction have to be detected explicitly. However, we note that in the applications of memory dependence prediction we present in latter chapters, we do not require this functionality. The address distance distribution is also useful in determining how much storage is required to keep the data values touched. This information will be proven useful in chapters 4 and 5 where we use memory dependence prediction to introduce novel memory value accessing and storage management mechanisms.

We present measurements for loads and stores separately, in figures 2.6 and 2.7 respectively. For this measurements we take samples at address distances of 16, 256, 4K and 64K address windows. Also shown (right-most sample) is the fraction of loads or stores that have dependences given an infinite address window (Section 2.3.2).

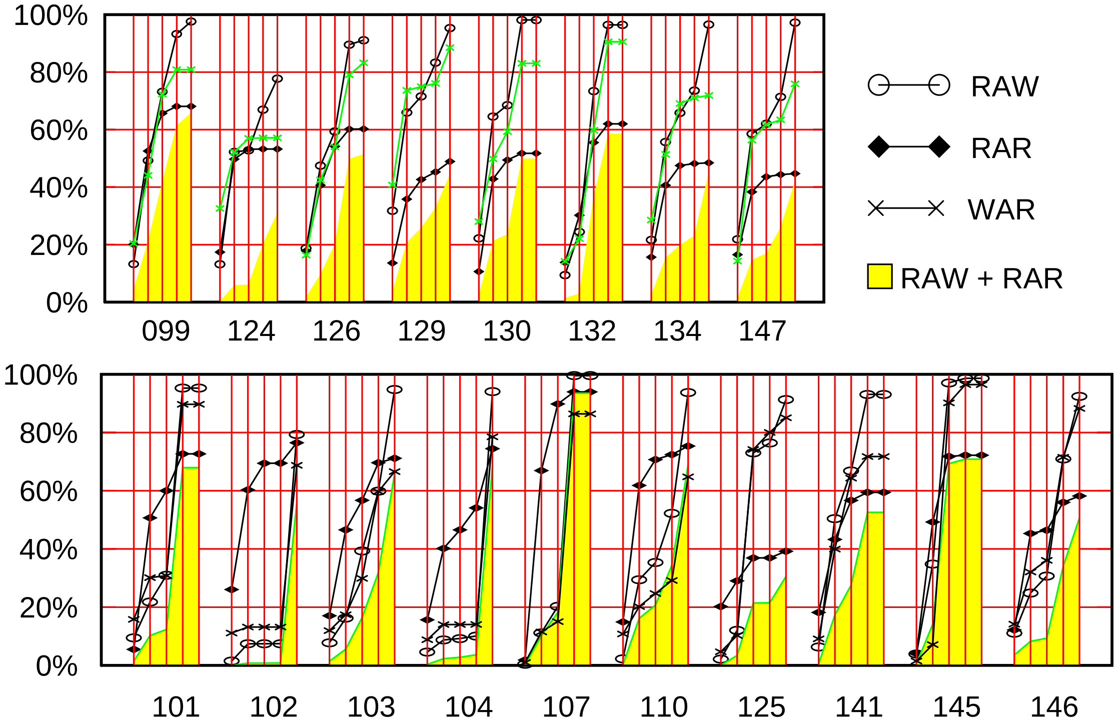


Figure 2.6: *Cumulative address-distance distribution of memory dependences as seen by loads. Samples are taken at address distances of 16, 256, 4K and 64K. Last value shown (right-most) reports all dependences independently of distance.*

Focusing first on the loads, we can observe that in most cases a 64K address window is sufficient to capture most of the dynamic memory dependences of a given type. This is not true for 102.swim and 104.hydro2d where most of the RAW and WAR dependences escape detection even when a 64K address window is used. We can also observe that the integer programs exhibit a relatively high percentage of RAW dependences that are detectable even with a 256-entry address window. In fact, it is typical to observe more than 50% of all RAW dependences within this limit. This phenomenon is not observed in most of the floating point programs where very few RAW dependences are detected even with a 4K address window. In contrast, RAR dependences appear to be more frequent in floating point programs especially when the smaller address windows are used. We can also observe that in most cases, the majority of loads with RAR dependences detected also have RAW dependences detected for the larger address windows. However, when the smaller address windows are used a considerable fraction of those RAW dependences escape detection. We will use this observation to streamline accessing memory values in Chapter 4. All the aforementioned observations are for the given input data set, result may vary for different inputs. However, in Chapter 4 we will show that as far a memory dependence prediction is concerned, using a different input data set does not affect considerably the fraction of loads and stores that have RAW or RAR dependences detected and predicted correctly within relatively small address windows.

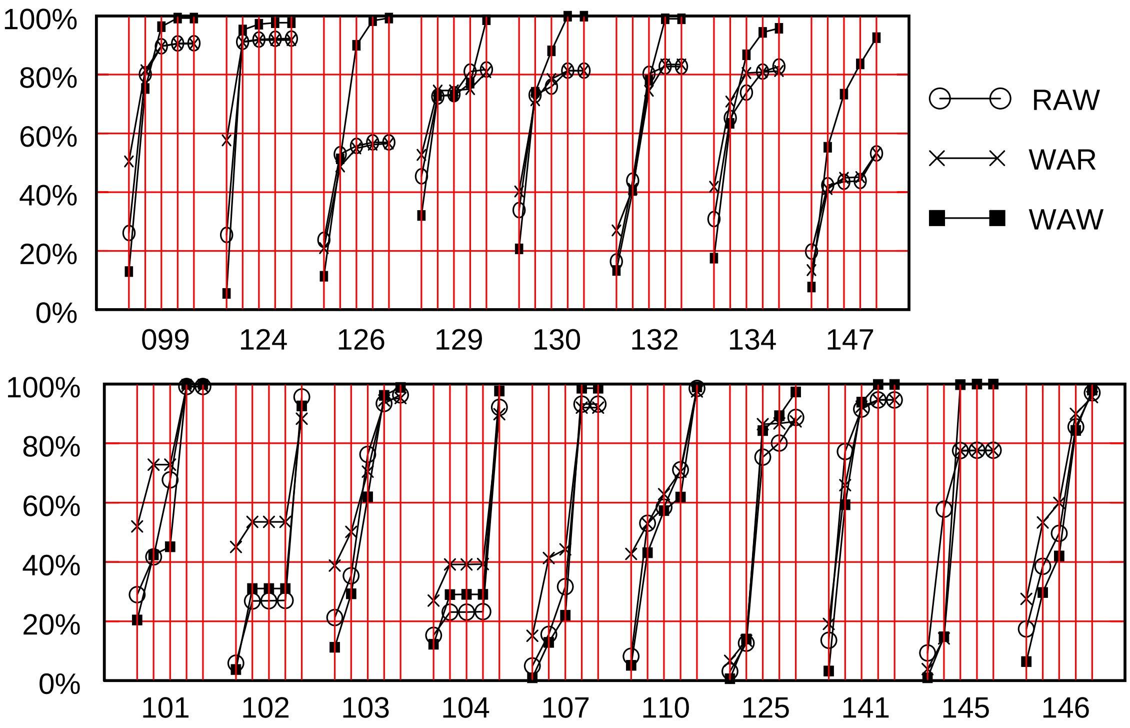


Figure 2.7: *Cumulative Address-Distance Distribution of memory dependences as seen by stores. Samples are taken at address distances of 16, 256, 4K and 64K. Last value shown (right-most) reports all dependences independently of distance.*

We can observe similar results for the store instructions that experience memory dependences. With few exceptions, virtually all of memory dependence activity is captured with a 64K address window. As it was the case with load instructions, most of the dependence activity is captured even with a relatively small address window of 256 entries. Interestingly, a large fraction of stores experience WAW dependences within the scope of this address window. We will exploit this phenomenon in Chapter 5 to reduce store traffic.

The results of this section suggest that for the programs we studied and for the given data sets a reasonable fraction of memory dependence activity can be captured using relatively small structures. For example, using a structure capable of recording information about the last 256 unique memory locations accessed we can detect roughly 56% (integer codes) and 23% (floating point codes) of all RAW memory dependences (measured on loads). With the same structure we can capture 44% (integer codes) and 53% (floating-point codes) of all RAR memory dependences (measured on loads). While a noticeable fraction of memory dependence escapes detection even when we consider relatively large address distances, we note that for the purposes of the applications we present in chapters 3 through 5 the fraction of loads and stores that have dependences detected is sufficient.

2.3.6 Memory Dependence Status Locality

The most primitive piece of memory dependence information that we consider in this work is whether a load or a store instance (1) will experience a dependence of a given type and, if so, (2) whether the instruction is the source or the sink. We will use the term *memory dependence status (MDS)* to refer to this piece of information (which is a binary value). For this type of memory dependence information, the exact memory dependences a load or a store has are not important, only whether such dependences exist is. For example, in the case of RAW dependences, the memory dependence status of a load instance indicates whether the load has a RAW dependence with a preceding store (which exact store is not important), while the RAW memory dependence status of a store instance indicates whether the store has a RAW dependence with a subsequent load (which exact load is not important). In the case of RAW and WAR dependences the information on whether the store or the load is the sink or the source of the dependence is implied by the dependence type. For RAR and WAW dependences however this information has to be provided explicitly. The example code fragments of Figure 2.8 are useful in explaining why the memory dependence status of instructions may vary over time. Shown in part (a) is a loop that contains a load of M(i) preceded by a conditional — based on the outcome of the call to foo(i)— store to the same memory location. The memory dependence status of the load is one only when foo(i) returns true, otherwise it is zero. The source RAW memory dependence status of the store is always one, because whenever the store is executed a load to the same memory location follows. Part (b) of the figure contains the same code except that the store has been replaced by another load of M(i), ld1. In this case the sink RAR MDS of the second load (shown in bold) will be one only when foo(i) returns true. The source RAR MDS of ld1 will be always one as whenever this instruction is executed, a load to the same memory location follows. Finally, the sink RAR MDS of ld1 is always 0 as no preceding load to same memory location is ever encountered.

```
for i = 0 to N
    if (foo (i)) st M(i)
   ld M(i)
```

(a)

```
for i = 0 to N
    if (foo (i)) ld1 M(i)
   ld M(i)
```

(b)

Figure 2.8: *Examples illustrating variation in the memory dependence status of a load.*

In this section we seek to obtain an indication of whether memory dependence status is amenable to history-based prediction techniques, that is whether is exhibits sufficient regularity. We do so by measuring the *memory dependence status locality* of loads and stores. We define the memory dependence status locality of a load or a store, as the probability that the memory dependence status *observed* by an instance of the instruction is the same as the one *observed* by the immediately preceding instance of the same static instruction. (Ignoring finite storage effects in the predictor implementation, the memory dependence status locality is the equal to the prediction accuracy of a simple last-status predictor.) We emphasize the word “observed” in our definition of memory dependence status locality to signify that as defined, memory dependence status locality is also a function of the mechanism used to detect memory dependences. We introduce this additional parameter in our treatment of memory dependence status locality since memory dependence status prediction becomes more interesting from a practical perspective when instead of considering the whole program we attempt to predict whether a load or a store has a dependence of a particular type under some constraint. This latter point will become apparent when we consider various applications of memory dependence status prediction. For example, in Chapter 5 we will use memory dependence status prediction to predict whether a load has a RAW or a RAR (sink) dependence within the last *n* unique memory addresses touched. We will do so in order to map such loads onto a small and fast storage structure that can hold only *n* memory values. Other constraints are both possible and interesting (for example, whether a load has a RAW dependence within the last *n* instructions, a variant of which we make use in Chapter 3 during the description of selective memory dependence speculation, where this information is used to delay the execution of loads with RAW memory dependences that are visible from within the active instruction window).

As our goal is to provide an indication that sufficient regularity exists, we present two types of measurements of memory dependence status locality, each assuming a different memory dependence status detection mechanism. These two mechanisms are the following: (1) an infinite address window where we consider the whole program, and (2) limited address windows of various sizes. The results are shown in Figure 2.9 for all four possible types of memory dependences. From left to right, we report the memory dependence status locality for the following address windows: 16, 256, 4K, 64K and infinite.

It can be seen that dependence status locality is extremely strong even within the relatively small scope of an address window of 16 entries. In almost all cases locality is above 90% and often approaches 100%. These results suggest that provided that space is available to record the relevant information, memory dependence status prediction should be fairly accurate. Note that prediction accuracy may exceed locality if we make use of some form of hysteresis (e.g., confidence counters). Finally, we can observe that dependence status locality is not directly correlated with the size of the address window. For example, in 129.compress RAR status locality (at the sink loads) drops from approximately 99% to 95% when we move from an address window of 256 to one of 4K. The reason is that in the latter case, many more dependences become visible. These dependences are not necessarily as regular as the ones seen via the smaller address window.

2.3.7 Memory Dependence Locality

For certain applications it may be necessary to also predict the exact set of dependences of a given type an instruction has. In contrast to the memory dependence status of loads and stores (Section 2.3.6) in this case we are interested not only on whether such dependences exist, but also which exactly these dependences are. In chapters 3 and 4 we will make use of such predictors for RAW and RAR dependences. Accordingly we restrict our attention to those two dependence types. It is conceptually convenient to think of memory dependence prediction as a two step process where first we predict the static instructions with which dependences exist and then predict the particular dynamic instances of those instructions. For the purposes of this study we focus on the first step as we will make use of different mechanisms to locate the appropriate instances depending on the application under consideration. A description of the mechanisms used to locate the appropriate instruction instances is given during the description of the specific applications in chapters 3 and 4.

To demonstrate that memory dependences may be amenable to history-based prediction we measure the *memory dependence locality* of loads and stores. Informally, the memory dependence locality of an instruction is a metric of the likelihood that the same dependences (of a given type) are observed in two consecutive executions of the same static instruction. Formally, we define memory dependence locality *n* for a given type of memory dependences, as the probability that the same dependence of the given type has been encountered within the last *n* dependences of the given type experienced by preceding instances of the same static instruction. When each instance encounters a single dependence, locality is exactly the probability that the same dependence has been encountered the last time an instance of the same static instruction was encountered. When multiple dependences are encountered per dynamic instance, locality should be viewed

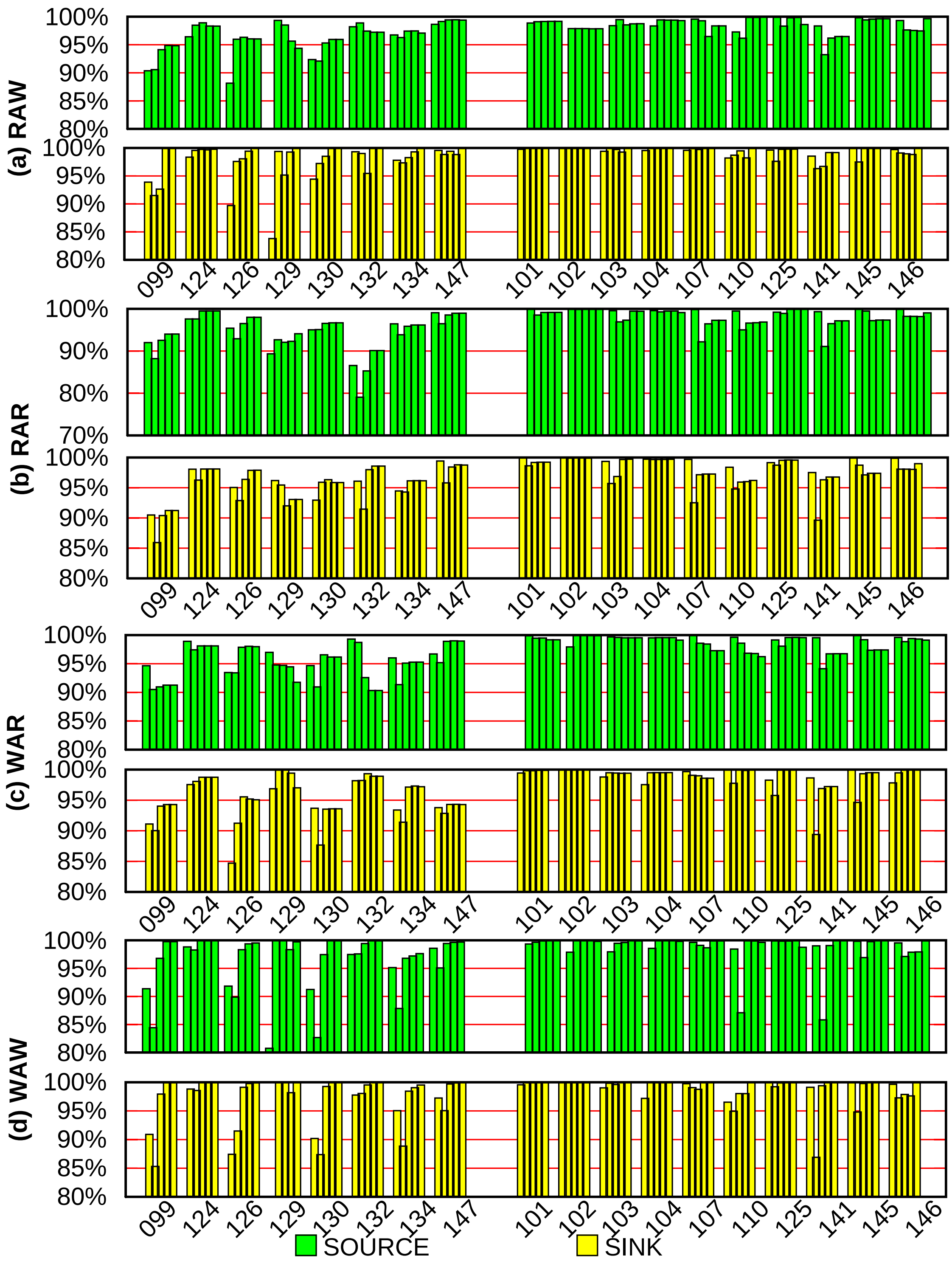


Figure 2.9: Memory dependence status locality as a function of address window size. Address windows shown are: 16, 256, 4K, 64K and infinite (left to right).

as a metric of how many dependences of the given type we have to remember per static instruction in order to correctly predict *all* the dependences of the given type for the next instance of the same static instruction (alternatively, locality in this case is a metric of the working set of the memory dependences of the given type per static instruction). In Section 2.3.7.1 we consider RAW dependences, while in Section 2.3.7.2 we consider RAR dependences.

2.3.7.1 Read-after-Write Dependences

In this section we measure the memory dependence locality of stores and loads that experience RAW dependences. The results are shown in Figure 2.10 where we report fractions over all executed loads (sink graph) or stores (source graph) that experience a RAW dependence. We report locality measurements in the range of 1 to 4. Focusing first on the sink loads we observe that memory dependence locality is very strong. In most programs, nearly 80% or more of all loads with RAW dependences experience the same exact dependence as the last time they were encountered. When we can remember the last two dependences observed by instances of each static load, locality raises above 80% for all programs except 099.go and 132.ijpeg. For those two programs locality remains below 80% even when we can remember the last 4 dependences experienced by instances of each load. There are two reasons why locality may be different than one: (1) loads have more than one RAW dependences which alternate in the execution stream, and (2) each dynamic instance of a load experiences multiple RAW dependences due to the use of different data types (i.e., the load accesses a larger data type than the stores it depends upon). As we have seen in Section 2.3.3, with very few exceptions, load instances experience a single RAW dependence. This observation suggests that the primary reason why locality is not always one is that loads have multiple static RAW dependences which alternate in the execution stream. For this reason, it may be important (depending on the level of memory dependence accuracy desired) to devise memory dependence predictors that are capable of handling more than a single RAW dependence per load instruction. In Chapter 4 we will present such a predictor that uses a level of indirection to represent the memory dependences loads and stores have. Moreover, in Chapter 3 we will validate this observation by demonstrating that for the purposes of memory dependence speculation and synchronization it is important to track multiple RAW dependences per static load (Section 3.8.6).

Focusing on the source stores of RAW dependences, we can observe that locality is generally not as strong as it was for the sink loads. The differences are relatively small for the integer programs while they are more pronounced for the floating point programs. Again multiple RAW dependences are the cause of this phenomenon. As we have seen in Section 2.3.3, a relatively large fraction of stores experiences multiple RAW dependences as many loads read the value they write. In fact, we have seen that this phenomenon is more pronounced for the floating point codes. This explains why locality is not as strong for those programs. To a lesser extent, another reason why locality is not as strong is that different instances of the same static store may experience different RAW dependences. The results of this experiment suggest that memory dependence predictors that record and represent multiple dependences per store may be required especially for the floating point codes. They also suggest that it might be more practical to design memory dependence predictors that let loads locate their producing stores rather than the other way around.

2.3.7.2 Read-after-Read Dependences

In this section we measure the memory dependence locality of loads with RAR dependences. For these experiments we consider two different address window sizes: infinite and 4K. The reason is that RAR dependences can be defined arbitrarily. Given a set of dynamic load instances that access a common memory location, any of the earlier loads can be identified as the source of a RAR dependence with any of the latter loads. As our goal is to provide indications that sufficient regularity for history-based prediction exists we limit our attention to the two aforementioned address windows and to marking the earlier possible load

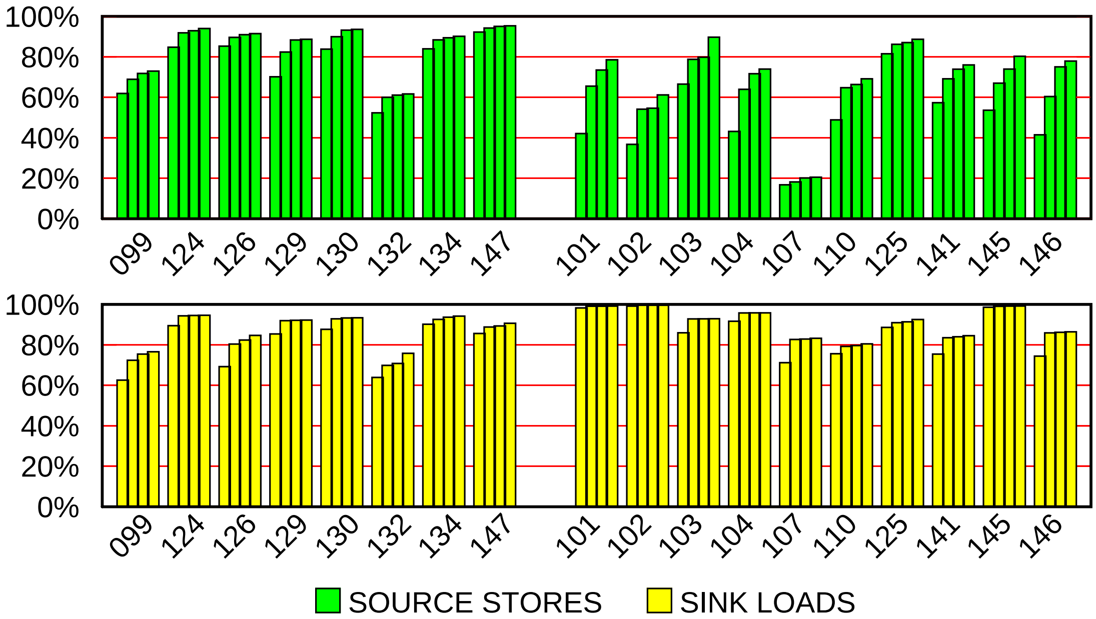


Figure 2.10: *Memory dependence set locality of read-after-write dependences. Locality range shown is 1 to 4 (left to right).*

as the source of RAR dependences. This is the definition of RAR dependence we utilize in the mechanisms presented in Chapter 4. In that chapter, we will also demonstrate that RAR dependence prediction is fairly accurate for other choices of address window sizes and policies (we will not include those loads that have RAW dependences).

The results of these experiments are shown in Figure 2.11 where we differentiate between loads that appear as the sink of RAR dependences and loads that appear as the source of RAR dependences (as per our definition of Section 2.1). Part (a) shows results with the infinite address window (the whole program is considered) and part (b) shows results with the 4K address window. Locality range shown is 1 to 4, while the Y axis reports fractions over all loads that have RAR dependences detected and they are the source (source graph) or the sink (sink graph). Focusing first on the sink loads we observe that the majority of sink loads observe the same dependence as they did the last they were executed (more than 50% of loads see a locality value of 1). With the exception of 099.go and 126.gcc the fraction of loads that observe the same dependence as last time is above 80%. This phenomenon is more pronounced for the floating point codes were this fraction is above 90% with the 4K address window. Since we have seen (Section 2.3.3) that most sink loads of RAR dependences see a single source load per dynamic instance, a locality value of more than 1 typically indicates that multiple source loads exist and that these alternate in the execution stream.

We see another interesting phenomenon when we compare the locality results with the infinite address window with those with the finite address window. Surprisingly, for most programs locality improves with the finite address window. This observation suggests that in choosing the source loads for RAR dependences, close-by loads typically exhibit better locality. We will observe similar phenomena in Chapter 4 where we will also see that in some cases distant RAR dependences exhibit less regularity than close-by RAR dependences.

Focusing on the source loads, locality is again strong though not as strong as it was for sink loads. One explanation is that, typically, a particular instance of a source load sees multiple RAR dependences with var-

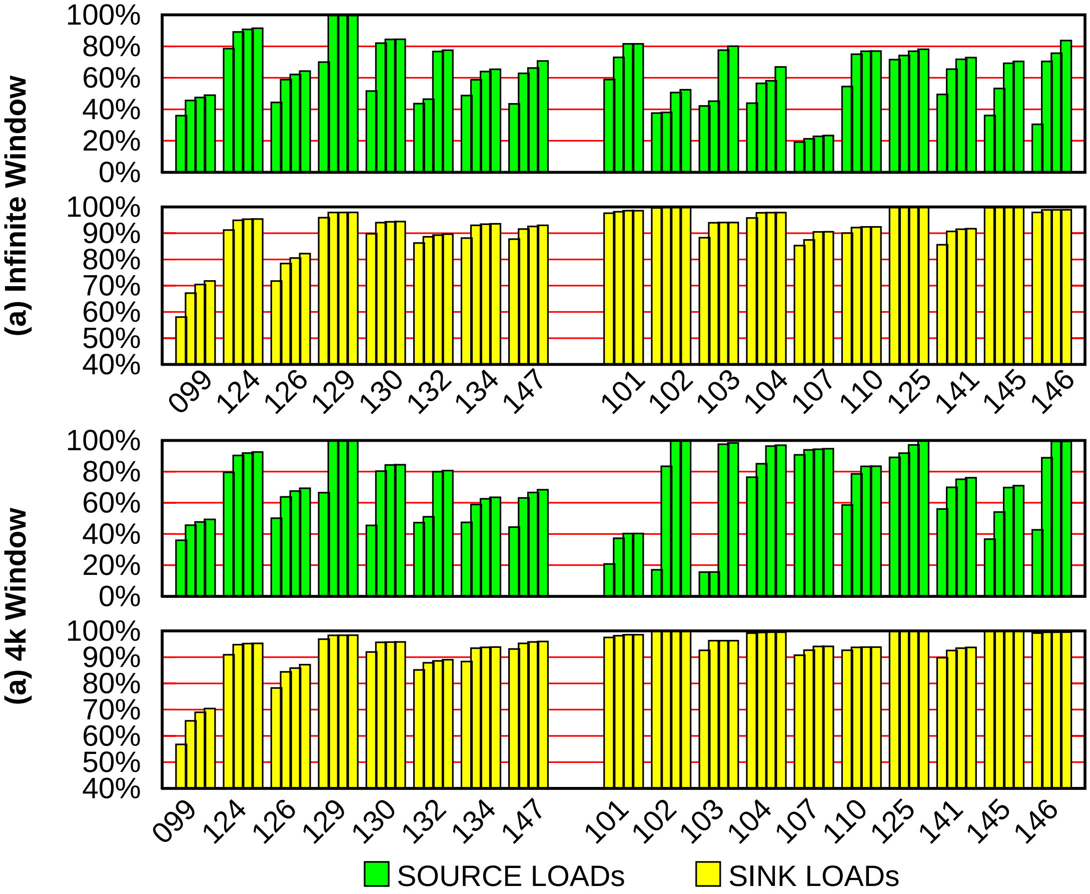


Figure 2.11: *Memory Dependence Set locality of read-after-read dependences. Locality range shown is 1 to 4. (a) Infinite address window. (b) 4K entry address window.*

ious instances of succeeding loads (i.e., many loads read the same location). We identified this phenomenon in Section 2.3.3.1. While it might be that every time the source load is executed it experiences the very same dependences, these dependences are more than 4. In conjunction with our previous measurements on the shape of RAR dependences (i.e., how may RAR dependences each load instance experiences) we can make another observation (Section 2.3.3.1). We have seen that in very few cases, RAR source loads experience a single RAR dependence (Figure 2.3, part (b), source loads, on page 30). However, the results of this section show that source loads with memory dependence set locality of 1 are much more frequent. These two observations suggest that in some cases an instance of a source load experiences dependences with many instances of the same static load.

For the purposes of this analysis we do not consider any other characteristics of RAR dependences. The results of Chapter 4 will provide additional indications that highly accurate prediction of RAR dependences is possible. However, we note that the results of this section suggest that there is regularity in the RAR dependence stream of the programs studied, suggesting that history-based prediction of RAR dependences may be possible.

2.4 Summary

In this section we reviewed what memory dependences are, and discussed the general concepts underlying a class of memory dependence predictors based on history. We were concerned with whether programs exhibit sufficient regularity in their memory dependence stream and whether relatively small structures should be sufficient to capture and predict memory dependence information. Briefly, our findings were:

1. Most loads and stores experience dependences. 90%-100% of all loads experience RAW dependences. 45%-75% of loads experience RAR dependences. 45%-90% of all loads experience WAR dependences. 50%-100% of all stores experience RAW and WAR dependences, while nearly all stores are overwritten by a latter store (WAW dependences).
2. No single part of the memory address space is responsible for most of memory dependence activity (Section 2.3.2.1).
3. Most of RAW and RAR dependences are across many dynamic instructions (more than 4K - Section 2.3.3.1).
4. (a) The instances of sink instructions of RAW and RAR dependences observe a single source store or load respectively (see Section 2.1 for a definition of sink and source instructions), (b) significant fractions of source stores of RAW dependences see multiple sink loads, and (c) the majority of source loads of RAR dependences see more than one sink loads (Section 2.3.3.1). These observations suggest this in predicting memory dependences it may be convenient to devise schemes in which the sink instructions attempt to predict the source instructions instead of the other way around, as source instructions typically see many sink instructions.
5. A significant fraction of loads and stores with RAW and RAR dependences have relatively large static dependence set sizes (i.e., more than 4 — Section 2.3.3.2).
6. The working set of instructions with dependences is relatively small for all four dependence types (i.e., less than 4K instructions in virtually all cases — Section 2.3.4).
7. Relatively small structures can be used to capture reasonable fractions of memory dependence activity. However, in some cases, a relatively large fraction of memory dependence activity escapes detection even when we can record the last 64K unique memory addresses accessed (see Section 2.3.5 for detailed measurements).
8. The memory dependence status locality of both source and sink instructions for all dependence types is relatively high (above 90% for an address window of 4K entries — Section 2.3.6).
9. The memory dependence locality of source and sink instructions for RAW and RAR dependences is high: 60%-100% for RAW sink loads, 58% -100% for RAR sink loads, 20%-95% for RAW source stores and 18%-90% for RAR source loads. Locality is stronger for sink loads mostly because source stores and loads experience multiple RAW or RAR dependences per dynamic instance (Section 2.3.7).

The results of this study provide a first indication that history-based prediction of memory dependences and of their status may be both practical and accurate.

Chapter 3

Dynamic Memory Dependence Speculation and Synchronization

Given the relatively high frequency of memory reads (loads account for roughly 20%-30% of all instructions executed), *memory latency*, that is the time it takes for memory to respond to requests, can have a significant impact on performance. The memory latency problem can be attacked directly. That is we may employ techniques which aim at reducing the time it takes memory to respond to load requests. This is the goal of traditional memory hierarchies where a collection of faster but smaller memory devices, commonly referred to as caches, is used to provide faster access to a dynamically changing subset of memory data. However, given the limited size of caches and imperfections in the caching policies, memory hierarchies provide only a partial solution to the memory latency problem.

An alternative, yet orthogonal direction of attacking the memory latency problem seeks to tolerate memory latency. The goal here is to send loads to memory earlier, as far in advance from the instructions that need the data that will be read. The net result this method hopes to achieve is overlapping memory latency with other useful computation. Of course, the utility of this approach extends beyond tolerating memory latency as performance may benefit by the parallel execution of instructions even when unit memory latencies are observed. Sending loads to memory as early as possible requires moving loads up in the execution order, placing them in a position that might be different than the one implied by the program. That is, it requires the ability to extract and exploit *load/store parallelism* and to execute instructions *out-of-order*. This motion of loads can be performed either statically or dynamically.

In this chapter we review previously proposed dynamic methods of executing memory operations out-of-order, and demonstrate that higher performance is possible if *memory dependence speculation* is used. In memory dependence speculation, a load may execute before a preceding store on which it *may be* data dependent (i.e., the store may be writing the data needed by the load). We discuss the trade-offs involved in using memory dependence speculation and explain that care must be taken to balance the benefits of correct speculation against the net penalty incurred by erroneous speculation. We demonstrate that as dynamically-scheduled ILP processors are able to schedule instructions over larger regions, the net performance loss of erroneous memory dependence speculation (mispeculation) can become significant. Specifically, we demonstrate the validity of this observation for the following two environments: (1) in a centralized, continuous window processor when preceding store address information is not available or when it is too expensive to look at, and (2) in a distributed, split-window processor even when loads can inspect the addresses of preceding stores before accessing memory.

Accordingly, we are concerned with methods of reducing net mispeculation penalty. Our focus is on methods to improve the accuracy of memory dependence speculation. We propose and evaluate techniques to: (i) predict those load instructions whose immediate execution would violate a true memory dependence, and (ii) delay their execution *only as long as it is necessary* to avoid the mispeculation. When these two goals are met, we succeed in avoiding mispeculations while retaining the benefits of aggressive out-of-order execution. We consider a number of alternative techniques and demonstrate that for the distributed, split-window processor, best performance is achieved when *memory dependence speculation/synchronization* is used. In this novel technique, memory dependence prediction is used to identify those loads and stores that have to be synchronized to avoid violating true memory dependences.

The rest of this chapter is organized as follows: in Section 3.1 we motivate the need for exploiting load/store parallelism and discuss the challenges raised by ambiguous (i.e., temporarily unknown) memory dependences. We use this discussion to motivate memory dependence speculation which we discuss in Section 3.2. Here we review how memory dependence speculation is being used today and provide qualitative arguments on why, techniques to improve the accuracy of memory dependence speculation might be useful. In Section 3.3 we discuss a number of memory dependence speculation policies and argue for memory dependence speculation and synchronization (speculation/synchronization for sort), a policy that aims at mimicking what is ideally possible. In Section 3.4, we discuss the requirements of speculation/synchronization. An implementation framework for our proposed technique we present in Section 3.5. We then address a number of important from a practical perspective issues in Section 3.6. We review related work in Section 3.7. We provide experimental evidence in support of the utility of our proposed technique and of our observations in Sections 3.8 and 3.9. In Section 3.8, we focus on a distributed, split-window processing model while in Section 3.9 we study memory dependence speculation under a centralized, continuous instruction window processing model. Finally, we summarize our findings in Section 3.10.

3.1 Using Load/Store Parallelism To Improve Performance

Sequential programs are written with an implied, total order where instructions are meant to execute one after the other and in the order specified by the program. However, the same results are produced if any two instructions that have no true (RAW) data dependences between them are allowed to execute in any order, possibly in parallel[1]. In this case, instruction-level parallelism exists in the program. We can exploit this property to improve performance by executing instructions in an order different than that implied by the program, possibly in parallel. This ability is also useful in tolerating slower memory devices by overlapping the processing of load requests with other useful computation. Moving loads as far ahead of the instructions that read their data, and in general exploiting instruction-level parallelism can be done statically (see related work section) or dynamically. In this work we focus on dynamic, hardware based techniques.

[1] Strictly speaking, program semantics are maintained so long as instructions read the same value as they would in the original program implied order. This does not necessarily imply that a dependent pair of instructions executes in the program implied order. We avoid making this distinction in the discussion of this chapter for clarity.

Typical modern dynamically-scheduled ILP processors, exploit instruction-level parallelism by forging ahead into the execution stream, building an *instruction window,* a set of instruction to execute. These processors, then attempt to convert the total, program implied order within this set into a partial order. The shape of the partial order and for that the performance improvements so obtained are heavily influenced by the processor's ability to uncover the true data dependences among the instructions currently under consideration. In the case of loads, the performance improvements obtained are determined by the processor's ability to send load requests to memory as early as possible without, however, allowing a load to access memory before a preceding store with which a true data dependence exists. One way of doing so, is to *first determine* the true dependences a load has and *then* use that information to schedule its execution. With this approach, we ensure that no true dependences are violated in the resulting execution order. In the case of loads and stores the process of determining the data dependences they have is commonly referred to as *disambiguation*.

Determining the data dependences among the instructions in the instruction window requires inspection of the named locations they access. Unfortunately, these named locations are not necessarily available immediately. This is typical for stores and loads which have to perform an address calculation to determine the memory address they will be accessing. As a result, at any point during execution, memory dependences may be *unambiguous* (i.e., a load consumes a value that is known to be created by a store preceding it in the total order) or *ambiguous* (i.e., a load consumes a value that *may* be produced by a store preceding it in the total order). During execution, an ambiguous dependence gets eventually resolved to either a true dependence, or to no dependence. We will use the term *false dependence* to refer to an ambiguous dependence that eventually gets resolved to no dependence. As we explain next, false dependences present a challenge to the out-of-order execution of load instructions.

Ambiguous memory dependences may obscure some of the parallelism that is present. The reason is that to maintain program semantics a load has to wait for a store with which an ambiguous dependence exist only if a dependence really exists. If the ambiguous dependence is a false dependence, any execution order is permissible, including ones that allow the load to execute before the store. This latter case, represents an opportunity for parallelism and for higher performance. Unfortunately, the mere classification of a dependence as ambiguous implies the inability to determine whether a true dependence exists without actually waiting for the addresses accessed by both instructions to be calculated. Worse, in the absence of any explicit memory dependence information (the common case today), a dynamically scheduled ILP processor has to assume that ambiguous dependences exist among a load and any preceding store that has yet to calculate its address (provided that no intervening store accesses the same address and has calculated its address).

As we will demonstrate in the evaluation section, significantly higher performance is possible if we could make loads wait only for those ambiguous dependences that get resolved to true dependences. Moreover, we demonstrate that this performance difference widens as the size of the instruction window increases. To expose some of the parallelism that is hindered by ambiguous memory dependences, memory dependences speculation can be used. This technique is the topic of the next section.

3.2 Memory Dependence Speculation

Memory dependence speculation aims at exposing the parallelism that is hindered by ambiguous memory dependences. Under *memory dependence speculation*, we do not delay executing a load until *all* its ambiguous dependences are resolved. Instead, we guess whether the load has any true dependences. As a result, a load may be allowed to obtain memory data speculatively before a store on which it is ambiguously dependent executes. Eventually, when the ambiguous dependences of the load get resolved, a decision is made on whether the resulting execution order was valid or not. If no true dependence has been violated, speculation was successful. In this case, performance may have improved as the load executed earlier than it would had it had to wait for its ambiguous dependences to be resolved. However, if a true dependence was violated, the speculation was erroneous (i.e., a misspeculation). In the latter case, the effects of the speculation must be undone. Consequently, some means are required for detecting erroneous speculation and for ensuring correct behavior. Several mechanisms that provide this functionality, in either software and/or hardware, have been proposed [39,26,27,28,37,56,65,71]. The hardware techniques used today work by invalidating and re-executing all instructions following the misspeculated load. We will use the term *squash invalidation* to refer to this recovery method.

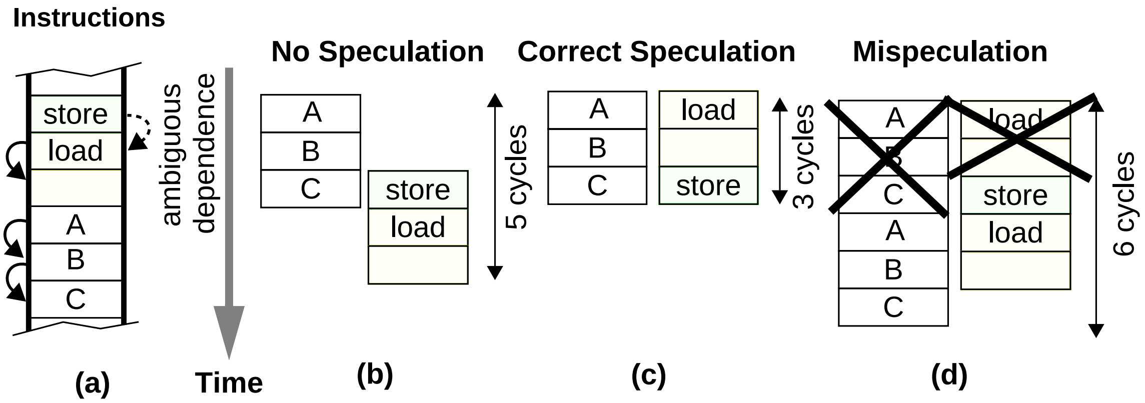


Figure 3.1: Using memory dependence speculation may affect performance either way. (a) Code with an ambiguous memory dependence. Continuous arrows indicate register dependences. Parts (b) through (d) show how this code may execute in a dynamically-scheduled ILP processor capable of executing two instructions per cycle. We assume that due to other dependences, the store may execute only after two cycles have passed. (b) Execution order when no memory dependence speculation is used. (c) Memory dependence speculation is used and the ambiguous dependence gets resolved to no dependence. (d) Memory dependence speculation is used, and the ambiguous dependence gets resolved to a true dependence.

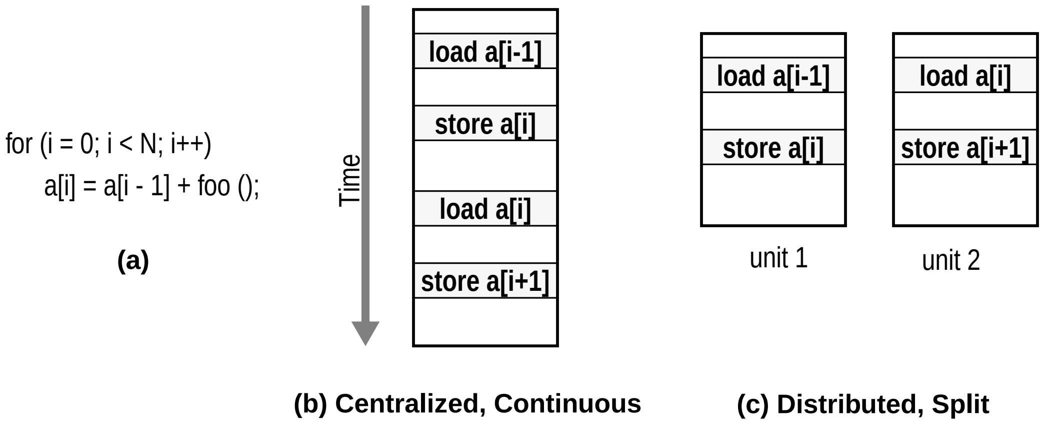
Though memory dependence speculation may improve performance when it is successful, it may as well lead to performance degradation when it is wrong. We demonstrate either possibility with the example of Figure 3.1. The reason is that a penalty is typically incurred on misspeculation. The penalty includes the following three components: (1) the work thrown away to recover from the misspeculation, which in the case of squash invalidation, may include unrelated computations, (2) the time, if any, required to perform the invalidation, and finally (3) the opportunity cost associated with not executing some other instructions instead of the mispeculated load and the instructions that used erroneous data. Consequently, in using memory dependence speculation care must be taken to balance the performance benefits obtained when speculation is correct against the net penalty incurred by erroneous speculation. To gain the most out of memory dependence speculation we would like to use it as aggressively as possible while keeping the net cost of misspeculation as low as possible. Ideally, loads would execute as early as possible while misspeculations would be completely avoided.

Prior to this work, memory dependence speculation was either not used at all or was used whenever the opportunity to execute a load existed. In the latter case, a load with ambiguous dependences was always allowed to access memory. We will use the term *naive memory dependence speculation* to refer to this form of memory dependence speculation, in order to signify that no explicit attempt is made to guess whether a load should wait. The reasons why memory dependence speculation was either not used or used without any effort to reduce misspeculations include the following: (1) in the relatively small instruction windows of those ILP processors there was often little to be gained from extracting load/store parallelism, and (2) the probability of a true memory dependence being violated when memory dependence speculation was used was relatively small. In this work we are interested on whether these observations change as we move toward larger effective instruction window sizes.

As we demonstrate in Sections 3.8.2, 3.9.2 and 3.9.3, in most cases, naive memory dependence speculation offers superior performance compared to having to wait until ambiguous dependences are resolved (i.e., no speculation). Moreover, we demonstrate that the benefits of memory dependence speculation increase as the size of the instruction window also increases (sections 3.8.1 and 3.9.1). More importantly however, we also demonstrate that further performance improvements are possible if we could avoid mispeculations. Specifically, we demonstrate that further performance improvements are possible under the following two execution models: (1) a centralized, continuous window ILP processor, and (2) in a distributed, split-window ILP processor. In the centralized, continuous window processor, the net penalty of mispeculation becomes significant when loads cannot inspect the addresses of preceding stores either because a mechanism is not provided (to simplify the design) or because of the latency required to inspect store addresses. In the distributed, spit-window processor mispeculations are problematic independently of whether loads can inspect preceding store addresses. Moreover, we demonstrate that the potential benefits increase as the size of the instruction window also increases in either processor environment.

At this point it is interesting to consider why, in the centralized, continuous-window execution model, mispeculations can typically be avoided if loads are allowed to inspect preceding store addresses before obtaining a memory value, while in the distributed, split-window execution model this technique proves ineffective. For this purpose, we will use the example of Figure 3.2. Part (a) of the figure shows a loop with a recurrence between the "load a[i - 1]" of iteration i and the "store a[i]" of iteration *i - 1*. (While this code is prone to static disambiguation, our goal here is not to demonstrate the power of dynamic memory dependence speculation/synchronization.) Part (b) shows how two iterations of this loop might get executed under the centralized, continuous window execution model. Under this model, instructions are fetched in order and the window is filled up gradually. How fast the window fills up is determined by several factors including the fetch bandwidth, the instruction cache characteristics, and branch prediction accuracy. In the particular processor we study in Section 3.9, the maximum fetch bandwidth is equal to the maximum execution bandwidth, and moreover, a scheduler that gives priority to older instructions (in program order) is used. As a result, by the time the dependent load (load a[i]) is encountered and calculates its address, the preceding store (store a[i]) with which a true dependence exists has also been fetched, and has also calculated its address. Under these conditions and provided that the load is allowed to inspect the addresses of preceding stores, it finds that it should wait and not speculatively access memory. As we demonstrate in Section 3.9.3, memory dependence mispeculations are virtually non-existent in this environment. We do argue however, our techniques can be used as a potentially lower complexity, shorter clock cycle alternative to scheduling load/stores by incorporating the load/store scheduling functionality in the existing register scheduler. While memory dependence mispeculations are not an issue for a centralized, continous window processor, future processors may utilize more aggressive front-ends and may have to rely on partitioning to balance between short clock cycles and larger instruction windows [26, 82, 66, 44, 87, 90, 72, 25, 85, 32]. Under this different set of assumptions, instructions are not necessarily fetched in program order, and moreover, enforcing program order priority in the scheduler may not be possible. For this reason lets us now consider a distributed, split-window execution model. Under this model, the two iterations of the loop may get assigned to different units, as shown in part (b) of Figure 3.2. As a result, the load may calculate its address long before the store has had a chance to do so. For this reason, even if the load could inspect preceding store addresses and even if that check could be done instantaneously, the mispeculation could not be avoided. (Information about Multiscalar, the distributed, split-instruction window execution model we use in this work, is given in Section 3.6.1.)

Motivated by the aforementioned observations, in this work we are concerned with techniques to minimize the net penalty of mispeculation, while maintaining the performance benefits of aggressive memory dependence speculation. We identify three possible directions: (1) minimizing the amount of work that is



```
for (i = 0; i < N; i++)
    a[i] = a[i - 1] + foo ();
```

(b) Centralized, Continuous

(c) Distributed, Split

Figure 3.2: *Executing a loop under: (b) a centralized, continuous-window execution model, and (c) a distributed, split-window execution model.*

lost on mispeculation, (2) reducing the time required to redo the work that is lost on mispeculation, and (3) reducing the probability of mispeculation. In this work we consider the third alternative. We review techniques that follow the other two directions in the related work section (Section 3.7).

3.3 Memory Dependence Speculation Policies

The ideal memory dependence speculation mechanism not only avoids mispeculations completely, but also allows loads to execute as early as possible. That is, loads with no true dependences (within the instruction window) execute without delay, while loads that have true dependences are allowed to execute only after the store (or the stores) that produces the necessary data has executed. It is implied that the ideal memory dependence speculation mechanism has perfect knowledge of all the relevant memory dependences.

An example of how the ideal memory dependence speculation mechanism affects execution is shown in Figure 3.3. In part (b), we show how the code sequence of part (a) may execute under ideal memory dependence speculation and in part (c) we show how the execution may progress under naive memory dependence speculation. The example code sequence includes two store instructions, *ST-1* and *ST-2*, that are followed by two load instructions, *LD-1* and *LD-2*. Ambiguous dependences exist among these four instructions as indicated by the dotted arrows. During execution, however, only the dependence between *ST-1* and *LD-1* is resolved to a true dependence (as indicated by the continuous arrow). Under ideal dependence speculation, *LD-2* is executed without delay, while *LD-1* is forced to synchronize with *ST-1*.

In contrast to what is ideally possible, in a real implementation, the relevant data dependences are often unknown. Therefore, if we are to mimic the ideal data dependence speculation mechanism, we have to attempt: (1) to predict whether the immediate execution of a load is likely to violate a true data dependence, and if so, (2) to predict the store (or stores) the load depends upon, and, (3) to enforce synchronization between the dependent instructions.

However, since this scheme seems elaborate, it is only natural to attempt to simplify it. One possible simplification is to use *selective* memory dependence speculation, i.e., carry out only the first part of the ideal 3-part operation. In this scheme the loads that are likely to cause mispeculation are not speculated. Instead, they wait until the all their ambiguous dependences are resolved; explicit synchronization is not performed. We use the term *selective memory dependence speculation* (or selective speculation for short) to signify that we make a decision on whether a load should be speculated or not. In contrast, in ideal dependence specula-

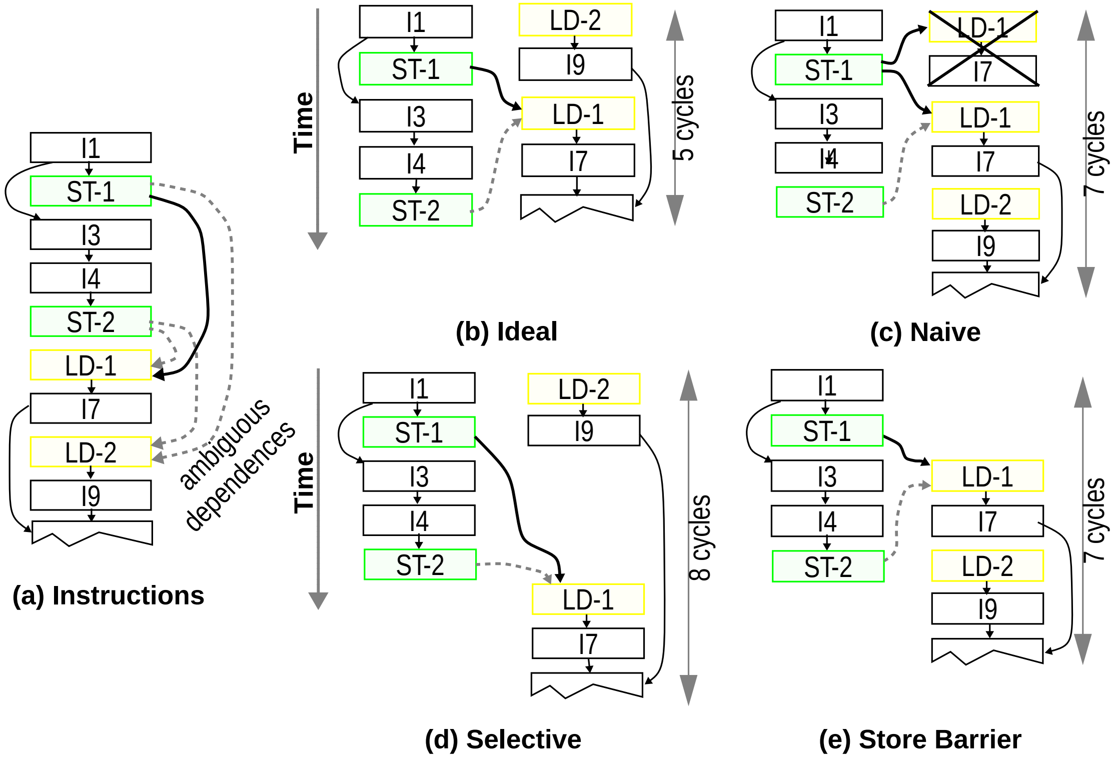


Figure 3.3: Example illustrating various memory dependence speculation policies. Arrows indicate dependences. Dependences through memory are indicated by thicker lines. Dotted arrows indicate ambiguous dependences that are resolved to no-dependence during execution.

tion, we make a decision on when is the right time to speculate a load. While selective memory dependence speculation may avoid mispeculations, due to the lack of explicit synchronization, this prediction policy may as well make loads wait longer than they should and for this reason may negatively impact performance. This case we illustrate with the example shown in part (d) of Figure 3.3. In this example, *LD-2* is speculated, whereas *LD-1* is not, since prediction correctly indicates that *LD-2* has no true dependences while *LD-1* does. However, as shown *LD-1* is delayed more than necessary as it has to wait not only for *ST-1* but also for *ST-2*. In practice, and as we demonstrate in Section 3.8.4, selective data dependence speculation can lead to inferior performance when compared to naive speculation (part (c) of Figure 3.3) even when perfect prediction of dependences is assumed, because, while this policy avoids mispeculations it often fails to delay loads only as long as it is necessary.

Another possible simplification that has been proposed (see related work section) is the *store barrier* policy. In this technique a prediction is made on whether a store has a true dependence that would normally get mispeculated. If it does, *all* loads following the store in question are made to wait until the store has posted its address for disambiguation purposes. While the store barrier policy can be successful in (1) eliminating mispeculations, and (2) delaying loads that should wait only as long as it is necessary, it may as well lead to inferior performance since it may unnecessarily delay other unrelated loads that have no true dependences that can be mispeculated. While, in the example of Figure 3.3, the store barrier policy is shown to perform better than selective speculation, the opposite can also be true (for example, if other loads, following *LD-1* existed they would too get delayed under the store barrier policy, while they wouldn't under the selective policy). In the evaluation section, we do not consider the store barrier policy for two reasons: (1) as proposed it is not compatible with the distributed, split-window architecture we use in our evaluation (variants may be possible however), and (2) it has been shown [17] (see related work section) that for a continuous instruction window processor the performance so obtained is inferior to the technique we describe in the next section.

Even though other simplifications to the 3-part ideal operation may be possible, in this work we restrict our attention to dependence speculation schemes that attempt to mimic the ideal data dependence speculation system. In the next section, we present *dynamic memory dependence speculation/synchronization*, a technique that utilizes memory dependence prediction to identify those store-load pairs that ought to be synchronized in order to avoid memory dependence violations while delaying load execution only as long as it is necessary.

3.4 Mimicking Ideal Memory Dependence Speculation

To mimic the ideal data dependence speculation system, we need to implement all the 3 components of the ideal system as described in the previous section. That is, we must: (1) dynamically identify the store-load pairs that are likely to be data dependent and whose normal execution will result in a memory dependence violation, (2) assign a synchronization mechanism to dynamic instances of these dependences, and (3) use this mechanism to synchronize the store and the load instructions.

To identify the store-load pairs that need to be synchronized we may use history-based memory dependence prediction. With this scheme, naive memory dependence speculation is initially used for all loads. That is, a load is initially allowed to execute as soon as its address is calculated and memory resources are available. With this policy, as execution progresses mispeculations will be encountered. Instead of discarding the information available when a mispeculation occurs (as we would under naive memory dependence speculation), we collect information about the instructions involved. For example, we may record the static dependence that was violated, that is a (store PC, load PC) pair. The next time a load or a store that has previously incurred a mispeculation is encountered, we can use the recorded information to predict whether synchronization has to take place in order to avoid a mispeculation. In the predictors we consider in this work, mispeculation history is associated with the static loads and stores using their PC.

As explained in Chapter 2, for history-based memory dependence prediction to be possible for our purposes, it is imperative that past mispeculation behavior to be indicative of future dependences that ought to be synchronized. In Chapter 2 we have provided evidence that *all* RAW memory dependences exhibit relatively high locality and small working sets, which both constitute strong indications that RAW memory dependences may be amenable to history-based prediction. While, the two aforementioned results do not constitute proof that similar behavior is exhibited when we restrict our attention to those dependences that are mispeculated, they do provide a indication that this may be true. As the results presented in Sections 3.8.5 through 3.8.7 and in Section 3.9.4 imply, history-based prediction is both possible and accurate even when we restrict our attention to only those dependences that would be mispeculated under naive memory dependence speculation.

With a mechanism to predict whether a load or a store needs to be synchronized we next need: (1) a synchronization mechanism, and (2) a method of having the appropriate dynamic instances of loads and stores locate each other through the synchronization mechanism. In the rest of this section we first discuss a synchronization mechanism. Then, we consider how load and store instances locate each other through this synchronization mechanism.

An apt method of providing the required synchronization dynamically is to build an association between the store-load instruction pair. Suppose this dynamic association is a condition variable on which only two operations are defined: *wait* and *signal*, which test and set the condition variable respectively. These operations may be logically incorporated into the dynamic actions of the dependent load and store instructions to achieve the necessary synchronization.

This concept we illustrate with the example of Figure 3.5 where we assume that some method exists to dynamically associate store-load instruction pairs with condition variables (we discuss these means later in this section). As shown in part (a), an earlier mispeculation results in the association of a condition variable with a subsequent dynamic instance of the offending store-load instruction pair. With the condition variable in place, consider the sequence of events in the two possible execution sequences of the load and store instructions. In part (b), the load is ready to execute before the store. However, before the load executes, it tests the condition variable; since the test of the condition variable fails, the load waits. After the store executes, it sets the condition variable and signals the waiting load, which subsequently continues its execution as shown. No mispeculation is observed, and the sequential order is preserved. In part (c), the order of execution is a store followed by a load. After the stores executes, it sets the condition variable and records a signal for the load. Before the load executes, it tests the condition variable; since the test of the condition variable succeeds, the load continues its execution as shown (the condition variable is reset at this point). One may wonder why synchronization is provided even when the execution order follows the program order (i.e., store followed by load). This scenario represents the case where dependence prediction correctly indicates that a dependence exists but fails to detect that the order of execution has changed. The order of execution may change, for example, either (1) in response to external events whose behavior is not easy or desirable to track and predict, such as cache misses or resource conflicts, or (2) because of the successful synchronization of another, unrelated dependence. Synchronization is desirable even in these cases since, otherwise, the corresponding load will be delayed unnecessarily.

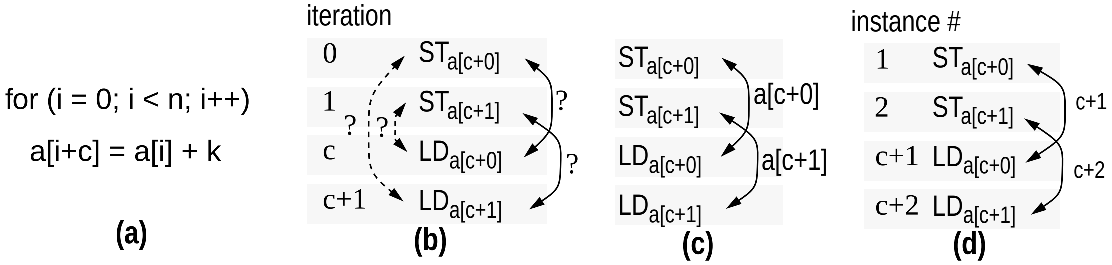


Figure 3.4: *Example code sequence that illustrates that multiple instances of the same static dependence can be active in the current instruction window. In parts (b), (c), and (d), the relevant store and load instructions from four iterations of the loop of part (a) are shown.*

Once condition variables are provided, some means are required to assign a condition variable to a dynamic instance of a store-load instruction pair that has to be synchronized. If synchronization is to occur as planned, the mapping of condition variables to dynamic dependences has to be unique at any given point of time. One approach is to use just the address of the memory location accessed by the store-load pair as a handle. This method provides an indirect means of identifying the store and load instructions that are to be synchronized. Unless the store location is accessed only by the corresponding store-load pair, the assignment will not be unique.

Alternatively, we can use the dependence edge as a handle. The static dependence edge may be specified using the (full or part of) instruction addresses (PCs) of the store-load pair in question. (Compared to using addresses, a potential advantage of this approach is that PC information is available earlier in the pipeline. This property could be exploited to reduce the effective latency of synchronization by having stores initiate synchronization in parallel or prior to the completion or initiation of their memory access.) Unfortunately,

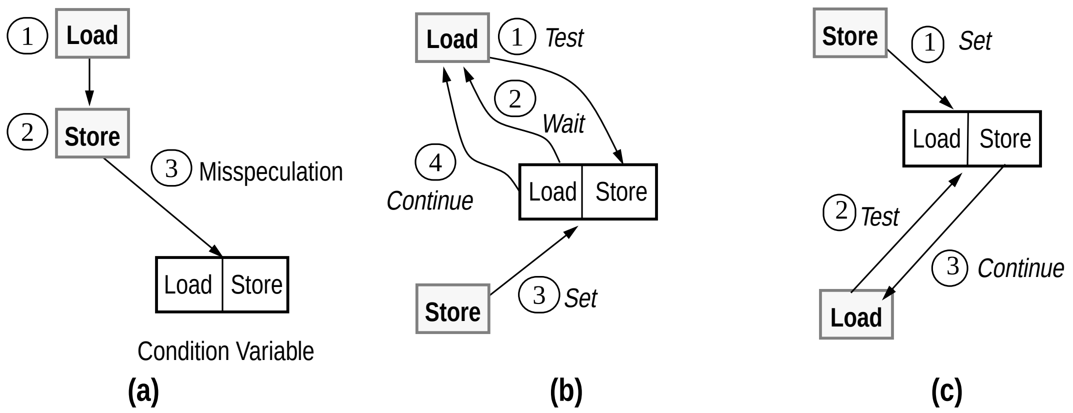


Figure 3.5: *Synchronization example*

as exemplified by the code sequence of Figure 3.4 part (b), using this information may not be sufficient to capture the actual behavior of the dependence during execution; the pair (PC_{ST}, PC_{LD}) matches against all four edges shown even though the ones marked with dotted arrows should not be synchronized. A static dependence between a given store-load pair may correspond to multiple dynamic dependences, which need to be tracked simultaneously.

To distinguish between the different dynamic instances of the same static dependence edge, a tag (preferably unique) could be assigned to each instance. This tag, in addition to the instruction addresses of the store-load pair, can be used to specify the dynamic dependence edge. In order to be of practical use, the tag must be derived from information available during execution of the corresponding instructions. A possible source of the tag for the dependent store and load instructions is the address of the memory location to be accessed, as shown in Figure 3.4 part (c). An alternate way of generating tags is to have a load synchronize with the closest preceding instance of the store identified by the static dependence. While this scheme may delay a load more than it should (as in our example, where $LD_{a[c+0]}$ will wait for $ST_{a[0+1]}$), the performance impact of this delay may not be large.

In this work, and as our focus is on a distributed, split-window execution model where instructions are not fetched in order, we use an alternate way of generating instance tags. The scheme we use is an approximation of the scheme shown in part (d) of Figure 3.4, where dynamic store and load instruction instances are numbered based on their PCs. The difference in the instance numbers of the instructions which are dependent, referred to as the *dependence distance*, may be used to tag dynamic instances of the static dependence edge (as may be seen for the example code, a dependence edge between ST_i and $LD_{i+distance}$ is tagged - in addition to the instruction PCs - with the value i+distance). We approximate this scheme by using the distance in processing units between the instructions that are mispeculated. Though all the aforementioned tagging schemes strive to provide unique tags, each may fall short of this goal under some circumstances (for example, the dependence distance may change in a way that we fail to predict, or the address accessed may remain constant across all instances of the same dependence).

In the rest of the discussion we restrict our attention to second scheme where the dependence distance is used to tag dependences. We note from a practical perspective, several inconveniences exist in the scheme we have just described (For example, how to track and predict multiple dependences per store or load). In the discussion that follows and for clarity, we initially ignore these issues and present an implementation framework in Section 3.5. With a basic understanding of how the support structures operate, we then, in Section 3.6, address a number of important from a practical perspective issues.

3.5 Implementation Aspects

As we discussed in the previous section, in order to improve the accuracy of data dependence speculation, we attempt: (1) to predict dynamically, based on the history of mispeculations, whether a store-load pair is likely to be mispeculated and if so, (2) to synchronize the two instructions. In this section, we describe an implementation framework for this technique. For the purposes of this section we assume a centralized implementation, ignore the possibility of multiple dependences per load or store, and assume fully-associative structures. In Section 3.6, we address these issues.

We partition the support structures into two interdependent tables: a *memory dependence prediction table* (MDPT) and a *memory dependence synchronization table* (MDST). The MDPT is used to identify, through prediction, those instruction pairs that ought to be synchronized. The MDST provides a dynamic pool of condition variables and the mechanisms necessary to associate them with dynamic store-load instruction pairs to be synchronized. In the discussion that follows, we first describe the support structures and then proceed to explain their operation by means of an example. We present the support structures as separate, distinct components of the processor. Other implementations may be possible and desirable.

MDPT: An entry of the MDPT identifies a static dependence and provides a prediction as to whether or not subsequent dynamic instances of the corresponding static store-load pair will result in a mispeculation (i.e., should the store and load instructions be synchronized). In particular, each entry of the MDPT consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), (4) dependence distance (DIST), and (5) optional prediction (not shown in any of the working examples). The valid flag indicates if the entry is currently in use. The load and store instruction address fields hold the program counter values of a pair of load and store instructions. This combination of fields uniquely identifies the static instruction pair for which it has been allocated. The dependence distance records the difference of the instance numbers of the store and load instructions whose mispeculation caused the allocation of the entry (if we were to use a memory address to tag dependence instances this field would not have been necessary). The purpose of the prediction field is to capture, in a reasonable way, the past behavior of mispeculations for the instruction pair in order to aid in avoiding future mispeculations or unnecessary delays. Many options are possible for the prediction field (for example an up-down counter or dependence history based schemes). The prediction field is optional since, if omitted, we can always predict that synchronization should take place. However, we note that in our experimentation we found that it is better if synchronization is enforced only after a load has been mispeculated a couple of times (e.g., three times).

MDST: An entry of the MDST supplies a condition variable and the mechanism necessary to synchronize a dynamic instance of a static instruction pair (as predicted by the MDPT). In particular, each entry of the MDST consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), (4) load identifier (LDID), (5) store identifier (STID), (6) instance tag (INSTANCE), and (7) full/empty flag (F/E). The valid flag indicates whether the entry is, or is not, in use. The load and store instruction address fields serve the same purpose as in the MDPT. The load and store identifiers have to uniquely identify, within the current instruction window, the dynamic instance of the load or the store instruction respectively. These identifiers are used to allow proper communication between the instruction scheduler and the speculation/synchronization structures. The exact encoding of these fields depends on the implementation of the OoO (out-of-order) execution engine (for example, in a superscalar machine that uses reservation stations we can use the index of the reservation station that holds the instruction as its LDID or STID, or if we want to support multiple loads per store, a level of indirection may be used to represent all loads waiting for a particular store). The instance tag field is used to distinguish between different dynamic instances of the same static dependence edge (in the working example that follows we show how to derive the value for this field). The full/empty flag provides the function of a condition variable.

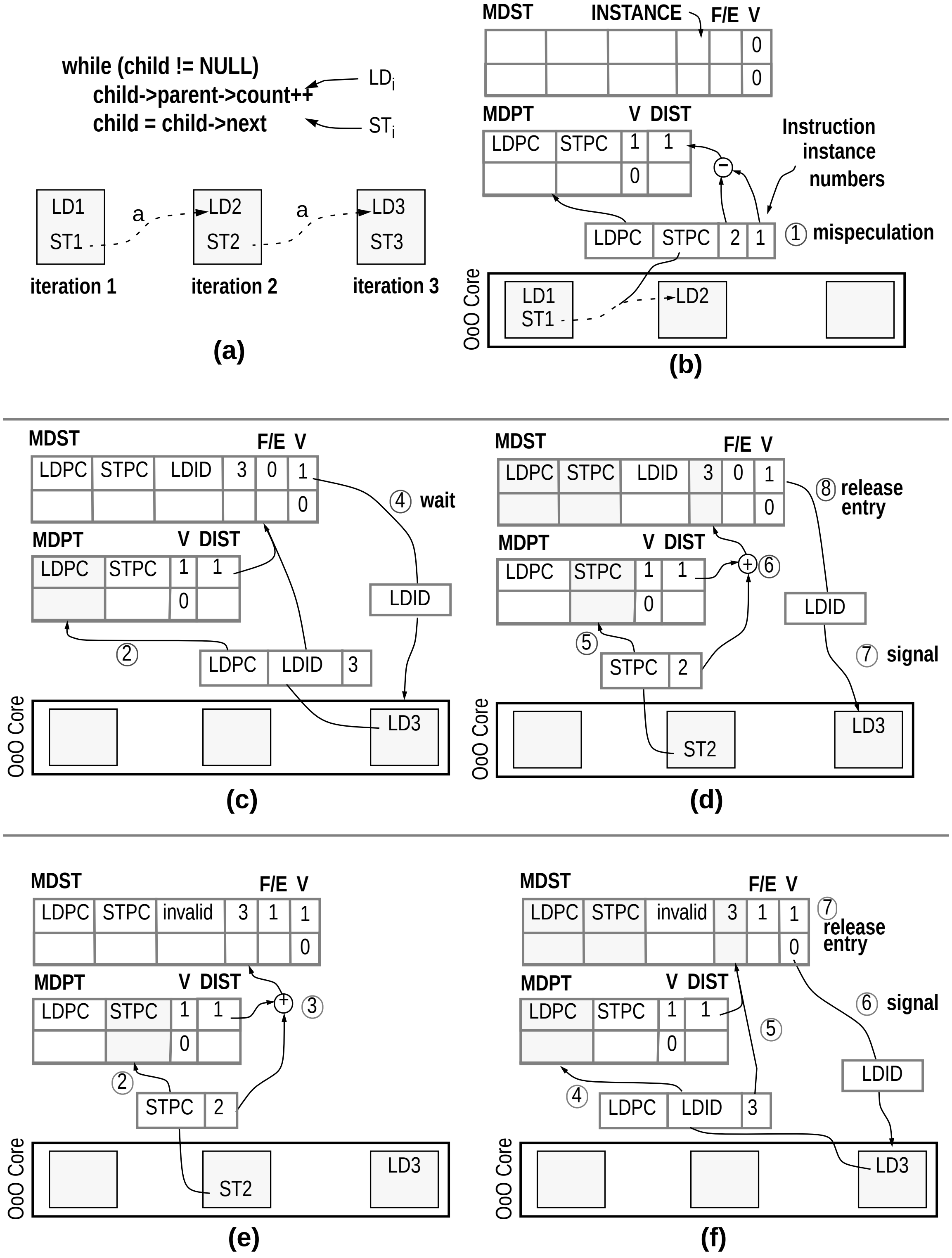
3.5.1 Working Example

The exact function and use of the fields in the MDPT and the MDST is best understood with an example. In the discussion that follows we are using the working example of Figure 3.6. For the working example, assume that execution takes place on a processor which: (1) issues multiple memory accesses per cycle from a pool of load and store instructions and (2) provides a mechanism to detect and correct mispeculations due to memory dependence speculation. For the sake of clarity, we assume that once an entry is allocated in the MDPT it will always cause a synchronization to be predicted.

Consider the memory operations for three iterations of the loop, which constitute the active pool of load and store instructions as shown in part (a) of the figure. Further, assume that *child->parent* points to the same memory location for all values *child* takes. The dynamic instances of the load and store instructions are shown numbered, and the true dependences are indicated as dashed arrows connecting the corresponding instructions in part (a). The sequence of events that leads to the synchronization of the ST2-LD3 dependence is shown in parts (b) through (d) of the figure. Initially, both tables are empty. As soon as a mispeculation (ST1-LD2 dependence) is detected, a MDPT entry is allocated, and the addresses of the load and the store instructions are recorded (action 1, part (b)). The DIST field of the newly allocated entry is set to 1, which is the difference of the instance numbers of ST1 and LD2 (1 and 2 respectively). As we noted earlier, we approximate the instance numbers using the distance in processing units (incidentally this is identical to the instance distance in our example). As a result of the mispeculation, instructions following the load are squashed and must be re-issued. We do not show the re-execution of LD2.

As execution continues, assume that the address of LD3 is calculated before the address of ST2. At this point, LD3 may speculatively access the memory hierarchy. Before LD3 is allowed to do so, its instruction address, its instance number (which is 3), and its assigned load identifier (the exact value of LDID is immaterial) are sent to the MDPT (action 2, part (c)). The instruction address of LD3 is matched against the contents of all load instruction address fields of the MDPT (shown in grey). Since a match is found, the MDPT inspects the entry predictor to determine if a synchronization is warranted. Assuming the predictor indicates a synchronization, the MDPT allocates an entry in the MDST using the load instruction address, the store instruction address, the instance number of LD3, and the LDID assigned to LD3 by the OoO core (action 3, part (c)). At the same time, the full/empty flag of the newly allocated entry is set to empty. Finally, the MDST returns the load identifier to the load/store pool indicating that the load must wait (action 4, part (c)).

When ST2 is ready to access the memory hierarchy, its instruction address and its instance number (which is 2) are sent to the MDPT (action 5, part (d)). (We do not show the STID since, as we later explain, it is only needed to support control speculation.) The instruction address of ST2 is matched against the contents of all store instruction address fields of the MDPT (shown in grey). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. As a result, the MDPT adds the contents of the DIST field to the instance number of the store (that is, 2 + 1) to determine the instance number of the load that should be synchronized. It then uses this result, in combination with the load instruction address and the store instruction address, to search through the MDST (action 6, part (d)), where it finds the allocated synchronization entry. Consequently, the full/empty field is set to full, and the MDST returns the load identifier to the load/store pool to signal the waiting load (action 7, part (d)). At this point, LD3 is free to continue execution. Furthermore, since the synchronization is complete, the entry in the MDST is not needed and may be freed (action 8, part (d)).



```
while (child != NULL)
    child->parent->count++
    child = child->next
```

Figure 3.6: *Synchronization of memory dependences.*

If ST2 accesses the memory hierarchy before LD3, it is unnecessary for LD3 to be delayed. Accordingly, the synchronization scheme allows LD3 to issue and execute without any delays. Consider the sequence of relevant events shown in parts (e) and (f) of Figure 3.6. When ST2 is ready to access the memory hierarchy, it passes through the MDPT as before with a match found (action 2, part (e)). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. However, no matching entry is found there since LD3 has yet to be seen. Consequently, a new entry is allocated, and its full/empty flag is set to full (action 3, part (e)). Later, when LD3 is ready to access the memory hierarchy, it passes through the MDPT and determines that a synchronization is warranted as before (action 4, part (f)). The MDPT searches the MDST, where it now finds an allocated entry with the full/empty flag set to full (action 5, part (f)). At this point, the MDST returns the load identifier to the load/store pool so the load may continue execution immediately (action 6, part (f)). It also frees the MDST entry (action 7, part (f)).

3.6 Issues

We now discuss a few issues which relate to the implementation framework we have described. For the most part we focus on the implementation of memory dependence speculation and synchronization under the Multiscalar execution model. Detailed information about the Multiscalar architecture can be found in [26,14,82,27,40,92,13]. A brief description of the Multiscalar execution model is given next, in Section 3.6.1. The rest of this section is organized as follows: In Section 3.6.2 we discuss where in a typical pipeline it may be possible to incorporate the speculation/synchronization functionality. In Section 3.6.3 we discuss what needs to be done when synchronization is incomplete. In Section 3.6.4 we discuss how a confidence mechanism aimed at improving prediction accuracy can be incorporated into the speculation/ synchronization structures. In Section 3.6.5 we consider what support might be required when speculation/ synchronization is used in conjunction with control-speculation. In Section 3.6.6 we discuss support for predicting and synchronizing multiple dependences per load or store. Finally, in Section 3.6.7 we discuss two distributed implementations of our speculation/synchronization method.

3.6.1 The Multiscalar Execution Model

For the purposes of this work it is sufficient to know that a Multiscalar processor relies on a combination of hardware and software to extract parallelism from ordinary, sequential programs. The effect Multiscalar aims to achieve is illustrated in Figure 3.7. Instead of relying on a large instruction window to scan through the dynamic instruction trace, Multiscalar relies on a collection of smaller instructions windows that simultaneously scan through different parts of the dynamic execution stream. Multiscalar uses a combination of software and hardware techniques to achieve this effect.

In this model of execution, the control flow graph (CFG) of a sequential program is partitioned by the compiler into portions called tasks. These tasks may be control and data dependent. When executed, a task corresponds to a continuous portion of the instructions stream that would have been generated if the program was executed sequentially. A Multiscalar processor sequences through the CFG speculatively, a task at a time, without pausing to inspect any of the instructions within a task. A task is assigned to one of a collection of processing units for execution by passing the initial program counter of the task. Multiple tasks execute in parallel on the processing units, resulting in an aggregate execution rate of multiple instructions per cycle. In this organization, the instruction window is bounded by the first instruction in the earliest executing task (the *head* task) and the last instruction in the latest executing task (the *tail* task). The head and tail task also define an order among the tasks currently executing which corresponds to the sequential, program implied order.

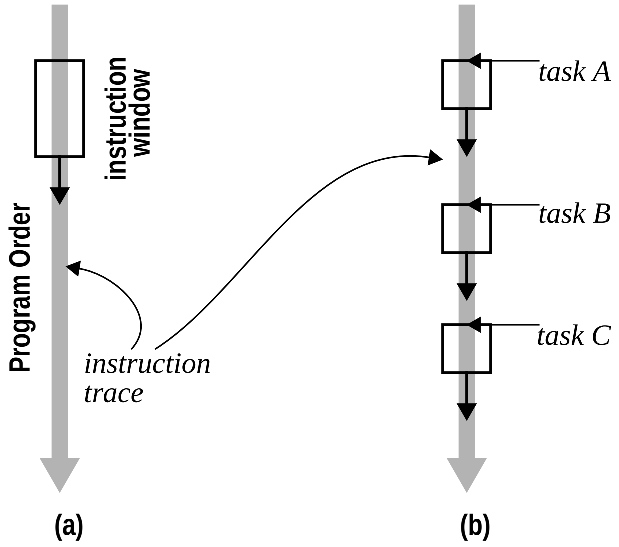


Figure 3.7: *The Multiscalar execution model. (a) Continuous, centralized instruction window (e.g., typical dynamically scheduled superscalar). (b) Multilscalar's way of building a large instruction window.*

To maintain program semantics while allowing the aggressive out-of-order execution of instructions, the model of the Multiscalar we used in this work relies on a combination of software and hardware for detecting data, that is register or memory dependences. To review these policies it is first important to note that under the Multiscalar execution model, data dependences may be characterized as *intra-task* (within a task) or *inter-task* (between individual tasks). For intra-task register dependences, Multiscalar relies on hardware mechanisms similar to those found in a typical, dynamically scheduled ILP processor. For inter-task register dependences, Multiscalar relies on compiler provided aggregate information about the registers read and written by each task. For memory dependences, no information is provided by the compiler. The Multiscalar model we used, relies on address-based disambiguation of intra-task memory dependences. That is a load may access memory only after it is determined that no ambiguous dependences exist with stores from the same task. For inter-task memory dependences, as originally proposed, Multiscalar uses naive memory dependence speculation. Support for inter-task memory dependence speculation and memory renaming, is provided by the *Address Resolution Buffer* (ARB) [26, 27] or by the *Speculative Versioning Cache* [31]. We use the ARB in our experimentation. Multiscalar uses squash invalidation to recover from memory dependence mispeculations. In particular, the execution of all tasks starting from the one that contained the offending load is invalidated. Each of these tasks has to resume execution from the very first instruction of the corresponding task. This implies even work preceding the mispeculated load may be lost on a memory dependence mispeculation.

In the discussion that follows we use the terms "unit" and "stage" interchangeably to refer to the processing elements where individual tasks execute. We also use the terms "memory dependence mispeculation", "memory dependence violation" and "mispeculation" interchangeably. Finally, we use the terms "speculation" and "memory dependence speculation" interchangeably.

3.6.2 Incorporating Speculation/Synchronization into a Pipeline

In the description of our proposed mechanism, we did not discuss where exactly in the pipeline of a dynamically scheduled processor the prediction and synchronization actions should take place. Since our predictor is PC-based, prediction can be initiated as soon as the PC of a store or a load becomes known. Similarly, since the information used to allocate synchronization entries and perform synchronization is PC based, allocation of synchronization entries can be done as early as desired, provided that the PC of the corresponding instruction is available. This property may be useful in overlapping synchronization with address calculation in order to avoid increasing load latency when no dependences exist.

3.6.3 Incomplete Synchronization

So far, we have assumed that any load which waits on the full/empty flag of an entry in the MDST, eventually sees a matching store that signals to complete the synchronization. Since an MDPT entry only provides a prediction, this expectation may not always be fulfilled. If this situation arises, the two main considerations are: (1) to avoid deadlock and (2) to free the MDST entry allocated for a synchronization that will never occur. The deadlock problem is solved if we assume that a load is always free to execute once all prior stores are known to have executed. De-allocating the MDST entry can also be done at this point. Under the Multiscalar execution model, the aforementioned actions can take place when the unit becomes the head (i.e., the oldest executing task).

Under similar circumstances to those described above, a store may allocate an MDST entry for which no matching load is ever seen. Since stores never delay their execution, there is no deadlock problem in this case. However, it is still desirable to eventually free the MDST entry. Unfortunately, we cannot de-allocate this entry when the store retires since this may in turn result in unnecessarily delaying a subsequent load (see discussion of Section 3.4). Under a continuous instruction window execution model, this de-allocation can take place when the next instance of the same static store is retired. In the implementations we consider under the Multiscalar execution model, we de-allocate store entries when the task that is predicted to contain the appropriate load instance commits.

3.6.4 Intelligent Prediction

Upon matching a MDPT entry, a determination must be made as to whether the instruction pair in question warrants synchronization. The simplest approach is to assume that any matching entry ought to be synchronized (i.e., the predictor field is optional). However, this approach may lead to unnecessary delays in cases where the store-load instruction pairs are mispeculated only some of the time. Instead, a more intelligent approach may be effective. Any of the plethora of known methods (counters, voting schemes, adaptive predictors, etc.) used to provide the intelligent prediction of control dependences may be applied, to the prediction of memory dependences. Regardless of the actual choice of mechanism, the prediction method ought to exhibit the quality that it strengthens the prediction when speculation succeeds and weakens the prediction when speculation fails. In this work we restrict our attention to either using counter-based confidence mechanisms or to using non-adaptive confidence mechanisms. As we demonstrate in the evaluation section (Sections 3.8.5 and 3.8.6), performance is superior when an adaptive confidence mechanism is incorporated with each MDPT entry.

To allow for adaptive predictors, we merge the MDST and the MDPT into a single structure, with a fixed number of synchronization variables per MDPT entry. In particular we allow for one synchronization variable per unit in each MDPT entry (the synchronization variables are implemented as a bit vector). Each of the synchronization variables comprises three bits: *wait*, *signal* and *sync*. The wait bit is set when synchronization is predicted on a load. The signal bit is set when synchronization is predicted on a store. The sync bit is set when synchronization takes place, by either a store or a load. At task commit time, each MDPT entry inspects the condition variable that corresponds to the specific unit adjusting the confidence automaton accordingly. Merging the MDPT and the MDST permits this operation to be done locally at each MDPT/ MDST entry, thus allowing all MDPT entries to be updated in parallel. Had we used a split MDPT/MDST design, a mechanism would be required to inspect each MDST entry and then propagate the changes to the corresponding MDPT entry.

Unfortunately, and as we will explain in the Section 3.6.7, in a distributed implementation where the prediction structure is replicated per processing unit, using adaptive predictors is inconvenient. It is so, as information about the success of failure of any synchronization attempt may have to be propagated to all copies of the prediction structure to keep them coherent. The potential bandwidth, the additional complexity and time required to perform this action may prove prohibitive. For this reason, we will not make use of adaptive predictors for the experiments that use a distributed memory dependence speculation and synchronization mechanism.

3.6.5 Control Mispeculations

In the event of control or data mispeculation, it is desirable, although not necessary, to invalidate any MDST entries that were allocated to instructions that are squashed. In the implementations we consider in this work, we do not expose intra-unit control mispeculations to the MDST. In the event of an inter-unit control mispeculation however, all synchronization entries allocated for the units being squashed are de-allocated. In a continuous, instruction window processor, it may possible to incorporate the synchronization functionality in the scheduler used for register dependences (see Section 3.9.4). For example, this can be done by using the id of the reservation station where the store resides to create an artificial, speculative dependence between the store and the load that should wait. Depending on the register scheduler implementation, no additional support may be required to cleanup the synchronization tags on control mispeculations (e.g., if the scheduler is implemented using the RUU model [81]).

3.6.6 Multiple Dependences Per Static Load or Store

Although not illustrated in the examples, it is possible for a load or a store to match multiple entries of the MDPT and/or of the MDST. This case represents multiple memory dependences involving the same static load and/or store instructions (for example in the code “*if (cond) store*$_1$ *M else store*$_2$ *M; load M,*” there are two dependences *(store*$_1$*, load)* and *(store*$_2$*, load)*) which may alternate in the dynamic execution stream. There are three challenges from a practical perspective: (1) how to predict multiple dependences per load, (2) how to allocate multiple MDST entries when multiple dependences are predicted on a single load, and (3) how to wake-up a load forced to wait on multiple MDST entries. One solution would be to define the problem away by tracking only a single dependence per store or load. However, as we will demonstrate in Section 3.8.6, support for multiple dependences per static instruction is very important. We consider two ways of providing this support.

In the first, each MDPT and MDST entry is augmented to track a single load and a plurality of stores as shown in Figure 3.8, part (a) (the use of the “TASK PC” fields is explained later on in this section. In this work we evaluate designs with 2, 4 and 8 stores per entry. This allows us to predict and synchronize multiple dependences per load by consulting a single MDPT entry and by using a single MDST entry. For stores we use the following approach: a separate MDPT entry is allocated per store. No information about the loads with which the store has to synchronize with is kept in the store entry. For synchronization purposes, a store associatively searches through the MDST and synchronizes with any loads that are waiting for it. If no load is waiting, a new entry is created in the MDST, marking that the particular store instance has executed. In Section 3.8.6, we demonstrate that this implementation offers performance very close to the one that does not limit the number of dependences per static instruction. An additional advantage of this implementation is that it does not require associative lookups in the MDPT as a single entry exists per store or load. However, associative lookups are still required the MDST.

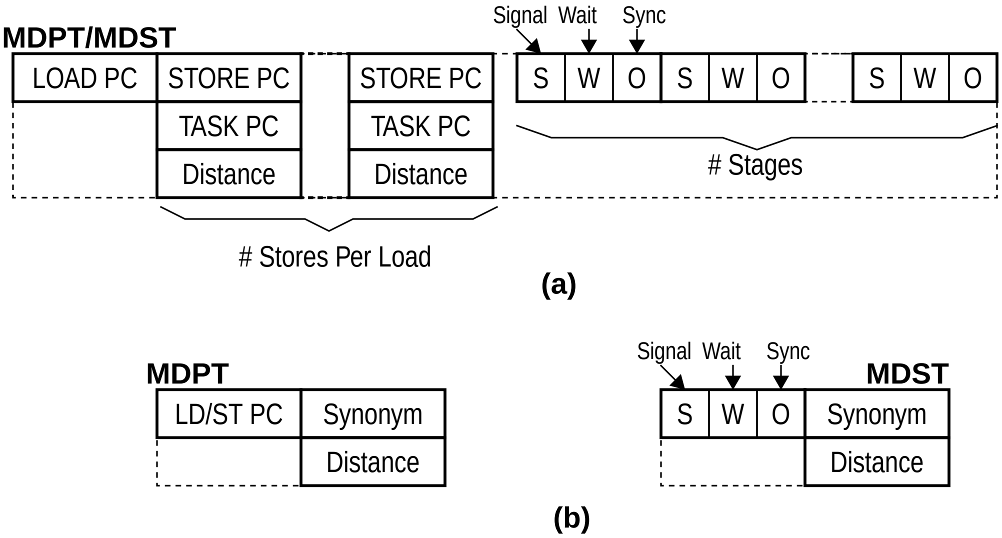


Figure 3.8: *Two schemes of supporting multiple static dependences per load or store. (a) Combined MDPT/MDST with multiple stores per load. (b) Split MDPT/MDST using a level of indirection to represent dependence sets.*

The second scheme uses *a level of indirection* to represent the set of all dependences that have a common store or load (for example in the code "if (cond) store$_1$M; else store$_2$M); load M);" both the (store$_1$, load) and the (store$_2$, load) dependences will be represented using a common tag). This approach was suggested in [62,17]**.** In this scheme, separate entries for loads and stores are allocated in the MDPT. The format of these entries is shown in part (b) of Figure 3.8. (Note that a split MDPT and MDST is illustrated in the figure. As we will explain in the next section, we utilize this split organization for a distributed implementation of our proposed mechanism.) As shown, in these entries, we do not record the dependences the corresponding instructions have. Instead we use a tag, to which we will refer to as a *synonym*. Synonyms are assigned using a global counter when mispeculations occur. If no synonym has been assigned to either the static load or the static store, a new synonym is generated using the global counter and is assigned to both instructions. If a synonym has been already assigned to only one of the instructions (as the result of mispeculating a different static dependence involving that instruction), the same synonym is assigned to the other instruction. If both instructions already have synonyms assigned to them which are different, the smallest one is assigned to both instructions. In this case, and if we were to be precise about the representation of dependences, we would have to replace all instances of the larger synonym with the new synonym (as suggested in our ealier work [62] and under the context of the applications we describe in Chapter 4). However, as this action would probably require an associative search and as suggested by Chrysos and Emer [17], we do not do so. Because the smallest tag is used to resolve conflicts, sooner or later all active dependences that share a load or a store will be assigned the same synonym.

A final consideration is whether a load that has multiple dependences predicted should wait for all or for just one of them. Although we do not report experimental data in support of this observation, we found that allowing loads to execute as soon as one of their predicted dependences is synchronized increases the number of mispeculations observed. This is possible when in the original program order, multiple stores write to the same memory location before the load reads from it. In this case, the load should wait for the last in program order store. However, dynamically, these stores may appear in any order. For this reason, allowing the load to execute as soon as one of these stores has executed may not help in avoiding the mispeculation. Accordingly, we evaluate mechanisms in which a load waits for *all* dependences predicted. However, this scheme is not without problems. We found that while it is very effective in avoiding mispeculations, it often fails to allow loads to execute as early as possible. This primarily happens when the load has multiple dependences which appear through different control flow paths. In this case, the load should wait only for those dependences that are currently active. We found that an effective solution to this problem is to maintain minimal control flow information with each load-store pair. In the case of Multiscalar we augmented the MDPT entries to record the store's task PC in addition to the (load PC, store PC) pair and the unit distance as shown in part (a) of Figure 3.8. In this case, synchronization is predicted only when the task at the predicted unit distance matches that recorded on the prediction table entry. All experiments reported that use the Multiscalar execution model make use of this optimization.

3.6.7 Centralized Versus Distributed Structures

So far we have been assuming that the MDPT and the MDST are centralized structures. However, as greater levels of instruction-level parallelism are exploited, greater numbers of concurrent memory accesses must be sustained. Under such conditions, it is important to assure that neither structure becomes a bottleneck. In the case of a distributed window processor, it is desirable to also partition both the prediction and the synchronization structures. We do consider such an option in this work. In particular, we use an organization where identical copies of the MDPT and the MDST are provided at each processing unit. In this case, the speculation/synchronization mechanism operates as follows: When a mispeculation occurs, the static dependence edge (store PC, load PC), along with the unit distance is sent to all copies of the MDPT. Each copy of the MDPT allocates the appropriate entries (if they do not already exist), one for the load and one for the store, as a centralized organization would do. Since, at any given cycle a single mispeculation can be signaled, a relatively low bandwidth mechanism should be sufficient for this purpose. In the implementations we evaluate we use a bus for this purpose.

When a load is ready to access memory, the local copy of the MDPT is consulted. If a prediction is made that synchronization should take place, the local copy of the MDST is consulted next. The MDST is searched to determine whether the predicted store — represented by either a (store PC, unit id) or a (synonym, unit id) — has already executed. If so, the load is allowed to access memory. If the store has not executed yet, a new entry is allocated in the MDST to indicate that a load of this unit is waiting for the particular store instance. When a store is ready to write to memory, it also consults the local MDPT copy. If synchronization is predicted, the store's identity — as either a (store PC, unit id) or a (synonym, unit id) pair — is send to all copies of the MDST. If loads are found waiting for the particular store, they may now proceed to access memory. If no loads are found waiting, an entry for the store is created in every MDST copy. In the models we consider in the evaluation section we use a bus to signal store execution. Finally, when a task commits it cleans up all local MDST entries.

As we noted in Section 3.6.4, using adaptive predictors in a distributed organization is inconvenient. The reason is that if we were to keep all MDPT copies coherent we would have to broadcast all changes done locally to every other copy of the MDPT. While support for such an approach might be possible, for the purposes of this work we restrict our attention to non-adaptive confidence mechanisms in the distributed MDPT. In particular, we simply use a 2-bit saturating counter that is updated when mispeculations occur (a system wide event). If synchronization fails, no attempt is made to adjust the confidence predictor.

3.7 Related Work

Ultimately, the goal of the techniques we proposed is to allow loads to access memory as early as possible, by scheduling its execution as far in advance from the instructions that need the memory data. The same effect can also be achieved by appropriately scheduling the code at compile time. At the core of all software based load scheduling techniques are static *disambiguation* or *alias analysis* techniques. The goal of these methods is to prove whether a given load and store can be data dependent during run-time. A plethora of techniques has been proposed. Initially research focused primarily on array variables [4,22,10], while recently methods have been proposed for dynamically allocated data types [23, 97].

A plethora of memory dependence speculation techniques has also been proposed. These techniques differ in whether software or hardware is used to: (1) perform load motion, (2) detect dependence violations and (3) recover from dependence violations. Nicolau proposed *run-time disambiguation* [65], a software only approach to dependence speculation. In his technique, loads can be speculatively scheduled before a preceding store with which an ambiguous dependence exists. Code is inserted after the store to detect whether a true dependence is violated (this is done by comparing the addresses accessed by the store and the load), and repair code is also inserted to recover from memory dependence violations. Another software only approach was proposed by Moudgill and Moreno [64]. Their approach differs from Nicolau's in that they compare values rather than addresses to detect violation of program semantics.

Gallagher, Chen, Mahlke, Gyllenhaal and Hwu [28, 16] proposed the *Memory Conflict Buffer (MCB)*, a software-hardware hybrid approach. In their technique, load motion and mispeculation recovery are done in software while mispeculation detection is done in hardware. Two copies of each speculated load are executed, one at the original program order (non-speculative) and the other as early as desired (speculative). Speculative loads record their addresses in the MCB. Intervening stores also post their addresses to the MCB, so that dependence violations are detected. The non-speculative load checks the appropriate entry in the MCB (the target register of the load is used as a handle), and if any dependence was violated, control is transferred to recovery code. Huang, Slavenburg and Shen proposed *speculative disambiguation* [36] another hybrid approach to memory dependence speculation. In their technique multiple versions of the same code are generated, one with speculation enabled and another with speculation disabled. These versions are then scheduled together using predication. Hardware similar to that used for *boosting* [78, 77] is used to invalidate all but the appropriate path during execution.

Naive memory dependence speculation was proposed for the Multiscalar architecture [26]. Support for dependence mispeculation detection and recovery was proposed in the form of the *Address Resolution Buffer* (ARB) [27] which also implements memory renaming. Other recently proposed techniques to support speculation of memory dependences, memory renaming and memory dependence mispeculation detection are presented in [31, 32]. Naive memory dependence speculation was used in the PA8000 processor [38] and in the Power 620 processor [1,50]. Knight also proposed using memory dependence speculation along with a hardware-based mispeculation detection mechanism in the context of speculative, parallel execution of otherwise sequential Lisp programs [47].

Finally, several hardware-based techniques have been proposed that aim at improving accuracy over naive memory dependence speculation. There are two closely related proposals. In the first proposal by Steely, Sager and Fite [84], mispeculated loads and stores are given tags derived from the addresses via which the mispeculation occur. These tags are used by the out-of-order scheduler to restrict load execution. Hesson, LeBlanc and Ciavaglia [33] describe the *store barrier cache* and the *store barrier* approach. An implementation of the store barrier cache was also presented [3]. The techniques we describe in this chapter were also reported [63,61].

Selective speculation is implemented in the Alpha 21264 processor [45] where an *independence predictor* is used to predict whether a load can execute freely. Finally, Chrysos and Emer proposed using a level of indirection for the purposes of memory dependence speculation/synchronization. In their *store set* approach a tag is used to represent the set of all stores that a load has had a memory dependence mispeculation. They proposed the incremental approach we also utilize to build memory dependence sets. Synchronization takes place through a separate table, and moreover, to preclude ordering problems on dependences with non-unit distances and to attain a simple synchronization table design, stores that have been assigned to the same store set (i.e., synonym) are executed in-order.

As we noted in Section 3.2, to reduce the net penalty of mispeculation we could alternatively utilize techniques to either (1) minimize the amount of work lost on mispeculation or (2) reduce the time required to redo the work lost on mispeculation. In the second category falls instruction reuse [79]. The applicability of this technique on distributed, split-window processing models is still under investigation. A technique to reduce the amount of work lost on mispeculation is *selective invalidation*. Selective invalidation aims at invalidating only those instruction that used erroneous data and has been first proposed in the context of load value prediction [54] and for a centralized, continuous instruction window processor. As we demonstrate in the evaluation section, under these assumptions memory dependence mispeculations are virtually non-existent, hence there is no problem with mispeculations and no need for a selective invalidation mechanism. In Chapter 4, we describe a selective invalidation mechanism and use it for a different purpose (i.e., speculating on the origin of load values).

3.8 Evaluation - Distributed, Spit-Window Processor Model

In this section we study our proposed methods using a model of the Multiscalar architecture. This section is organized as follows: we first demonstrate that exploiting load/store parallelism has the potential to improve performance significantly (Section 3.8.1). In Section 3.8.2, we demonstrate that naive memory dependence speculation can be used to extract some of the parallelism that is present, but also demonstrate that the net mispeculation penalty is high. Then, in Section 3.8.3, we consider using an address-based scheduler to extract and exploit load/store parallelism and demonstrate that even if such a device could be incorporated into our processor model, memory dependence mispeculations remain frequent and the net penalty of mispeculation though lower, remains high. For this reason, we next consider techniques to improve the accuracy of memory dependence speculation. In Section 3.8.4, we consider selective memory dependence speculation and demonstrate that even if perfect memory dependence prediction was possible, this technique is not robust. We use the aforementioned results to motivate the use of memory dependence speculation/synchronization. Initially, we consider a centralized implementation that explicitly tracks all memory dependences per static load or store (as detailed in Section 3.5). Having shown that such a mechanism is very effective in reducing memory dependence speculations while maintaining the benefits of aggressive speculation we then consider the two more practical distributed implementations of our proposed mechanism we discussed in sections 3.6.6 and 3.6.7.

Before we proceed into presenting our findings, it is important to discuss: (1) the exact memory dependence speculation model and the compiler support used in our experiments, (2) the methodology we followed to approximate ideal memory dependence speculation.

In all experiments reported in this section, intra-task memory data dependences are not speculated. That is, mispeculations may *only* occur for store-load pairs whose dependence edge crosses task boundaries. Furthermore, the results reflect execution with no compiler supported disambiguation of these memory dependences. This detail implies that even in cases where an unambiguous memory dependence exists, it is treated no differently than an ambiguous memory dependence during execution. At first glance, the reader may be tempted to conclude that the results of this section are not very useful since many dependences could be classified as unambiguous, even with a rudimentary compiler. However, this conclusion is not necessarily correct. Multiscalar, as other dynamically scheduled ILP processors, dynamically converts the sequential program order into a parallel execution order. In this environment, the only condition that prevents the OoO execution of two instructions is the existence of a dependence that the OoO execution engine can detect without executing the instructions. This implies that even if the compiler knows that a particular memory dependence exists, nothing prevents the dynamic speculation of the corresponding load instruction. Consequently, to prevent the speculation of a dependence, the compiler has either: (1) to identify by some means (for example through ISA extensions) that a load should not be speculated immediately and to enforce synchronization between dependent instructions (perhaps by using signal and wait operations on compiler generated synchronization variables or via full/empty bits), or (2) make assumptions about the scheduling properties of the target OoO engine and delay load execution somehow (probably via the use of ad-hoc techniques).

In the experiments that follow we make extensive use of an oracle memory dependence speculation mechanism. The intention is to simulate a mechanism that has perfect, advance knowledge of all dependences and that is capable of utilizing this information to achieve the highest possible performance (i.e., the ideal memory dependence speculation mechanism of Section 3.3). However, we should point out that even when perfect dependence information is available deciding exactly which execution order will result in best performance is a very challenging task, even theoretically (it amounts to scheduling). For this reason, the oracle mechanism we simulated should only be viewed as an approximation of its ideal counterpart. In practice, we simulated the oracle mechanism as follows: (1) loads appearing on the correct control path were allowed to execute as soon as all their dependences were satisfied (i.e., either no dependence exists, or the appropriate store has written its data to memory), and (2) loads on an incorrect control speculated path were allowed to access memory freely as we found this has a prefetching effect that can be significant in terms of performance (control paths tend to re-converge quickly). Even with this policy, we found that primarily for 102.swim and 104.hydro2d other speculation mechanisms that used imperfect dependence information exhibited better performance. One reason why this is so, is that control speculation and the loads that appear on incorrect control speculated paths may be different for different speculation policies Another cause of this anomaly can be traced to the access combining that takes place in the simulated memory system. In our simulator's memory model [13], loads to the same cache bank and block are combined into a single request when they appear simultaneously on the bus. This combining may help performance as it reduces the number of data cache requests and hence contention for data cache ports. Sometimes, delaying load execution by a couple of cycles increases the probability that combining will take place. Unfortunately, we found no straightforward way of predicting which particular execution schedule will (1) allow loads to execute as early as possible, (2) increase combining as much as possible, and (3) send those load requests on incorrectly control speculated paths that have a prefetching effect. To compensate for this limitation in our experiments we calculated the performance of the oracle speculation mechanism by taking the maximum performance obtained with any of the policies studied.

3.8.1 Performance Potential of Load/Store Parallelism

In this section we demonstrate that significant performance improvements are possible if we can exploit load/store parallelism. To do so, we compare the performance of (1) a configuration that makes loads wait until it is known that all preceding stores have executed (i.e., it does not speculate on memory dependences), and (2) the performance possible with an oracle dependence speculation mechanism. With the first policy, a load is allowed to access memory under the following two conditions: (a) preceding stores from the same task can supply the data needed by the load, and (b) if (a) is not true, when the load's task is the head task. Figure 3.9 reports the relative performance with oracle disambiguation compared to the configuration that does not exploit load/store parallelism. Two configurations are studied, one with 4-units and one with 8-units (these configurations were detailed in Chapter 1). We can observe that exploiting load/store parallelism has the potential for significant performance improvements, which are larger for the larger instruction window machine (8 stages). We can also observe that the performance improvements are typically higher for the floating-point programs. This behavior can be attributed to the frequency of RAW dependences that are visible from within the instruction window. As we have seen in Chapter 2 (Section 2.3.2.2), the frequency of RAW dependences is typically much lower for the floating-point codes when we consider relatively short instruction distances. The performance potential of exploiting load/store parallelism is higher in this case, as more load/store parallelism exists.

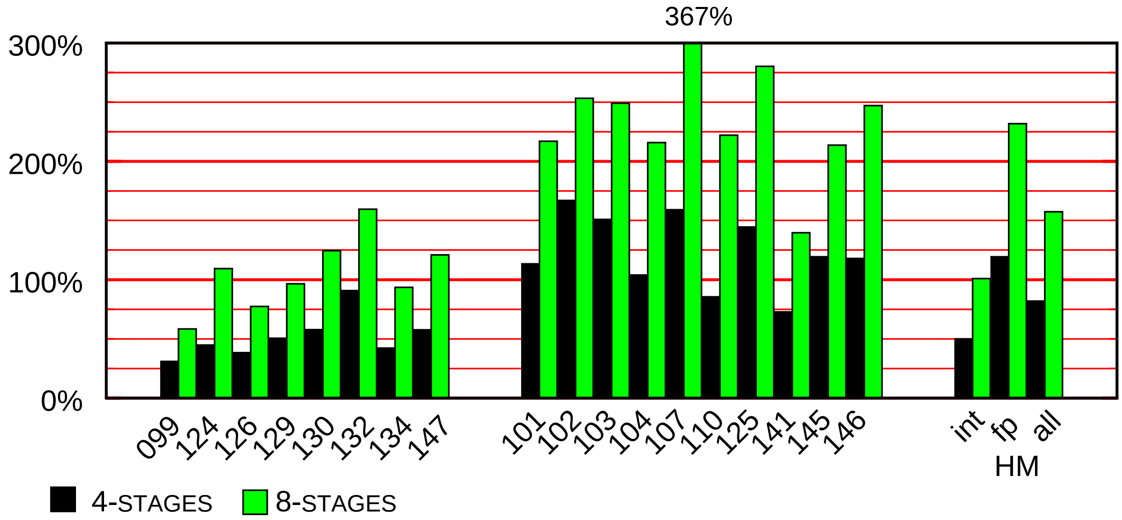


Figure 3.9: *Comparing no speculation with oracle speculation. Shown are the speedups obtained with oracle memory dependence speculation over no speculation.*

The metrics reported in Table 3.1 provide additional insight on the performance behavior shown in Figure 3.9. In this table we report the IPC and the frequency of false memory dependences for the base configuration, i.e., the one that does not speculate on memory dependences. We account for false dependences once per committed load and at the time the load has calculated its address, has resolved all local dependences and could otherwise access memory. At this point we check whether a true dependence exists with a store from a preceding unit that has yet to write to memory (the store may have not been encountered yet). If no such store exists, we count a false dependence as no true dependence would have been violated if the load was allowed to execute immediately. We report false dependences as a fraction over all committed loads.

Comparing the IPC of the 4-unit configuration with that of the 8-unit configuration we can observe that virtually no improvement results from a larger instruction window when load/store parallelism is not exploited. This in part is the reason why the performance potential of oracle dependence speculation is significantly higher for the 8-unit configuration. Focusing on the false dependence frequency measurements we can observe that they are relatively high, which in part explains why the no-speculation configuration is low performing. A phenomenon, which may seem surprising, is that the frequency of false dependences sometimes drops when we move to the 8-unit configuration. What happens, is that in the 8-unit configuration many more memory dependences become visible from within the instruction window. For this reason, more loads do have a true dependence at the time they could otherwise access memory. This observation provides a hint on why speculation/synchronization might be more useful —as opposed to selective speculation— as the instruction window increases. With a larger instruction window, more loads have dependences, and most of these dependences are between stores and loads that are quite distant. Simply not speculating loads with dependences (i.e., selective speculation) may result in many loads waiting for significantly longer then they should.

	False Dependences%		Base IPC			False Dependences%		Base IPC	
	4	8	4	8		4	8	4	8
099	33.1%	32.7%	1.00	1.01	***101***	16.6%	16.6%	1.12	1.12
124	75.8%	77.3%	1.45	1.49	***102***	21.1%	21.1%	1.09	1.09
126	51.7%	47.3%	1.12	1.15	***103***	15.9%	15.8%	1.10	1.10
129	45.4%	40.9%	1.13	1.13	***104***	53.0%	53.2%	1.00	1.01
130	61.0%	58.3%	0.97	0.98	***107***	3.29%	1.27%	1.27	1.27
132	35.4%	43.0%	1.42	1.43	***110***	21.7%	21.9%	1.17	1.18
134	54.7%	49.4%	1.21	1.22	***125***	50.2%	50.7%	1.26	1.26
147	55.8%	45.1%	1.18	1.24	***141***	35.2%	34.2%	1.10	1.11
					145	27.2%	25.6%	1.03	1.03
					146	33.22%	32.77%	1.17	1.17

Table 3.1: *Characteristics of the no-speculation configurations. Shown are the frequency of false dependences, and the IPC. Two configurations are shown with 4 and 8 units respectively.*

3.8.2 Naive Memory Dependence Speculation

In this section we study how naive memory dependence speculation affects performance. Specifically, we demonstrate that: (1) naive speculation can be used to exploit some of the load/store parallelism offering most the performance benefits possible with oracle speculation, and (2) the performance difference between naive and oracle speculation can be significant especially for a wider instruction window configuration.

Figure 3.1 reports the performance improvements obtained when naive memory dependence speculation is used over the same configuration that does not speculate memory dependences (“no speculation” configuration of the previous section). Also shown are the performance improvements possible with oracle speculation. We can observe that naive memory dependence speculation is quite effective. For most programs, naive speculation offers most of the performance improvements possible with oracle speculation. However, the difference between naive and oracle is significant. The latter observation is more clearly shown in Table 3.2 where we report: (1) the speedups possible with oracle speculation over naive speculation, and (2) the memory dependence misspeculation rates with naive speculation. We measure the memory dependence mispeculation rate by diving the number of all observed memory dependence mispeculations with the number of all committed loads. In this calculation we do not include mispeculations encountered on loads that appear on incorrectly speculated control paths.

We observe that the performance difference between naive and oracle speculation can become significant. Moreover, this performance difference is larger for the 8-stage configuration. As we have noted earlier, the net penalty of dependence mispeculations is the cause of this phenomenon. As it can be seen there is not

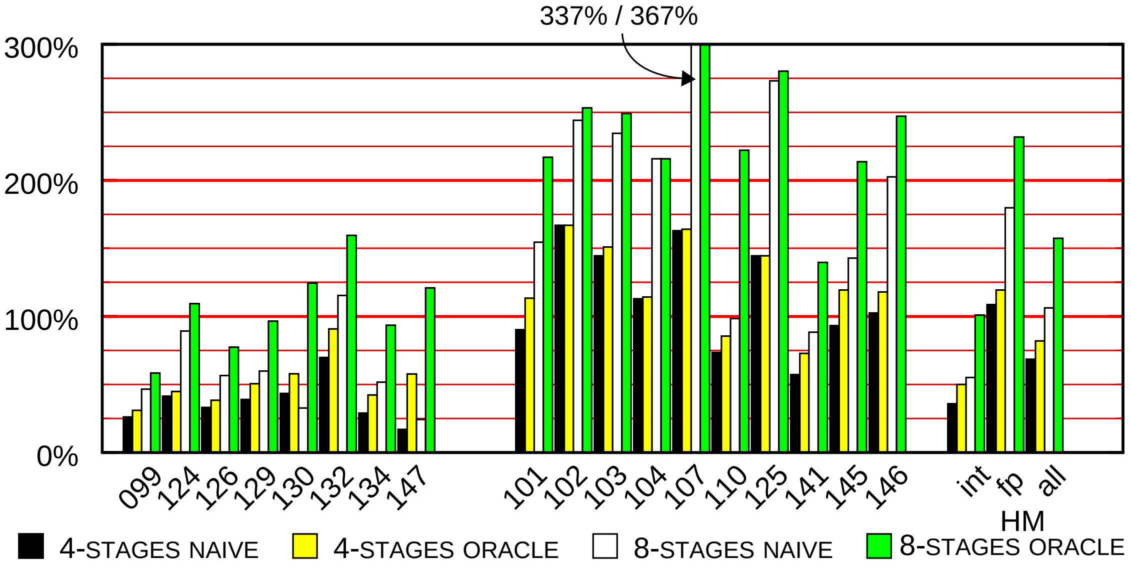


***Figure 3.1:** Performance with naive memory dependence speculation relative to no-speculation.*

necessarily a direct correlation between the mispeculation rate and the performance difference between oracle and naive. The reason is that the mispeculation frequency only indicates how often mispeculations happen. It does not however indicate the amount of work lost or the performance improvement possible via exploiting load/store parallelism. For example, consider 099.go and 146.wave5 on the 8-stage configuration. While 099.go exhibits a mispeculation rate of 3.6% had we avoided these mispeculations we could have improved performance by 8.11%. In contrast, performance for 146.wave5 would improve by 29.20% even though the mispeculation rate is a mere 0.9%. These two programs exhibit quite different execution characteristics. 099.go has relatively poor control prediction behavior and moderate levels of load/store parallelism. 146.wave5 on the other hand, exhibits excellent control prediction behavior and the potential benefits from load/store parallelism are much higher (roughly 2 times as much compared to 099.go).

The results of this section suggest that while naive speculation can be used to extract most of the performance benefits possible by exploiting load/store parallelism, it also suffers from the net penalty incurred due to dependence mispeculations. For this reason, techniques that aim at improving the accuracy of memory dependence speculation, as the ones we have earlier proposed have the potential of improving performance, often significantly. Moreover, the results of this section suggest that the potential performance improvements increase for larger instruction window processors.

3.8.3 Using Store Address Information To Improve Speculation Accuracy

So far, we have been assuming that loads cannot inspect the addresses of stores from other units. However, if that information was available somehow, some of the memory dependence mispeculations could be avoided. For the purposes of this experiment, we are not concerned whether building such a mechanism is possible and if so what implications such a mechanism might have on clock cycle, load latency and design complexity. Rather, we optimistically assume that store addresses can be made visible to all other units with an 1 cycle delay. No limit is imposed on the number of stores that can post their addresses every cycle or on the number of loads that can inspect store addresses. Moreover, posting a store's address does not consume any resources. A load is allowed to execute as soon as its address becomes available and there is no preceding store that writes to the same memory address. However, a load may execute if there are preceding stores

	Speedup% Oracle over Naive		MR%			Speedup% Oracle over Naive		MR%	
Stages	**4**	**8**	**4**	**8**	***Stages***	**4**	**8**	**4**	**8**
099	3.97%	8.11%	2.4%	3.6%	***101***	12.20%	24.56%	1.6%	2.1%
124	2.44%	10.64%	1.8%	3.1%	***102***	0.00%	2.67%	0.2%	0.2%
126	4.03%	13.33%	2.8%	4.3%	***103***	2.60%	4.35%	0.4%	0.5%
129	8.3%	22.90%	5.6%	9.3%	***104***	0.00%	0.00%	0.4%	0.5%
130	10.07%	69.23%	4.9%	7.0%	***107***	0.00%	6.80%	0.1%	0.3%
132	12.45%	20.45%	2.8%	3.2%	***110***	6.90%	62.39%	1.5%	1.6%
134	10.25%	27.56%	4.3%	6.3%	***125***	0.00	1.91%	0.1%	0.1%
147	34.78%	77.92%	6.8%	7.3%	***141***	9.83%	27.27%	1.4%	2.0%
HM int	10.05%	31.21%			***145***	7.59%	29.20%	0.9%	1.2%
					146	13.57%	14.68%	0.7%	0.9%
					HM fp	4.37%	17.35%		

Table 3.2: *Comparing naive and oracle speculation. Shown are the speedups possible over naive speculation with oracle speculation and the frequency of memory dependence mispeculations (MR% columns).*

that have yet to calculate their address. (Note that since tasks are fetched and executed in parallel, it possible to issue a load before a preceding in program store has even been fetched.) We make use of these optimistic assumptions about store address availability in order to demonstrate that memory dependence mispeculations remain frequent enough to justify the use intelligent memory dependence speculation techniques such as those we propose.

The results of this experiment are shown in Table 3.3. The “AV” columns report relative performance over naive speculation when store addresses are available for load inspection. The “oracle” columns report the speedups possible with the oracle speculation mechanism. Two processor configurations are simulated. The first is the default 8-unit configuration we have been using in all preceding experiments. The processing units in this configuration are equipped each with a 16-entry window, an 8-entry scheduler, 1 memory port and 2 copies of all other functional units. This configuration is marked as “1x8” in the table. The other configuration, which is marked as “4x8”, has also 8-units, but its units have 32-entry windows (all entries are visible to the scheduler) and four copies of all functional units (4 memory ports also). Moreover, the bandwidth of the L1-data cache and the ARB is quadrupled. We include this second configuration to increase confidence on the observations we make in this section. In particular, we aim to address the concern on whether, the relatively small instruction window sizes used in by our default configuration (16 instructions and 8 instruction scheduler) impose artificial delays on store address calculation latency. With such small instruction windows, it is possible for a store that could otherwise calculate its address to get delayed just because there isn’t enough space in the local instruction scheduler.

From the results shown in Table 3.3 we can observe that exposing store addresses has the potential of reducing mispeculations and improving performance over naive speculation. However, when compared with the performance potential of oracle speculation, we can observe that for most programs there is still a significant performance difference. This is can also be seen by the average (harmonic mean) speedups possible for the two policies. For example, with the "4x8" configuration the performance improvements when store addresses are exposed are 16.2% and 4.6% for the integer and the floating-point programs respectively. With oracle speculation the performance improvements rise to 24.2% and 10.3% respectively.

	Speedup% over Naive					Speedup% over Naive			
	1x8		4x8			1x8		4x8	
	AV	oracle	AV	oracle		AV	oracle	AV	oracle
099	0.7%	8.1%	4.5%	8.5%	***101***	8.1%	24.6%	8.9%	25.6%
124	1.8%	10.6%	6.4%	8.1%	***102***	0.5%	2.7%	-0.6%	0.0%
126	3.3%	13.3%	8.7%	12.6%	***103***	0.3%	4.3%	0.0%	2.9%
129	5.6%	22.9%	13.4%	24.1%	***104***	0.3%	0.0%	0.9%	0.9%
130	44.6%	69.2%	27.3%	36.8%	***107***	1.3%	6.8%	0.3%	0.8%
132	13.6%	20.5%	22.2%	27.0%	***110***	3.4%	62.4%	17.2%	30.4%
134	6.5%	27.6%	12.3%	26.1%	***125***	0.0%	1.9%	2.1%	2.4%
147	5.2%	77.9%	44.1%	70.3%	***141***	4.3%	27.3%	13.6%	26.9%
HM int	8.8%	31.2%	16.2%	24.2%	***145***	6.4%	29.2%	4.8%	22.2%
					146	5.9%	14.6%	2.4%	3.6%
					HM fp	3.0%	17.3%	4.6%	10.3%
					HM all	5.5%	23.5%	9.5%	16.1%

Table 3.3: *Impact of exposing store addresses on performance. Relative performance over naive speculation is shown. The "AV" columns report performance when store addresses are visible and the "oracle" columns report performance with oracle speculation. Two processor configurations are simulated per policy. The "1x8" is the configuration we have used in all previous experiments, while the "4x8" uses a memory system that has four times more bandwidth, and uses units that have 32 entry windows and 4 copies of all functional units including memory ports.*

Before we proceed to the next section we should explain why an anomaly is observed for 104.hydro2d and the "1x8" configuration. In this case, performance when store addresses are exposed is slightly (i.e., 0.3%) higher compared to oracle speculation. A slight increase in the number of memory accesses that are combined in the memory system are the cause of this behavior. Since when store addresses are exposed, some loads may get delayed, accesses that would otherwise proceed at different times may now proceed simultaneously benefiting for the combining that takes places at the bank of the simulated memory system (see discussion of Section 3.8, last paragraph). Even so, the difference in performance is extremely small. Another anomaly is observed for 102.swim and for the 4x8 configuration, where a small performance degradation is observed when store addresses are exposed. The probable cause of this phenomenon is delayed or no execution of loads as the result of control speculation. A load may get delayed by a store that is on an incorrectly speculated intra-unit control path. In this case, the load may get unnecessarily delayed until that store is squashed locally at its unit. The same applies when both a store and a load are on an incorrectly speculated control path (inter-unit). In this case, the load which is later squashed could have a prefetching effect if it was allowed to execute before the corresponding store (as it could be the case when no address information is available).

3.8.4 Selective Memory Dependence Speculation

In this section we demonstrate that selective memory dependence speculation is not a robust technique. For this purpose we assume perfect memory dependence prediction and demonstrate that even under such optimistic assumptions about prediction accuracy, selective speculation may result in lower performance compared to naive speculation. As we explained in Section 3.3, under this policy, a load that has a dependence is forced to wait for all preceding stores. As a result, the load may get delayed unnecessarily. Figure 3.1 reports the relative performance of selective speculation over naive speculation. We can observe that while selective speculation improves performance for some of the programs, for most of them it either does not affect performance by much or results in performance degradation. For the latter programs it is often the case that it takes less time to incur a misspeculation and re-execute the code rather than having to wait until all previous tasks commit. For 130.li and 147.vortex, selective speculation improves performance significantly. As we have seen (Table 3.2) these two programs exhibit relatively high mispeculation rates (7.0% and 7.3% respectively for the 8-unit configuration) and also the potential benefits are quite high (approximately 69% and 78% respectively). For most floating point programs selective speculation often results in significant performance degradation. As we have seen, these programs exhibit relatively low mispeculation rates. Avoiding those few mispeculations by stalling the corresponding loads, often results in stalling execution in the corresponding unit. This is mostly the result of: (1) the relatively small instruction windows employed in each unit, and (2) the relatively larger task sizes for the floating-point programs [92].

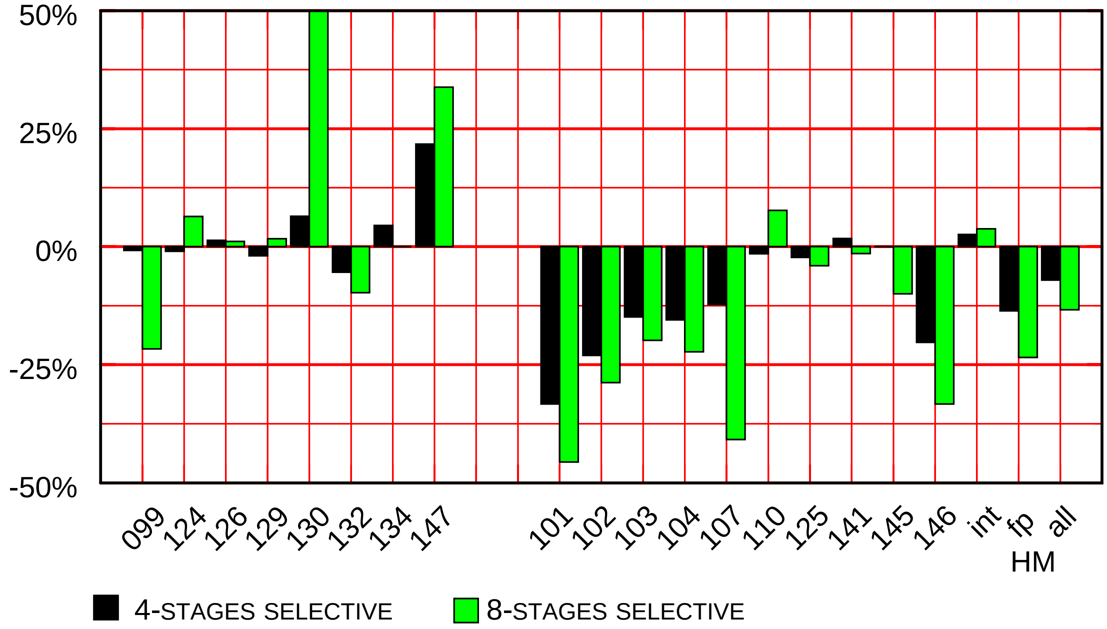


Figure 3.1: *Relative performance of selective memory dependence speculation over naive memory dependence speculation.*

Even though we do not report this result here, we have found [61] that in a configuration that used relatively larger local instruction windows (i.e., 64-entry per unit where all entries were visible to the scheduler) and for the integer programs of SPEC92, performance with selective speculation was often better compared to naive. However, even then selective speculation sometimes resulted in inferior to naive speculation performance supporting our observation that selective speculation is not a robust technique. This observation provides an indication that the performance degradation observed with selective speculation might also be the result of stalling execution due to the limited size of the unit instruction schedulers used in our configuration.

The results of this section suggest that while selective memory dependence speculation can potentially reduce the number of mispeculations observed, it falls short of the second goal of memory dependence speculation which is to delay loads only as long as it is necessary.

3.8.5 Speculation/Synchronization - Centralized Mechanism

Motivated by the observations that (1) the net penalty of mispeculation with naive speculation is significant, and (2) selective speculation is not robust, in this section we study performance under our proposed speculation/synchronization technique. In particular, we demonstrate that a centralized implementation of this technique can be used to attain performance that is very close to that possible with oracle speculation. The rest of this section is organized as follows: first we provide information on the operation and organization of the mechanism we simulated. Then we present a breakdown of the memory dependence status prediction on loads. This result indicates whether we correctly identify if a load should wait or not. We next report the mispeculation rates observed with speculation/synchronization mechanism. This result indicates whether we successfully avoid mispeculations on those loads that have dependences predicted. Finally, we report performance results as ultimately the utility of the proposed technique can only be judged when performance is taken into consideration. This result provides an indication on whether we successfully synchronize with the appropriate store (or stores) and attain the performance benefits of aggressive out-of-order execution of loads and stores. In the experiments that follow we first study an implementation capable of recording up to 256 load/store pairs. At the end of this section we study how performance varies for other prediction table sizes. All MDPT/MDST structures we consider in this section are fully-associative. In Section 3.8.6, we consider an implementation that uses a 2-way set-associative structures.

The mechanisms we study in this section employ a combined MDPT and MDST in which a vector of synchronization bits are associated with each static dependence as explained in Section 3.6.4. Along with each entry, a 3-bit saturating counter with a threshold value of three is used for prediction purposes; synchronization is enforced only when the counter is above three. Counter updates occur when memory dependence violations are detected, or when a task commits by inspecting the corresponding wait, signal and sync bits and as explained in Section 3.6.4. Along with each static dependence we also record the distance in units between the corresponding store and load. This information is recorded when a memory dependence violation is serviced. Also, recorded is the task PC of the store instruction which is used to avoiding predicting synchronization on dependences that cannot possibly appear on the currently predicted control path (as explained in Section 3.6.6). Moreover, the structure is fully-associative and uses LRU replacement. Finally, in all experiments that follow we restrict our attention to the 8-unit configuration where both the potential for performance improvement and the number of mispeculations observed is higher (i.e., more strain is placed on the prediction and synchronization mechanisms).

We first report results on the prediction accuracy of the memory dependence predictor used. In this case we are interested on whether the predictor correctly predicts the dependence status of loads instructions. This is the first step in speculation/synchronization. While memory dependence status prediction is a binary decision problem as is, for example, branch prediction, a single number is not very useful in interpreting memory dependence prediction results. The reason is that, when prediction is incorrect there are two possibilities: (1) either we will make a load wait for more than it should, or (2) we will fail to make a load wait and thus incur a memory dependence mispeculation. Accordingly, we present prediction accuracy results into which predictions are classified into four categories depending on whether: (1) a dependence is predicted, and (2) a dependence really exists. In the results shown we include the predictions made on those loads that were either committed or were invalidated as the result of a dependence mispeculation. We do not include the predictions made on those loads that were squashed by control mispeculations. Predictions are recorded once per dynamic load and at the time the load is ready to access the memory hierarchy. Furthermore, for those loads on which a dependence is predicted, the prediction is recorded after we have checked the synchronization entries for the first time (as we discussed in Section 3.4, stores enable the synchronization bit even when no load is currently waiting). Table 3.4 reports the breakdown of predictions in the form "Predicted/Actual". Correct predictions fall under the "Y/Y" and "N/N" categories. False dependences correspond to the "Y/N" category. We should explain that when compared to the number of mispeculation observed with naive speculation (Table 3.2) these results may be different. The reason is the execution now progresses in a different manner. Some mispeculations are avoided and as a result more dependences are exposed which may or may not get mispeculated. We can observe that the particular memory dependence predictor correctly predicts most of the loads that do have dependences ("Y/Y" vs. "N/Y"). As it can be seen, the fraction of loads that have dependences and this predictor misses ("N/Y" column) is less than 0.26% for all programs. Moreover, false dependences are rather infrequent.

	Predicted/Actual %			
	N/N	**N/Y**	**Y/N**	**Y/Y**
099	93.886	0.126	3.253	2.735
124	95.728	0.144	0.091	4.037
126	91.726	0.028	2.139	6.107
129	89.350	0.017	0.114	10.519
130	89.177	0.026	0.100	10.696
132	94.913	0.112	0.083	4.892
134	91.230	0.058	0.030	8.682
147	81.147	0.019	1.954	16.880
101	95.458	0.009	0.955	3.578
102	97.484	0.000	2.148	0.368
103	98.831	0.001	0.256	0.912
104	96.445	0.000	3.197	0.358
107	97.588	0.002	1.340	1.071
110	96.535	0.047	0.268	3.150
125	99.622	0.000	0.216	0.162
141	96.320	0.068	0.853	2.759
145	95.263	0.001	0.975	3.761
146	97.874	0.260	0.268	1.597

Table 3.4: *Breakdown of memory dependence status prediction on loads.*

Predicting whether a load has a dependence is the first step in speculation/synchronization. The next is avoiding a dependence mispeculation by synchronizing with the appropriate store. A metric of how successful our mechanisms is at attaining this goal is the dependence mispeculation rate. These results are shown in Table 3.5. Also shown are the mispeculation rates with naive speculation. We can observe that the synchronization/speculation mechanism reduces mispeculations by at least an order of magnitude for all programs except 099.go, 126.gcc, 147.vortex, 103.su2cor and 104.hydro2d. As we shall demonstrate later in this section, for the first three of these programs, most of the mispeculations that are not avoided are the result of limited space in the prediction/synchronization structure. These three programs have larger dependence working sets. Most of the mispeculations that are not avoided in 103.su2cor and 104.hydro2d are mostly the result of incorrectly synchronizing loads with an earlier instance of the corresponding store. However, the absolute mispeculation rates for these two programs are extremely small to start with.

	Mispeculation Rate			Mispeculation Rate	
	Synchronization	Naive		Synchronization	Naive
099	2.1%	3.6%	***101***	0.1%	2.1%
124	0.029%	3.1%	***102***	0.0022%	0.2%
126	1.1%	4.3%	***103***	0.2%	0.5%
129	0.5%	9.3%	***104***	0.2%	0.5%
130	0.014%	7.0%	***107***	0.025%	0.3%
132	0.041%	3.2%	***110***	0.012%	1.6%
134	0.013%	6.3%	***125***	0.0059%	0.1%
147	1.0%	7.3%	***141***	0.3%	2.0%
			145	0.3%	1.2%
			146	0.008%	0.9%

Table 3.5: *Memory dependence mispeculation rates as a percentage over all committed loads with speculation/synchronization ("Synchronization" column) and naive speculation.*

We finally measure the performance improvements obtained through the use of our mechanism and compare it against the performance potential of oracle speculation. Figure 3.1 reports speedups over naive speculation for our mechanism ("synchronization") and oracle speculation. We can observe that for most programs, our mechanism is capable of extracting most of the performance potential of oracle speculation. However, for 099.go, 126.gcc, 147.vortex, and to a lesser extent for 103.su2cor, 141.apsi, and 145.fpppp, our mechanism is not as close to oracle speculation. As will we observe next most of this potential is lost due to limited space in the prediction tables.

Finally, we study how the performance of our proposed mechanism varies as a function of the number of static dependences it can track. Figure 3.2 shows the relative performance of speculation/synchronization with respect to oracle speculation for various predictor sizes. We vary the number of the MDPT/MDST entries from 64 to up to 2048 (shown from left to right, grey bar marks the 64-entry table). We can observe that even with a 64-entry table performance is close to the oracle speculation for most programs. The only programs that seem to be quite sensitive to the number of static dependences we can track are 099.go, 126.gcc, 147.vortex, 141.apsi and 145.fpppp. These programs benefit from larger prediction/synchronization structures. With the exception of 141.apsi performance is within 1.5% of oracle speculation with the 2K-entry table for these programs. Performance for 141.apsi levels off at roughly 2% lower of oracle for tables with 1K entries or more. Similar behavior is exhibited by other programs. There are two reasons why this is so: (1) sometimes a load is forced to wait although no dependence exists, and (2) prediction may fail to properly identify the appropriate store instance a load should wait for. In these two cases, loads are forced to wait for more than they should.

The results of this section demonstrate that a centralized implementation of our proposed mechanism, capable of tracking up to 256 dependences can be used to improve the accuracy of naive speculation and attain some and for the majority of the programs studied, most of the performance benefits possible with oracle speculation.

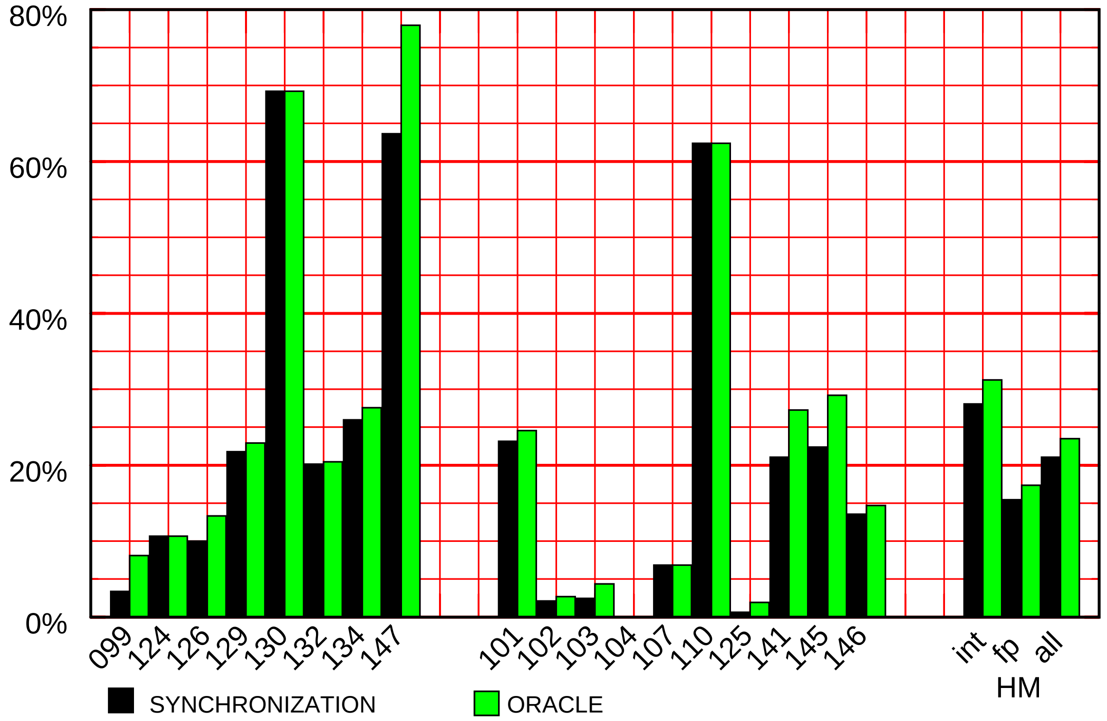


Figure 3.1: *Performance improvements over naive speculation with our speculation/synchronization mechanism capable of tracking 256 dependences (dark bars) and with oracle speculation (grey bars).*

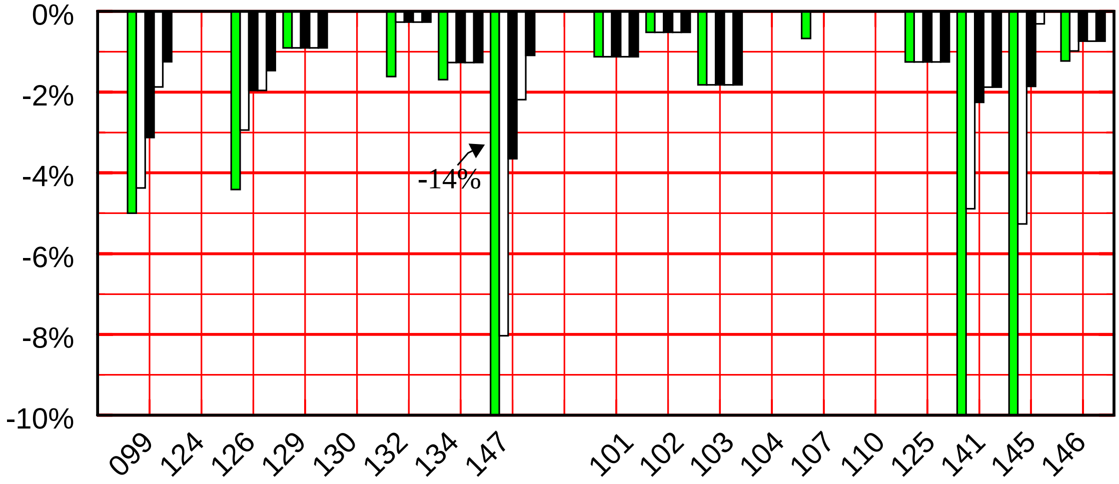


Figure 3.2: *Relative performance of speculation/synchronization with respect to oracle speculation and as a function of the number of entries in the MDPT/MDST. Range shown is 64 to 2K entries, from left to right in steps that are powers of two. The gray bar (left-most per benchmark) marks the 64-entry table.*

3.8.6 Speculation/Synchronization - Distributed Mechanisms

In the previous section we studied the characteristics of a centralized, fully-associative implementation of our proposed technique. As discussed in Section 3.6, this mechanism has a number of features that are undesirable from a practical perspective. In this section we study the two distributed organizations of our proposed technique we discussed in Sections 3.6.6 and 3.6.7. The first mechanism differs from the one we used in the previous section in that: (1) a local copy of the combined MDPT/MDST structure is placed in every unit, (2) separate entries are allocated for loads and stores, (3) multiple stores can be recorded per load, (4) system-wide, only one store can signal for synchronization purposes every cycle (so that a bus can be used — while do not report this result we note that for the configuration studied no noticeable difference in performance was observed when we limited store signal bandwidth to one per cycle in the centralized implementation), and (5) a non-adaptive counter (i.e., it only counts up) is utilized for prediction purposes (threshold is 3). The second mechanism has the following characteristics: (1) it utilizes a level of indirection to identify the set of static dependences each load or store has, (2) it also uses the same 2-bit non-adaptive counters per prediction table entry, (3) it also limits the number of stores that can signal synchronization to one per cycle, and (4) uses separate MDPT and MDST structures.

Table 3.6 reports the relative performance of the first distributed mechanism over its centralized counterpart. Both the distributed and the centralized mechanisms use 256-entry combined MDPT/MDST structures. We include results for four different predictors which are capable of recording, 1, 2, 4 or 8 dependences per static load respectively. We can observe that the predictor that can record just one dependence per load does not perform very well for most of the programs. However, as we increase the number of dependences that can be recorded per load, performance improves. Some programs are very sensitive to the number of static dependences that can be tracked per load (for example, 147.vortex and 110.applu). For the integer codes, most of this sensitivity can be attributed to loads that have multiple dependences that appear on different control paths. For 110.applu the sensitivity is mostly the result of loop unrolling. This program spends a significant portion of its time executing a multi-nested loop that exhibits a recurrence. The distance among the iterations that are dependent through this recurrence varies over time. As a result of loop unrolling the static load and store involved are copied multiple times, and this in turn results in dependences between every copy of the load and store.

The experiments with the first distributed implementation of our synchronization/speculation mechanism suggest that performance is most sensitive to the number of dependences that can be tracked per load and not so much to the distribution of the prediction and synchronization structures. We have shown that a distributed implementation capable of tracking of up to 4 static dependences per load, resulted in performance that was only 3.17% in the worst case and 0.76% on the average, less than that attained with a centralized implementation.

We next report performance results with the second distributed implementation that utilizes a level of indirection to represent the set of all possible dependences per store or load. The particular mechanism we simulated includes a 4K 2-way set associative MDPT and a 64-entry 2-way set associative MDST. Table 3.7 reports performance with respect to the centralized mechanism of the previous section that has 256-entries. It can be seen that for most programs the performance degradation is typically higher when compared to the first distributed mechanism that can track two static dependences per load. The cause can be traced to loads that have multiple dependences that appear at different unit-distances. Since the second mechanism records a single unit distance per static load it fails to synchronize with the appropriate store when the latter appears at unit distances that vary over time. In those cases, the load is either eagerly synchronized with an earlier store, or the load is forced to wait for a store that never appears (i.e., in this case our mechanism degenerates to selective speculation). A potential solution to this problem would be to keep a distance vector along with each entry to allow for multiple unit distances to be predicted. However, we do not investigate this possibil-

	Stores Per Load					Stores Per Load			
	1	2	4	8		1	2	4	8
099	-1.31%	-0.65%	0.00%	0.00%	***101***	-6.55%	-1.71%	-0.57%	-0.57%
124	-3.21%	-0.32%	0.00%	0.00%	***102***	0.00%	0.00%	0.00%	0.00%
126	-2.53%	-1.01%	-0.51%	0.00%	***103***	-0.53%	-1.33%	-0.27%	-0.27%
129	-9.63%	-2.75%	0.00%	0.00%	***104***	0.00%	0.00%	0.00%	0.00%
130	-5.91%	-3.64%	-2.27%	0.00%	***107***	-5.22%	-3.87%	-2.02%	0.00%
132	-4.59%	-2.43%	-0.81%	0.00%	***110***	-37.89%	-18.42%	-1.05%	-0.26%
134	-9.01%	-2.15%	-1.72%	0.00%	***125***	-0.42%	-0.21%	0.00%	0.00%
147	-21.43%	-16.67%	-3.17%	-1.19%	***141***	-8.70%	-1.58%	-0.79%	-0.40%
HM	-6.89%	-3.48%	-1.05%	-0.15%	***145***	-1.63%	-1.31%	-0.33%	-0.33%
					146	-6.72%	-3.23%	-0.25%	-0.00%
					HM	-5.87%	-2.93%	-.052%	-0.18%
					HM all	-6.32%	-3.17%	-0.76%	-0.17%

Table 3.6: *Performance of the first distributed mechanism over the centralized mechanism that uses a 265-entry MDPT/MDST.*

ity as our goal is to demonstrate that a relatively straightforward implementation of synchronization/speculation can offer most of the performance benefits of oracle speculation (that this mechanism offers most of the potential performance benefits is shown in Section 3.8.7).

The results of the experiments with the second distributed mechanism suggest that while it performs worse than the other two mechanisms we studied, it still offers performance that is for most programs close to that possible with the centralized mechanism. Provided that this mechanism seems to be the most straightforward and less costly to implement, it may still be a good choice.

3.8.7 Comparison of Speculation/Synchronization Mechanism

Finally, we summarize our performance analysis by reporting, in Table 3.8 the relative over naive speculation performance of the following mechanisms: (1) centralized 256-entry combined MDPT/MDST ("CENT" columns), (2) distributed 256-entry combined MDPT/MDST with 2 stores per load ("D1" columns), and (4) the last distributed mechanism we studied ("D2" columns) which utilizes a level of indirection to represent dependence sets. Focusing on the harmonic means we can observe that the centralized mechanism performs best and offers performance very close to that possible with oracle speculation. The first distributed implementations performs slightly worse, but the difference is relatively small. Finally, the second distributed implementation still offers most of the performance improvements of oracle speculation. However, the performance difference is rather large. However, this last mechanism seems to have the advantage of being the most simple to implement.

	Slowdown		Slowdown
099	0.00%	***101***	-2.85%
124	-1.60%	***102***	0.00%
126	-1.01%	***103***	-2.92%
129	-7.80%	***104***	0.00%
130	-4.09%	***107***	-5.89%
132	-5.68%	***110***	-23.95%
134	-7.30%	***125***	-0.21%
147	-16.67%	***141***	-5.53%
HM int	-5.29%	***145***	1.31%
		146	-6.47%
		HM fp	-4.53%
		HM all	-4.87%

Table 3.7: *Performance of the second distributed mechanism over the centralized mechanism that uses a 265-entry MDPT/MDST.*

3.9 Evaluation - Centralized, Continuous -Window Processor Model

In this section, we study various methods of extracting load/store parallelism and their interaction with memory dependence speculation under a centralized, continuous instruction window execution model. Specifically: (1) we demonstrate that higher performance is possible if we could extract load/store parallelism and that the performance improvements are higher when the instruction window is larger (Section 3.9.1). (2) We demonstrate that naive memory dependence speculation can be used to attain some of the performance benefits possible. However, we also demonstrate that the net penalty of mispeculation is significant (Section 3.9.2). (3) We consider using an address-based scheduler to extract this parallelism and show that higher performance is possible than when naive memory dependence speculation is used (Section 3.9.3). In an address-based scheduler, loads and stores post their addresses as soon as possible, and loads are allowed to inspect the addresses of preceding stores before obtaining a memory value. (4) We show that performance drops rapidly when inspecting preceding store addresses increases load execution latency (i.e., when going through the address-based scheduler increases load latency — Section 3.9.3). (5) We demonstrate that an organization where memory dependence prediction is used to schedule load/store execution —instead of using an address-based scheduler— offers performance similar to that possible had we had perfect in-advance knowledge of all memory dependences (Section 3.9.4).

We note that the various load/store execution models we consider in this section are derived from meaningful combinations of the following parameters: (1) whether loads are allowed to inspect preceding store addresses before obtaining a value from memory, (2) whether stores wait for both data and base registers to become available before posting their addresses for loads to inspect, (3) whether loads with ambiguous dependences can issue (i.e., whether memory dependence speculation is used).

	Policy/Mechanism					Policy/Mechanism			
	Oracle	CENT	D1	D2		Oracle	CENT	D1	D2
099	8.11	3.38	3.37	3.37	***101***	24.56	23.17	22.45	19.65
124	10.64	10.64	10.64	8.86	***102***	2.67	2.13	2.13	2.13
126	13.33	10.00	9.44	8.89	***103***	4.35	2.44	2.17	-0.54
129	22.90	21.78	21.78	12.29	***104***	0.00	0.00	0.00	0.00
130	69.23	69.23	65.38	62.31	***107***	6.80	6.80	4.67	0.54
132	20.45	20.13	19.15	13.31	***110***	62.39	62.39	60.68	23.50
134	27.56	25.94	23.78	16.76	***125***	1.91	0.63	0.63	0.42
147	77.92	63.63	58.44	36.36	***141***	27.27	21.05	20.09	14.35
HM int	31.21	28.04	22.49	20.24	***145***	29.20	22.40	22.00	24.00
					146	14.68	13.55%	13.27	6.21
					HM fp	17.35	15.43	10.96	9.01
					HM all	23.50	21.03	16.08	14.00

Table 3.8: *Comparison of four speculation policies/mechanism: (1) oracle, (2) "CENT" centralized 256-entry MDPT/MDST (Section 3.8.5), (3) "D1" first distributed mechanism of Section 3.8.6, 4 stores per load, and (4) "D2" second distributed mechanism of Section 3.8.6 (level of indirection). Reported are speedups (%) over naive speculation.*

3.9.1 Performance Potential of Load/Store Parallelism

An initial consideration with the techniques we proposed is whether exploiting load-store parallelism can yield significant performance improvements. The reason is that in order to determine or predict memory dependences we need additional functionality: (1) To determine memory dependences we need a mechanism where loads and stores can post their addresses and execute accordingly to the dependences detected, that is, we need an address-based scheduler. (2) To predict memory dependences we need a memory dependence predictor and also a synchronization mechanism. For this reason an important consideration is whether this additional functionality is justified. Accordingly, we motivate the importance of exploiting load-store parallelism by comparing a model of a typical dynamically-scheduled ILP processor that does not attempt to determine and exploit load-store parallelism with one that is identical except in that it includes an oracle load-store disambiguation mechanism. Under this model, execution proceeds as follows: After an instruction is fetched, it is decoded and entered into the instruction window where its register dependences and availability are determined. If the instruction is a store, an entry is also allocated in a store buffer to support memory renaming and speculative execution. All instructions except loads can execute (i.e., issue) as soon as their register inputs become available. Stores *wait* for *both* data and address calculation operands before issuing. Loads wait in addition for all preceding stores to issue. As a result, loads may execute out-of-order only with respect to other loads and non-store instructions. The second configuration includes an oracle disambiguation mechanism that identifies load-store dependences as soon as instructions are entered into the instruction window. In this configuration, loads may execute as soon as their register and memory dependences (RAW) are satisfied. Since an oracle disambiguator is used, a load may execute out-of-order with respect to stores and does not need to wait for all preceding stores to calculate their addresses or write their data.

Figure 3.1 reports the performance improvements possible when perfect memory dependence information is available. We consider two base configurations, one with a 64-entry instruction window and one with a 128-entry window. For all programs, exploiting load/store parallelism has the potential for significant performance improvements. Furthermore, we can observe that when loads wait for all preceding store (“NO” bars) increasing the window size from 64 to 128 results in very small improvements. However, when the oracle disambiguator is used, performance increases sharply. This observation suggests that the ability to extract load/store parallelism becomes increasingly important performance wise as the instruction window increases.

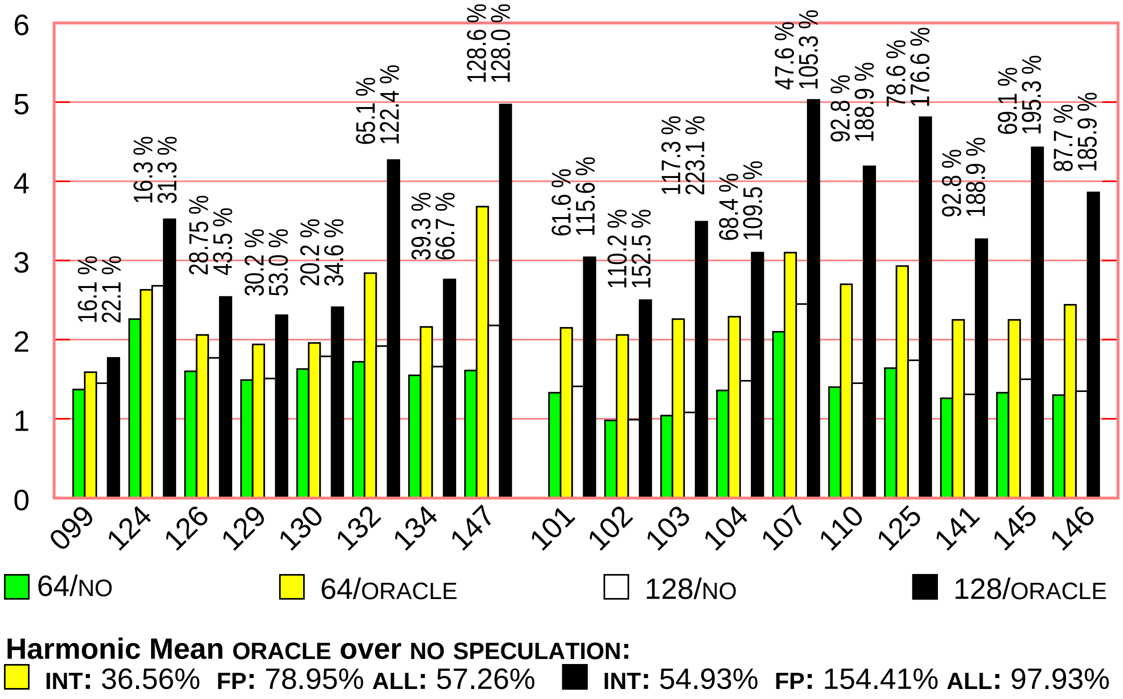


Figure 3.1: *Performance (as IPC) with and without exploiting load/store parallelism. Notation used is “instruction window size”/”load/store execution model”. Speedups of ORACLE speculation over NO speculation are given on top of each bar.*

When loads are forced to wait for all preceding stores to execute it is false dependences that limit performance. The fraction of loads that are delayed as the result of false dependences along with the average false dependence resolution latency are given in Table 3.9. We report false dependences as a fraction over all committed loads. We account for false dependences once per executed load and the time the load has calculated its address and could otherwise access memory. If the load is forced to wait because a preceding store has yet to access memory, we check to see if a true dependence with a preceding yet un-executed store exists. If no true dependence exists, we include this load in our false dependence ratio (this is done only for loads on the correct control path). We define, *false dependence resolution latency* to be the time, in cycles, a load that could otherwise access memory is stalled, waiting for all its ambiguous memory dependences to get resolved (i.e., all preceding stores have executed). We can observe that the execution of many loads and in some cases of most loads, is delayed due to false dependences and often for many cycles.

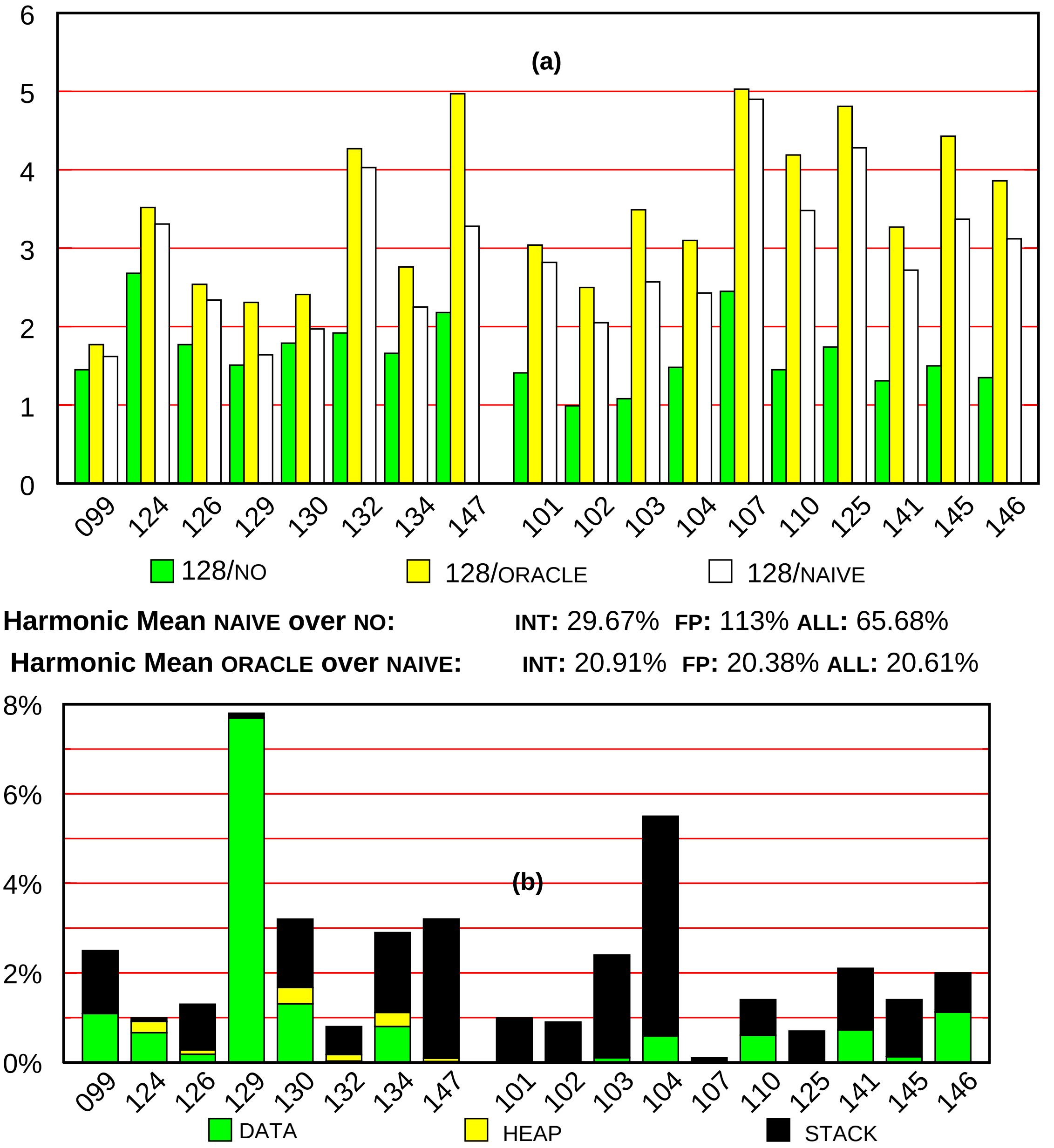
	False Dependences%	Resolution Latency
099	26.4%	13.7
124	59.9%	14.8
126	39.0%	47.3
129	70.3%	18.5
130	44.2%	39.1
132	70.3%	22.9
134	59.8%	39.1
147	67.2%	54.5
101	61.2%	36.3
102	91.0%	5.4
103	79.6%	91.2
104	85.2%	9.7
107	45.4%	26.6
110	45.4%	26.6
125	77.0%	55.6
141	77.5%	78.7
145	88.7%	51.4
146	83.6%	9.7

Table 3.9: *Fraction of loads with false dependences and average false dependence resolution latency (cycles) for the 128-entry instruction window processor.*

3.9.2 Performance with Naive Memory Dependence Speculation

As we have seen, extracting load/store parallelism can result in significant performance improvements. In this section we measure what fraction of these performance improvements naive memory dependence speculation can offer. For this purpose, we assume the same processor model assumed in the previous section but we allow loads to speculatively access memory as soon as their address operands become available. All speculative load accesses are recorded in a separate structure, so that preceding in the program order stores can detect whether a true memory dependence was violated by a speculatively issued load. Figure 3.1, part (a) reports performance (as IPC) for the 128-entry processor model when, from left to right, no speculation is used, when oracle dependence information is available and when naive memory dependence speculation is used. We can observe, that for all programs naive memory dependence speculation results in higher performance compared to no speculation. However, the performance difference between naive memory dependence speculation and the oracle mechanism is significant, supporting our claim that the net penalty of mispeculation can become significant. Memory dependence mispeculations are at fault. The frequency of memory dependence mispeculations is shown in part (b) of Figure 3.1. We measure mispeculation frequency as a percentage over all committed loads. A breakdown in terms of the address-space through which the mispeculation occurs is also shown. We can observe, that though loads cannot inspect preceding store addresses, mispeculations are rare. Nevertheless, the impact mispeculations have on performance is large.

In this context, the memory dependence speculation/synchronization methods we proposed could be used to reduce the net performance penalty due to memory dependence mispeculations. However, before we consider this possibility (which we do in Section 3.9.4) we first investigate using an address-based scheduler to extract load/store parallelism and its interaction with memory dependence speculation.



Harmonic Mean NAIVE over NO: **INT:** 29.67% **FP:** 113% **ALL:** 65.68%

Harmonic Mean ORACLE over NAIVE: **INT:** 20.91% **FP:** 20.38% **ALL:** 20.61%

Figure 3.1: *Naive memory dependence speculation. (a) Performance results (IPC). (b) Memory dependence misspeculation frequency.*

3.9.3 Using Address-Based Scheduling to Extract Load/Store Parallelism

We have seen that using oracle memory dependence information to schedule load/store execution has the potential for significant performance improvements. In this section we consider using address-based dependence information to exploit load/store parallelism. In particular we consider an organization where an address-based scheduler is used to compare the addresses of loads and stores and to guide load execution. We confirm that even in this context, memory dependence speculation offers superior performance compared to not speculating loads. However, we also demonstrate that when having to inspect addresses increases load execution latency, performance drops compared to the organization where oracle dependence information is available in advance (which we evaluated in the preceding section).

In the processor models we assume, stores and loads are allowed to post their addresses for disambiguation purposes as soon as possible. That is, stores *do not wait* for their data before calculating an address. Furthermore, loads are allowed to inspect preceding store addresses before accessing memory. If a true dependence is found the load always wait. When naive memory dependence speculation is used, a misspeculation is signaled only when: (1) a load has read a value from memory, (2) the value has been propagated to other instructions, and (3) the value is different than the one written by the preceding store that signals the mispeculation. As we noted earlier, under this processor model mispeculations are virtually non-existent. There are three reasons why this is so: (1) loads get either delayed because they can detect that a true dependence exists with a preceding store, (2) loads with unresolved dependences that correspond to true dependences are allowed to access memory but before they have a chance of propagating the value read from memory they receive a value from a preceding store, or (3) loads are delayed because preceding stores consume resources to have their addresses calculated and posted for disambiguation purposes.

Figure 3.2 reports how performance varies when naive memory dependence speculation is used compared to the same configuration that performs no speculation of memory dependences. For these experiments we use an 128-entry window processor model. We also measure how performance varies in terms of the time it takes for loads and stores to go through the address-based scheduler. We vary this delay from 0 to up to 2 cycles. In the calculation of the relative performance with naive speculation of part (a) of the figure, we should note that the base configuration is different for each bar. The absolute performance (i.e., the IPC) of the corresponding base configuration is reported in part (b). It can be seen that, for most programs, naive memory dependence speculation is still a win. Performance differences are not as drastic as they were for the model where loads could not inspect preceding store addresses, yet they are still significant. More importantly the performance difference between no speculation and memory dependence speculation increases as the latency through the load/store scheduler also increases. For some programs, naive memory dependence speculation results in performance degradation. These programs are 147.vortex for all scheduling latencies and 145.fpppp when the scheduling latency is 0. It is not mispeculations that cause this degradation. This phenomenon can be attributed to loads with ambiguous dependences that get to access memory speculatively only to receive a new value from a preceding store before they had a chance to propagate the value they read from memory. These loads consume memory resources that could otherwise be used more productively. This phenomenon supports our earlier claim that there is an opportunity cost associated with erroneous speculation.

While including an address-based scheduler does help in exploiting some of the load/store parallelism, a load may still be delayed even when naive memory dependence speculation is used. The reason is that in a real implementation for preceding stores to calculate their addresses and post them for scheduling purposes, they must consume resources. These resources include issue bandwidth, address calculation adders and load/store scheduler ports. The same applies to loads that should wait for preceding stores. If perfect knowledge of dependences was available in advance, stores would consume resources only when both their data and address calculation operands become available. For this reason, we next compare the absolute performance of the processor models that use an address-based scheduler to that of the processor model that utilizes oracle dependence information to schedule load execution (Section 3.9.1).

Figure 3.3 reports relative performance compared to the configuration that uses no speculation but utilizes an address-based scheduler with 0 cycle latency (the IPC of this configuration was reported in Figure 3.2, part (b), 0-CYCLE configuration). From left to right, the four bars per program report performance with: (1) oracle disambiguation and no address-based scheduler (Section 3.9.1), (2) through (4) naive memory dependence speculation and address-based scheduler with a latency of 0, 1 and 2 cycles respectively (which we

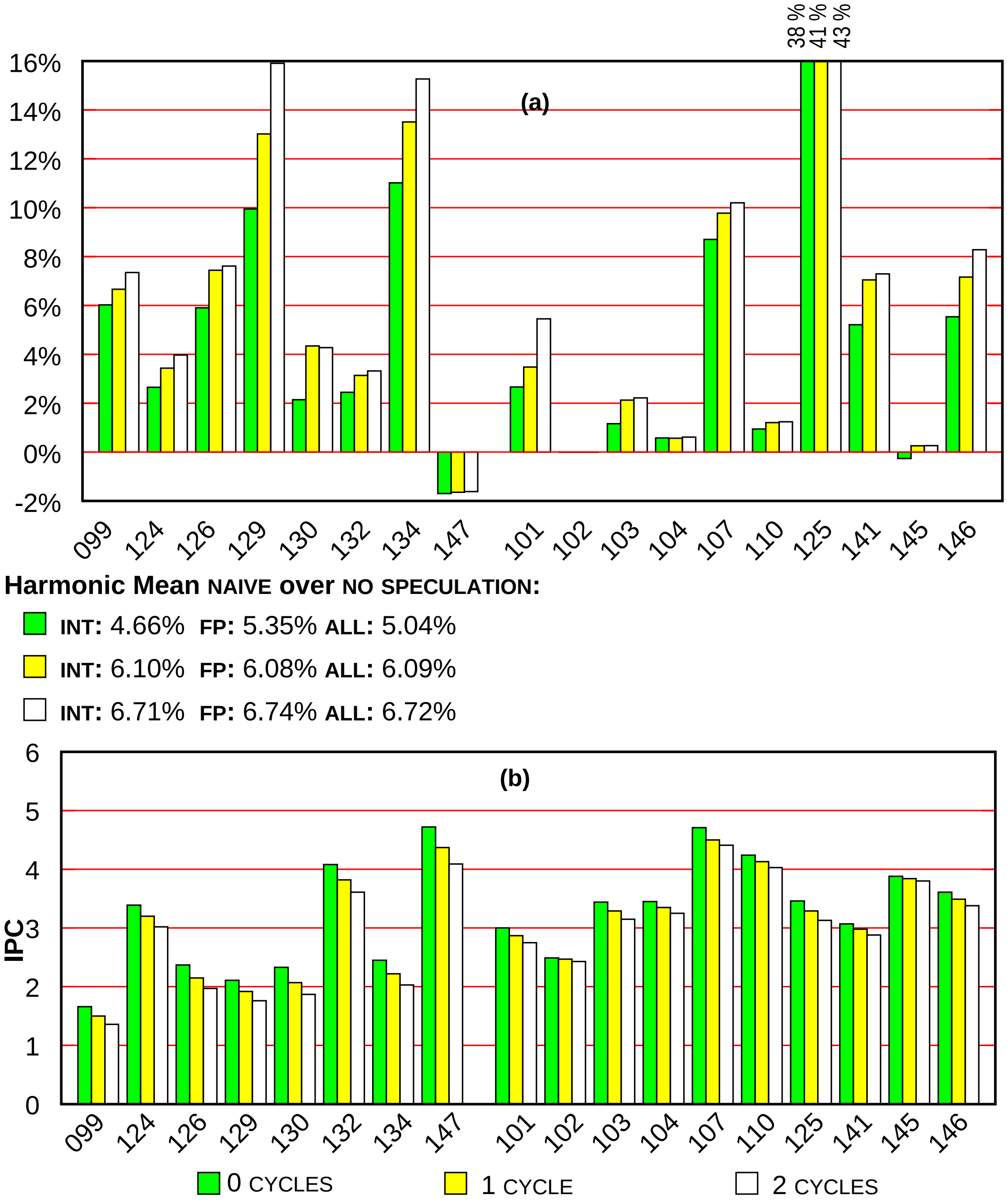
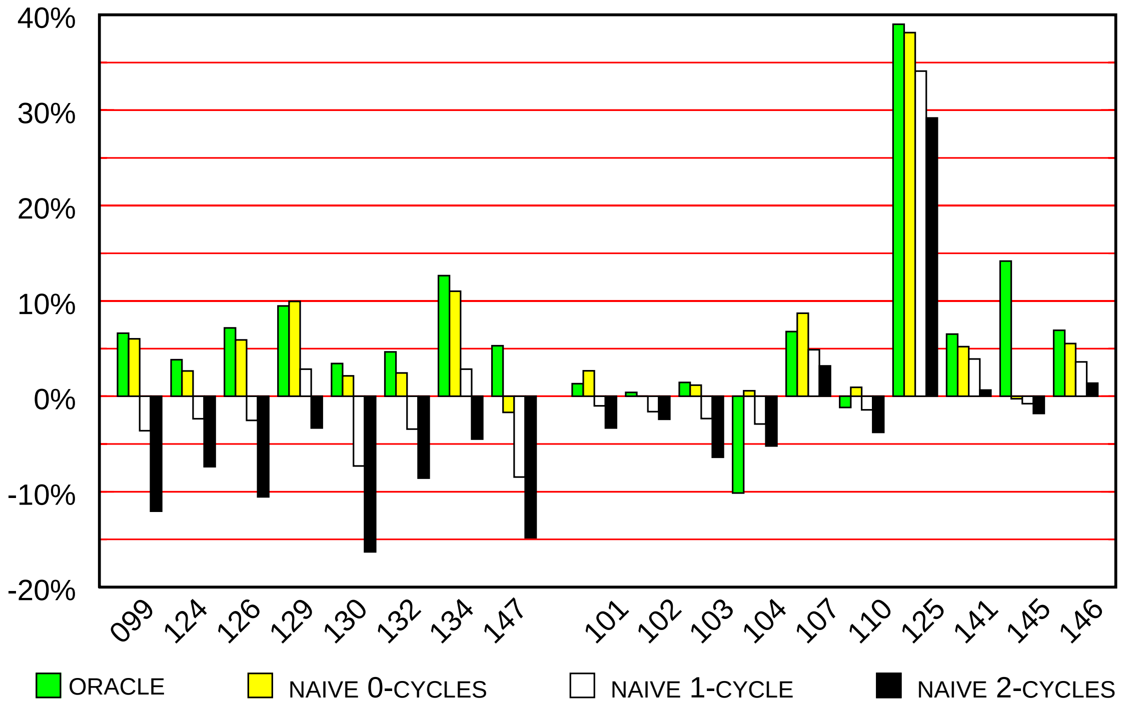


Figure 3.2: (a) Relative performance of naive memory dependence speculation as a function of the address-based scheduler latency. Performance variation is reported with respect to the same processor model that does not use memory dependence speculation. Base performance (IPC) is shown in part (b).

evaluated earlier in this section). We can observe that, with few exceptions, the 0-cycle address-based scheduler that uses naive speculation and the oracle mechanism perform equally well. Interestingly, the oracle configuration performance significantly better for 147.vortex and 145.fpppp supporting our earlier claim about resource contention and the opportunity cost associated with erroneous speculation. We can also



Harmonic Mean NAIVE over NO SPECULATION 0-CYCLE:

🟩 **INT:** 6.64% **FP:** 6.51% **ALL:** 6.57% □ **INT:** -2.75% **FP:** 3.63% **ALL:** 0.79%

🟨 **INT:** 4.81% **FP:** 6.26% **ALL:** 5.61% ■ **INT:** -9.68% **FP:** 1.13% **ALL:** -3.68%

Figure 3.3: *Comparing oracle disambiguation and address-based scheduling plus naive memory dependence speculation.*

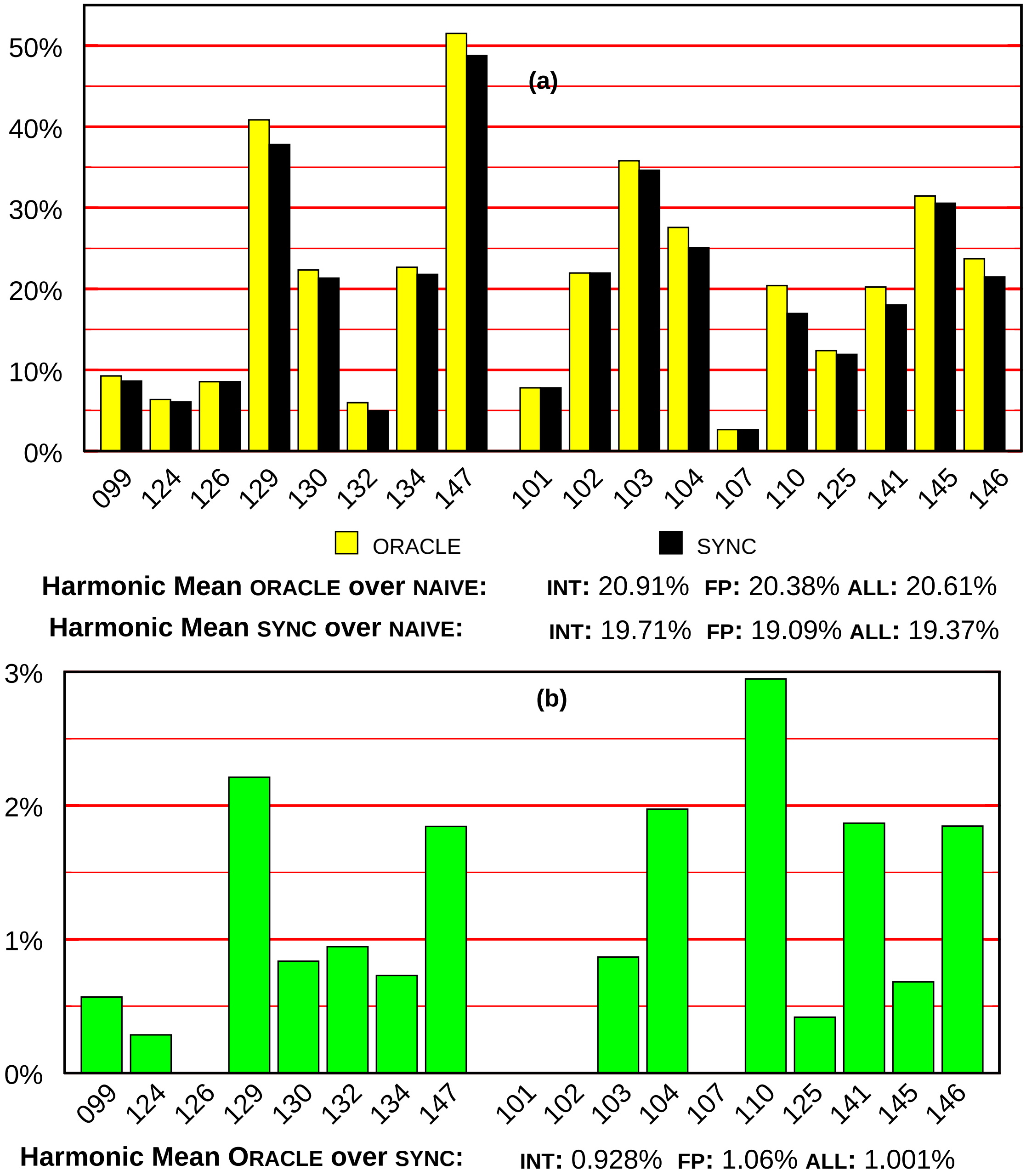
observe that once it takes 1 or more cycles to go through address-based disambiguation the oracle configuration has a clear advantage. The only exception is 104.hydro2d where the oracle configuration does significantly worse. This result may come as a surprise, however it is an artifact of our euphemistic use of the term "oracle". In the oracle model we assume, a store is allowed to issue only after both its data and address operands become available. As a result, dependent loads always observe the latency associated with store address calculation, which in this case is 1 cycle to fetch register operands and 1 cycle to do the addition. Under these conditions, dependent loads can access the store value only after 3 cycles the store has issued. When the address-based scheduler is in place, a store may calculate its address long before its data is available and dependent loads can access the store's value immediately. In an actual implementation, it may be possible to overlap store address calculation and store data reception without using an address-based scheduler (e.g., [34, 84]).

3.9.4 Speculation/Synchronization

In this section we consider using an implementation of our speculation/synchronization approach to improve accuracy over naive memory dependence speculation. As we have observed in the previous chapter, in a continuous window processor that utilizes an address-based load/store scheduler along with naive memory dependence speculation, mispeculations are virtually non-existent. In such an environment there is no need for our speculation/synchronization method. However, as we observed in Section 3.9.2, if an address-based scheduler is not present then the net penalty of misspeculation resulting from naive memory dependence speculation is significant. Moreover, and as we have seen in the previous section, the performance potential of a method such a speculation/synchronization (oracle configuration of Figure 3.3) is often close or exceeds the performance possible with an address-based scheduler with 0 cycle latency. Accordingly, in this section we restrict our attention to a configuration that does not use an address-based load/store scheduler. In the rest of this section we first provide the details of the speculation/synchronization mechanism we simulated and then proceed to evaluate its performance.

The speculation/synchronization mechanism we used in these experiments comprises a 4K, 2-way set associative MDPT in which separate entries are allocated for stores and loads. Dependences are represented using synonyms, i.e., a level of indirection. The algorithm used to generate synonyms is the one described in Section 3.6.6 with the only difference being that no unit distance is associated with each synonym (units distances were used in the Multiscalar execution model). It is implied that a load is synchronized with the last preceding store instance that has been assigned the same synonym. MDPT entries are allocated —if they don't exist already— upon the detection of a memory dependence violation for both offending instructions. No confidence mechanism is associated with each MDPT entry; once an entry is allocated, synchronization is always enforced. However, we flush the MDPT every million cycles to reduce the frequency of false dependences (this method was proposed in [17]). The functionality of the MDST is incorporated into the register-scheduler, which we assume to follow the RUU model [81]. This is done as follows: an additional register identifier is introduced per RUU entry to allow the introduction of speculative dependences for the purposes of speculation/synchronization. Stores that have dependences predicted use that register identifier to mark themselves as producers of the MDPT supplied synonym. Loads that have dependences predicted by the MDPT, use that register identifier to mark themselves as consumers of the MDPT supplied synonym. Synchronization is achieved by: (1) making loads wait for the immediately preceding store (if there is any) that is marked as the producer of the same synonym, and (2) having stores broadcast their synonym once they issue, releasing any waiting loads. A waiting load is free to issue one cycle after the store it speculatively depends upon issues.

Figure 3.4, part (a) reports performance results relative to naive memory dependence speculation (Section 3.9.2). As it can be seen our speculation/synchronization mechanism offers most of the performance improvements that are possible had we had perfect in advance knowledge of all memory dependences (oracle). This is more clearly shown in part (b) of the same figure, where we report the relative over our speculation/synchronization mechanism performance of oracle speculation. On the average, the performance obtained by the use of our mechanism is within 1.001% of that possible with the oracle mechanism. For four programs (126.gcc, 101.tomcatv, 102.swim and 107.mgrid) our mechanism results in performance that is virtually identical to that possible with oracle speculation. For the rest of the programs the differences are relatively minor (3% in the worst case) when compared to the performance improvements obtained over naive speculation. To help in interpreting these differences we also present the misspeculation rates exhibited when our mechanism is in place. These results are shown in Table 3.10 (reported is the number of misspeculations over all committed loads). As it can be seen, misspeculations are virtually non-existent. This observation suggests that for the most part the performance differences compared to oracle speculation are the result of either (1) false dependences, or (2) of failing to identify the appropriate store instance with which a load has to synchronize with. False dependences are experienced when our mechanism incorrectly predicts that a load should wait although no store is actually going to write to the same memory location. In some cases and even when prediction correctly indicates that a dependence exists, our mechanism may fail to properly identify the appropriate store instance. This is the case for memory dependences that exhibit non-unit instance distances (e.g., a[i] = a[i - 2]). In such cases, a load is delayed unnecessarily as it is forced to wait for the very last preceding store instance that has been assigned the same synonym.



Harmonic Mean ORACLE over NAIVE: **INT:** 20.91% **FP:** 20.38% **ALL:** 20.61%

Harmonic Mean SYNC over NAIVE: **INT:** 19.71% **FP:** 19.09% **ALL:** 19.37%

Harmonic Mean ORACLE over SYNC: **INT:** 0.928% **FP:** 1.06% **ALL:** 1.001%

Figure 3.4: *Performance of an implementation of speculation/synchronization.(a) Performance improvements over naive speculation. (b) Relative performance of oracle speculation over our speculation/ synchronization.*

The results of this section suggest that our speculation/synchronization method can offer performance that is very close to that possible had we had perfect, in advance knowledge of all memory dependences. However, further investigation is required to determine how selective speculation and the store barrier policy would perform under this processor model.

	Mispeculation Rate%			Mispeculation Rate%	
	NAIVE	SYNC		NAIVE	SYNC
099	2.5%	0.0301%	***101***	1.0%	0.0001%
124	1.0%	0.0030%	***102***	0.9%	0.0017%
126	1.3%	0.0028%	***103***	2.4%	0.0741%
129	7.8%	0.0034%	***104***	5.5%	0.0740%
130	3.2%	0.0035%	***107***	0.1%	0.0019%
132	0.8%	0.0090%	***110***	1.4%	0.0039%
134	2.9%	0.0029%	***125***	0.7%	0.0009%
147	3.2%	0.0286%	***141***	2.1%	0.0148%
			145	1.4%	0.0096%
			146	2.0%	0.0034%

Table 3.10: *Memory dependence mispeculation rate with our speculation/synchronization mechanism (“SYNC” columns) and with naive speculation (“NAIVE” columns).*

3.10 Chapter Summary

In this chapter we studied how naive memory dependence speculation will interact with future, wide-window dynamically scheduled processors. Under naive memory dependence speculation a load is always allowed to access memory when ambiguous dependences with preceding stores exist. We have shown that as instruction windows get larger, the net penalty of dependence mispeculations can become significant. We used this result to motivate techniques that aim at improving the accuracy of memory dependence speculation. Moreover, we argued that the goals of these techniques should not only be (1) to reduce mispeculations, but also (2) to delay loads *only as long as it is necessary* to avoid mispeculation. Accordingly, we proposed memory dependence speculation and synchronization, a technique that uses history-based memory dependence prediction to enforce synchronization of those load-store pairs that are otherwise mispeculated.

We have studied various memory dependence speculation policies using timing simulation of two processor models. The first utilized a distributed, split-window approach to establish a wide window (Multiscalar). The second used a centralized window (typical, modern superscalar).

For the distributed, split-window processor model our findings were:

1. Naive memory dependence speculation offers significant performance improvements over no speculation. Speedups of roughly 50% and 180% were observed for the integer and floating point programs and for an 8-unit configuration.
2. Perfect knowledge of all memory dependences could potentially lead to a 31% (integer) and 17% (floating-point) performance improvement over naive speculation (for the 8-stage configuration).
3. Using store address information to detect memory dependences and avoid mispeculations could help if it was made available. The potential performance improvements were 8.8% (integer) and 3.0%. However, the net penalty of dependence mispeculations was still high.
4. Selective speculation, i.e., using prediction to avoid speculating on those loads that would otherwise cause a mispeculation, is not a robust technique. In fact, for most programs it performed worse

than naive speculation.

5. We evaluated three alternative implementations of our dependence speculation and synchronization technique and found that they could offer most of the performance benefits possible with perfect knowledge of all memory dependences. The speedups over naive speculation were: (1) 28% and 15% (integer and floating-point), (2) 22% and 11% and (3) 20% and 9%, for the three mechanisms respectively.

For the centralized, continuous window processor model that employed fetch and execution mechanisms of equal bandwidth and a program order priority scheduler, our findings were:

1. Naive memory dependence speculation can improve performance over not speculating memory dependences independently on whether loads can inspect preceding store addresses.
2. Exposing store addresses can eliminate virtually all memory dependence mispeculations. Performance is close to that possible with perfect knowledge of all memory dependences if it takes no time to inspect store addresses before performing a load. However, performance degrades when load latency is increased.
3. The memory dependence speculation and synchronization technique we proposed could be useful in a design that does not use an address-based scheduler to extract load/store parallelism. In this environment, memory dependence mispeculations are frequent and the performance improvements possible over naive speculation are rather large. Had we had perfect knowledge of all memory dependences, performance would improve by 20.9% (integer) and 20.4% (floating-point). An implementation of our speculation/synchronization method offers most of this potential, as it resulted in performance improvements of 19.7% (integer) and 19.1% (floating-point). The potential advantage of this design is that it may incorporate the load/store scheduling functionality over the existing register dependence scheduler.

While most modern processors utilize a centralized, instruction window it is not unlikely that future processors will have to resort to distributed, split-window organizations. As many argue, the reason is that the existing, centralized methods may not scale well for future technologies and larger instruction windows (e.g., [66]). An example were techniques similar to those we presented have been put to use already exists: the Alpha 21264 processor utilizes both a split-instruction window approach and selective memory dependence speculation to reduce the net penalty of dependence mispeculations [45]. As our results indicate, techniques to predict and synchronize memory dependences can greatly improve the performance of future split-window processors, allowing them to tolerate slower memory devices by aggressively moving loads up in time while avoiding the negative effects of memory dependence mispeculations. Moreover, speculation/synchronization may be used as a potentially lower complexity/faster clock cycle alternative to using an address-based load/store scheduler for exploiting load/store parallelism.

Chapter 4

Speculative Memory Cloaking and Bypassing

In the previous chapter, we were concerned with techniques to send load requests to memory as early as possible. Our goal was to exploit instruction level parallelism, tolerating slower memory devices by overlapping memory latency with other useful work. In this chapter we investigate a different, yet orthogonal approach in our attempt to meet the latency and bandwidth requirements of high performance processing. We observe that memory can be treated differently depending on whether it is viewed from the perspective of single instruction or from the perspective of the program as a whole. From the perspective of a single instruction memory appears simply as storage. However, when viewed from the perspective of the program as a whole, memory can be treated as an interface used to synthesize a desired, possibly elaborate action. This observation suggests another direction of improving performance: identifying what the intended action is (i.e., what is memory being used for), and if possible, developing methods to perform the same action faster.

In this chapter we identify two common uses of memory. The first is *inter-operation communication* where a store writes a value to memory so that loads may later read it. The second is *data-sharing* where memory is used to hold data which is read repeatedly. In both cases, we argue that the traditional, address-based memory interface introduces both overheads — in the form of address-calculation and disambiguation — and inconveniences in managing the storage structures used to implement memory. However, we also observe that the traditional address-based way of expressing both actions is not the only possible way and in particular, we argue that an explicit specification of either action has advantages. Specifically, in an explicit representation of inter-operation communication the identity of the producing store is known by the consuming loads and vice versa. As a result, values can flow directly from stores to loads without address calculation and disambiguation being necessary. Similarly, an explicit representation of data-sharing loads can identify an earlier load that accesses the same data. This information can be used to place the repeatedly read data into a separate name space where it can be accessed directly (i.e., by identifying the earlier load that accessed it), without incurring the overheads associated with the traditional address-based memory.

Motivated by the aforementioned observations, we focus on methods of converting the traditional, address-based specification of inter-operation communication and of data-sharing into an explicit form.

Observing that inter-operation communication between a store and a load manifests as a RAW dependence and that repeated read accesses manifest as RAR dependences, we propose *speculative memory cloaking* (*cloaking* for short), a technique that dynamically, and in an architecturally transparent way converts the traditional implicit specification of the two aforementioned actions into an explicit, albeit speculative specification. In this technique, memory dependence prediction is used to transparently create a new, speculative name space free of aliases. Via this new name space, (1) stores may pass values directly to the loads that need it, and (2) loads may access data by identifying a preceding load that also read it. Since we make use of memory dependence prediction, the proposed technique is speculative and so are the values obtained through its use. Accordingly, value verification through the traditional memory name space is necessary. However, as this verification can take place while the speculative values are used for further processing, the observed memory latency is reduced; instructions that would otherwise wait for the traditional memory hierarchy to provide a value may now execute earlier, possibly even before the memory access is initiated.

Aiming at further reducing memory value access latency, we observe that loads and stores typically do not change the data they read or write respectively. These instructions are really used as agents retrieving or placing memory data that other instructions produce or consume; the value written by a store is produced by another instruction, while the value read by a load is passed to other instructions. Motivated by this observation we propose *speculative memory bypassing* (bypassing for clarity), a conceptually straightforward extension to cloaking. This technique speculatively bypasses loads and stores, linking producing and consuming instructions directly. When inter-operation communication takes place, bypassing converts DEF-STORE-LOAD-USE chains into DEF-USE ones, while when data sharing occurs, this technique consolidates a series of LOAD_1-USE_1...LOAD_N-USE_N chains into a single LOAD_1-USE_1...USE_N producer-consumer graph. As with cloaking, bypassing requires verification of the speculatively communicated values.

The rest of this chapter is organized as follows. In Section 4.1 we discuss the two common uses of memory that concern us and review the implications a traditional address-based interface has on performing either of them. Here we describe the rationale for our proposed approach. In Section 4.2 we describe speculative memory cloaking and its requirements. In Section 4.3 we introduce speculative memory bypassing. In Sections 4.2 and 4.3 we focus on inter-operation communication. Having described how cloaking and bypassing works for inter-operation communication, we briefly explain how these techniques can be extended to support data sharing. We do so in Section 4.4. In Section 4.5 we comment on related work. A quantitative assessment of the proposed techniques is presented in Section 4.6. Finally, a summary of the chapter is given in Section 4.7.

4.1 Two Common Uses of Memory

From the perspective of a single instruction, memory can be viewed as a storage mechanism: an instruction may read data from memory or write data to it. A simple interface is provided for both reading and writing: instructions have to first calculate an address and then use it to access the corresponding memory location. Although this is a simple interface it is also quite powerful and when viewed from the perspective of the program as a while, memory can be used to synthesize elaborate actions. One common use of memory is inter-operation communication, where a memory location is used to pass a value from one instruction to another. This inter-operation communication is performed using a store and a load: the store binds the value to an address and the load has to access the value using the same address. Memory may also be used to hold data that is read repeatedly by the program (data sharing). In this case, every time the value is needed, a load instruction is used to bring it into a register so that other instructions can access it.

While the traditional memory address-based interface is simple and powerful, it is also general and as such, it may introduce unnecessary overheads or inconveniences when used to express the aforementioned actions. In the Sections 4.1.1 and 4.1.2, we discuss how the traditional memory interface impacts inter-operation communication and data sharing. In Section 4.1.3 we use this discussion to argue when an explicit specification of these actions has advantages.

4.1.1 Memory as an Inter-operation Communication Agent

As we have seen in Chapter 2 a large fraction of all executed loads read a value written by a preceding store. This observation suggests that memory is often used as an inter-operation communication agent passing values from stores to loads. Faster processing however, requires faster inter-operation communication (latency). This is especially important for future wide-issue instruction-level-parallel processors, where even a single cycle will correspond to many instruction issue opportunities. Furthermore, to support a higher-degree of instruction level parallel execution, the mechanisms used to implement inter-operation communication need to also support multiple simultaneous operations (bandwidth).

In the traditional, address-based memory interface, memory communication is expressed implicitly. The load or the store provide no indication of the communication that has to happen. To detect the communication and establish the communication link both the store and the load have to calculate their address and go through disambiguation. The latter action entails comparing the addresses of stores and loads taking program order into account; a store and a load communicate if (1) they access the same address, and (2) no intervening store (in the program order) accesses the same address. Both address calculation and disambiguation introduce overheads as the value may be available long before either action completes. An example is shown in Figure 4.1 where communication is to take place between the STORE and LOAD instructions of the code fragment shown in part (a). Part (b) shows a possible sequence events during execution. Initially the store is fetched, its address is calculated and later the store data becomes available. Later on, the load is encountered. At this point both instructions that need to communicate have been encountered and the value is available. Yet, communication is delayed until LOAD has calculated its address and has passed through disambiguation to establish the dependence with STORE. Depending on whether memory dependence speculation is used (see Chapter 3), accessing the memory value may be further delayed until it is established that no other intervening store accesses the same memory location. For example, LOAD may get delayed until STORE$_1$ also calculates its address and goes through disambiguation.

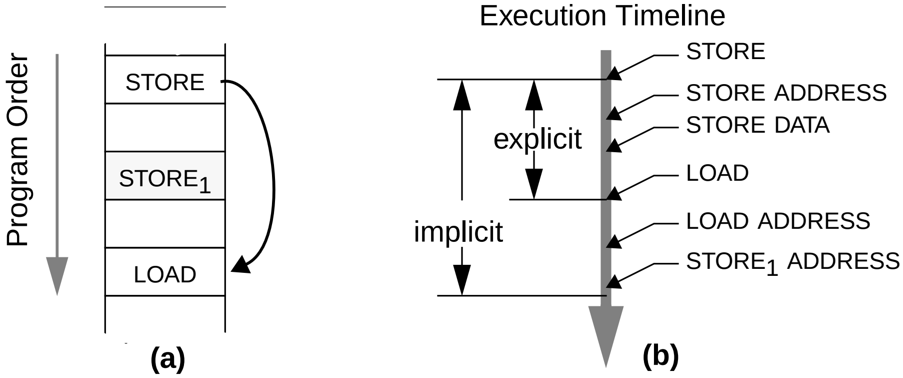


Figure 4.1: *An example of inter-operation memory communication. (a) Program segment with a store and a load that will communicate. (b) Timeline of execution. With an implicit specification, communication cam take place after address-calculation and disambiguation. With an explicit specification communication can take place as soon as the two instructions are encountered and the value becomes available.*

Even when the value becomes available after both the load and the store complete address-calculation and disambiguation, we will still observe the latency associated with accessing the value through the memory hierarchy, i.e., through the store buffer or the data cache. It is likely that the memory hierarchy latency will be higher for future wide-issue long instruction window processors. Here are a few reasons why this might be the case. Store buffers may be slower as these processors will have to establish and scan through a larger instruction window which will include more stores. Accessing the data cache may require more cycles as a result of two trends. The first is the working set of typical programs which traditionally increases with time. If caches have to increase in size to compensate for larger working sets, they may become relatively slower [98]. The second is implementation technology. Many argue that as feature sizes decrease, wire lengths will dominate [57] and this in turn may result in caches being slower. Finally, unless an increased number of memory ports is provided, the lack of additional cache access bandwidth may manifest as an increase in cache access latency.

4.1.2 Memory As A Value Place Holder

In Chapter 2 we have seen that many memory values are read multiple times. This is no surprise as after all the temporal locality exhibited by programs can give rise to this phenomenon. As we demonstrate later on however, locality is stronger when viewed in terms of the read-after-read dependences formed when two or more loads access the same memory location. That is, if at some point we observe two loads accessing a common memory location, chances are the same loads will soon access a common, but possibly *different* memory location. As with inter-operation communication, overheads are introduced by the traditional memory interface when accessing values repeatedly. Even though the value is not changed, the corresponding loads have to calculate an address and go through disambiguation. The latency associated with accessing the value from the memory hierarchy is also incurred and the discussion of the previous section applies in this case also. (While the mechanisms introduced in Chapter 3 can be used to avoid incurring the disambiguation latency before accessing memory all loads will still have to calculate an address and go through the memory hierarchy to obtain their value.)

4.1.3 Using Memory Dependence Prediction To Streamline Memory Accesses

The overheads introduced by the traditional memory interface when performing inter-operation communication and when data sharing occurs can be eliminated if we opt for an explicit representation of both actions. In an explicit representation of inter-operation communication the producing store and the consuming load both know that communication will take place and can locate each other. As a result, communication can take place as soon as the two instructions are encountered and the value becomes available. Since both instructions can locate each other, there is no need for address-calculation and disambiguation. Similarly, in an explicit representation of data sharing, all loads that access a common memory location have knowledge of this fact and can locate each other. As a result, loads may initiate their access by identifying the first load in program order that accessed the common memory location. If that load has completed its access, the value is available immediately without having to first calculate an address to locate it.

Both inter-operation communication and data-sharing give rise to dependences. Inter-operation communication gives rise to true (RAW) dependences and data-sharing gives rise to RAR dependences. An explicit representation of either action requires a representation of the corresponding dependences. We could attempt to determine and specify these memory dependences statically as for example, was done in dataflow machines [18, 91]. Even though this is an interesting option, we will not consider it further for two main reasons. First, a static representation would involve changing the program representation and would create a legacy issue for future processor generations. Second, a static approach may have difficulty in identifying that inter-operation communication or data-sharing takes place either because the dependences cannot be determined statically (i.e., they are ambiguous) or because they are transient (i.e., do not occur every time).

For these reasons, we opt for a dynamic approach in which the conversion is done dynamically while the program is running using architecturally invisible structures.

In the sections that follow we introduce *cloaking* where we utilize memory dependence prediction to dynamically annotate loads and stores with the information necessary to identify inter-operation communication and data-sharing, allowing them to establish direct communication links. We do so as follows: the first time a RAW or a RAR dependence is encountered we record the identities of the dependent instructions. We use this dynamically collected dependence information to create a new, speculative name space and to associate its names with the dependent instructions directly. This allows subsequent instances of the dependent loads and stores to derive the corresponding name based solely on their identity (e.g., PC) and to locate each other without having to perform address-calculation and go through disambiguation. As an additional benefit, the name space created via memory dependence prediction, being more compact, can be mapped onto a relatively small and for that, fast, storage structure. As we rely on speculative dependence information any values obtained through our technique have to be eventually verified via the traditional memory name space. This implies that the proposed technique does not eliminate the need to access a value via an address. Even so, when speculation is successful, the proposed technique may reduce the latency *observed* by those instructions that use the communicated value. This is possible when the value is available long before the address-based access completes through the traditional memory hierarchy.

While cloaking may reduce memory access latency, inter-operation communication still occurs via stores and loads and every access to shared data requires a separate load. Taking a closer look at inter-operation memory communication we can observe that the store and load instructions used do not change the values communicated (ignoring sign-extension and data-type conversions). They too are part of the memory interface acting as agents that write and read values from memory locations. The value written by a store is actually produced by another instruction and passed to the store via a register. The value read by a load is not consumed by the load itself, rather it is propagated to other instructions that use it for further processing. These observations suggest that further reduction in communication latency may be possible if we bypass the load and store instructions, linking the producing and consuming instructions directly. Similar observations apply to data-sharing, where multiple loads are used to propagate the same memory value to multiple consuming instructions. This observation suggests that further reduction in memory latency may be possible if we bypass all but the first load, linking all the consumers of the accessed memory value directly with the first load. To link producing and consuming instructions directly, bypassing loads and stores, we introduce *speculative memory bypassing.*

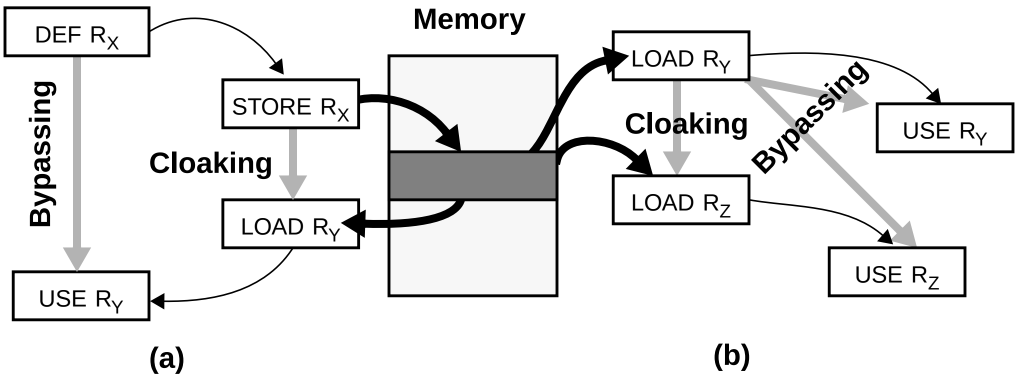


Figure 4.2: *Speculative Memory Cloaking and Bypassing. (a) Inter-operation communication: exploiting read-after-write dependences. (b) Data sharing: exploiting read-after-read dependences.*

Figure 4.2 illustrates the effect we aim to achieve with cloaking and bypassing. As shown, cloaking permits loads to obtain a speculative value by locating a preceding store or load that wrote or read it respectively. In the case of inter-operation memory communication, bypassing links directly the actual producer and consumer of a value. In the case of data sharing, bypassing links one of the loads with all the consumers of shared memory value.

The rest of this chapter is organized as follows. In Section 4.2 we describe cloaking and then in Section 4.3 we present bypassing. Initially, we describe both techniques in the context of inter-operation communication. With an understanding of what is involved in streamlining inter-operation communication via cloaking and bypassing, we describe how these techniques can be extended to also handle data sharing in Section 4.4.

It should be noted that the discussion that follows and the mechanisms presented are geared toward a continuous instruction window instruction-level parallel processor. However, cloaking and bypassing may be applicable in other environments where inter-operation communication and data-sharing through memory occurs (e.g., split-window processing models as Multiscalar or explicitly parallel processing models). Changes may be required in the schemes producers and consumers use to locate each other in those environments. For example, in Chapter 3 we saw how a different naming scheme was used in the Multiscalar architecture —non-continuous, split window — to identify instances of loads and stores (we used a PC, stage number pair). Nevertheless, the concepts that underline these techniques may still apply.

4.2 Speculative Memory Cloaking

The purpose of cloaking is to streamline memory communication by dynamically converting the implicit specification of communication into an explicit form. In cloaking, memory dependence prediction is used to identify loads and stores that are likely dependent. The high-degree of locality found in the dynamic dependence stream of the programs we studied (Chapter 2) suggests that such an approach may be successful. In cloaking, once a dependence is deemed predictable, the dependent load and store are explicitly linked via a new name, a *synonym* which uniquely identifies the dependence (e.g., the synonym can be the (load PC, store PC) pair). One may wonder how using a different name may help in streamlining the actual communication. After all, data addresses and synonyms are just names that the dependent instructions use to link to each other. The answer lies in the nature of the association between the name and the instructions that use it and in the information associated with the existence of the name itself. In contrast to an address, the synonym is intended to uniquely identify the dependent instruction pair. This allows the load and the store to derive the synonym based solely on their identity (PC) locating the appropriate value without having to first perform an address calculation and go through disambiguation. Furthermore, the mere association of a synonym with a load or a store is intended to indicate that the instruction is involved in inter-operation communication.

The process of cloaking is illustrated in Figure 4.3. As shown in part (a), detecting a load-store dependence results in an association among the load, the store and a function that can be used to derive preferably unique synonyms for future instances of the dependence. When a subsequent instance of the store instruction is brought into the instruction window and a dependence is predicted (part (b), action 1), this association results in the generation of a synonym and in the allocation of physical storage for this synonym (action 2). Storage for the synonym is preferably provided in the *Synonym File (SF)* which is a small, low latency/high bandwidth storage structure. The storage element is initially marked as empty, indicating that no value is yet available. When the store obtains its value, it also updates the synonym file entry marking it as full (action 3). Finally, when the store computes its address, the value is also written to the traditional memory system (action 4). When the appropriate instance of the load is encountered, the memory dependence prediction mechanism is probed. Provided that the dependence is predicted correctly, the association is used again to derive the synonym (part (c), action 5) and consequently, to locate the appropriate element in the synonym file (part (c), action 6). Instructions that use the load value may at this point execute speculatively using the value found in the synonym file (action 7). When the load's address becomes available, the memory system is accessed to read the actual value (action 8). The memory value is compared with the value obtained earlier via the cloaking mechanism. If the two values are the same, cloaking was successful and no further action is required. Otherwise, data value mispeculation occurs, and any instructions that used wrong data have to be re-executed. It should be noted that the above discussion covers one possible sequence of events. In practice, events may occur in a order different than the one just described. For example, the load may be encountered before the store writes a value in the synonym file. In any case, cloaking still provides the benefit of establishing a communication link early without requiring address-calculation and disambiguation.

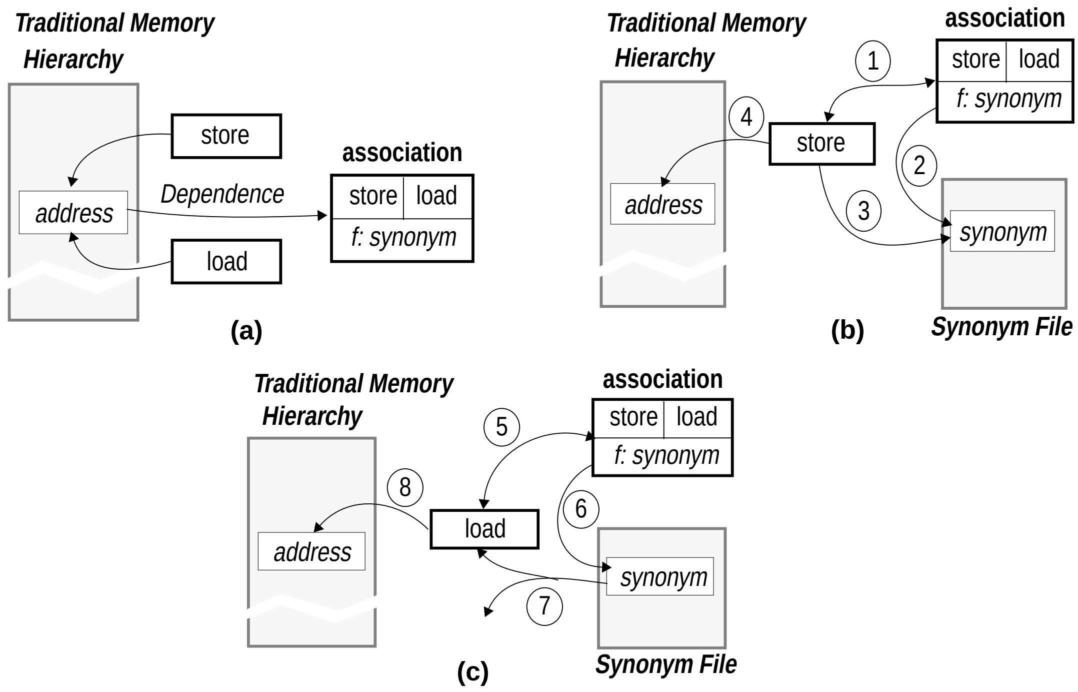


Figure 4.3: *Streamlining memory communication via cloaking: (a) Detecting a read-after-write dependence results in an association between the dependent load and store. (b) A subsequent instance of the store creates a synonym. (c) A subsequent instance of the load locates the synonym and uses its data speculatively.*

We use the term *memory cloaking* to signify that the original address-based specification of memory communication and the mechanisms used to implement memory itself are hidden when values are communicated via cloaking. With this technique, communication takes place through a dynamically created name space and *without* knowledge of the address used by the program. No association between the storage used by the synonym and the address is ever built. Furthermore, we clarify the technique as speculative since the use of speculative dependence information makes communication via cloaking speculative.

Speculative memory cloaking requires the following functionality: (1) predicting dependences, (2) creating synonyms, associating them with the dependent instructions and assigning storage for the communication, and (3) verifying the speculatively communicated values. In Sections 4.2.1 through 4.2.3 we discuss each of these requirements in detail. Finally, we present an implementation of cloaking in Section 4.2.4.

Before we proceed with our description we should note that in the discussion that follows we make the assumption that the value read by a load is always produced by a single store. However, since loads and stores may operate on different data types, this might not be always the case. While support for memory communication among loads and stores that operate on different data types might be possible [62] we do not consider such options here because as we have seen in Chapter 2, such dependences rarely occur in practice.

4.2.1 Detection and Prediction of Dependences

For cloaking to be successful, we have to be able to predict memory dependences. As the results of Chapter 2 indicate, programs behavior is such that history-based prediction could potentially be used to predict memory dependences with high accuracy. For history-based prediction to be possible we need: (1) a mechanism to detect RAW dependences, and (2) a mechanism to record information about detected RAW dependences and to use this information for prediction purposes. In the rest of this section we first discuss a mechanism that can be used to detect RAW memory dependences and then discuss a history-based RAW memory dependence predictor.

Since history-based predictors rely on information about past behavior to make their predictions, we first need a mechanism to detect store-load dependences. There are two considerations: (1) how dependences should be reported by the detection mechanism, and (2) what is the desired scope of this mechanism (i.e., how many loads and stores it should detect dependences for). Given that we have decided to associate predictions with the instructions themselves (a property that allows us to access values using instruction-based information), the detection mechanism should provide sufficient information to identify the two dependent instructions. For the purposes of this work we require that the dependence detection mechanism indicates dependences using the PCs of the dependent instructions by reporting dependences as (store PC, load PC) pairs. Moreover, for the purposes of cloaking it is desirable to predict dependences on as many loads and stores as possible (in contrast to the applications we presented in Chapter 3, where we only cared about those loads and stores that are mispeculated). As the analysis of Chapter 2 suggests, to capture a large fraction of the dynamic dependences we need to be able to detect dependences over several addresses (e.g., 256), preferably over regions that most likely exceed the instruction window used. This is possible if we maintain a record of recent stores (e.g., their PC) along with the memory address each touched. Dependences can be detected when loads access memory by inspecting these records to determine which was the last store, if any that updated the particular memory location. At this point the identities of both the load and the store are known. A relatively inexpensive and straightforward implementation is via a *Dependence Detection Table* (*DDT*) which is nothing more than a regular but very small cache that records the PC of the store that last touched each recorded address. Note that since the information collected by the detection mechanism is used only for prediction purposes, relatively long detection latencies and detection errors may be tolerable.

With a dependence detection mechanism in place, the next step is devising a history-based dependence prediction scheme. The most straightforward prediction scheme is to record and predict dependences as (load PC, store PC) pairs. Unfortunately, such a scheme may have to predict among many possible dependences since, as we have seen in Chapter 2, different instances of the same static store often observe dependences with instances of different static loads and vice versa. Furthermore, with such a scheme we may have to predict multiple dependences per dynamic store when its value is used by many loads. For these reasons, it is conceptually convenient to treat dependence prediction as a two step process. In the first step, a prediction is made on whether the given load or store *has* a dependence (i.e., the dependence status of the instruction), and in the second step, a prediction is made to decide with which load or store the dependence is with.

In Chapter 2 we have seen that the dependence status of instructions rarely changes. This observation suggests that in predicting whether a dependence exists, high-accuracy should be possible. Predicting the actual dependence however, requires more effort. Although it is conceivable and desirable to design a predictor that attempts to predict the actual dependence directly, we found it sufficient for the purposes of this work to use a level of indirection in representing dependences. To do so we use a scheme which assigns a common tag to all dependences that have common producers (stores) or consumers (loads) and uses that tag to identify all these dependences collectively. We can then determine which of all the possible dependences is currently observed by a mere inspection of the incoming instruction stream (this is similar to what is done for register dependences). Assignment of these tags can be done using an incremental approach. We explain this method using the example code fragment of Figure 4.4 part (a). There are two read-after-write dependences in this code: (STORE$_1$, LOAD) and (STORE$_2$, LOAD). The first time the loop iterates, one of the dependences, for example the (STORE$_1$, LOAD), is detected. A new tag is allocated and associated with both STORE$_1$ and LOAD. When later the (STORE$_2$, LOAD) dependence is detected, we associate the same tag with STORE$_2$ also (the tag is readily available as it is associated with LOAD). When this method is used it is possible to encounter a case where a dependent store and load have different tags assigned to them. Consider for example the code fragment of Figure 4.4, part (b). There are four possible dependences among the two store and the two load instructions. It is possible to encounter dependences in the following order: first (STORE$_1$, LOAD$_1$), then (STORE$_2$, LOAD$_2$), and finally, (STORE$_1$, LOAD$_2$). When the first two dependences are detected, two different tags are assigned to each of them as they share no instructions. As a result, when the third dependence is encountered, STORE$_1$ and LOAD$_2$ already have tags assigned to them which, unfortunately are different. At this point it is desirable to merge all dependences together by assigning one common tag to all four instructions. One way of achieving this is by replacing all instances of one tag with the other [62]. Doing so would probably require a broadcast mechanism, an undesired feature. Alternatively, we could use an incremental approach as the one suggested by Chrysos and Emer in the context of speculation/synchronization [17] (we have described this approach in Section 3.6.6). Throughout the evaluation of cloaking and bypassing we make use of this incremental approach noting that we observed virtually no difference compared to a full merge mechanism.

```
loop:
    if (cond) STORE$_1$ M$_a$
    else STORE$_2$ M$_a$
    LOAD  M$_a$
```

(a)

```
loop:
    if (cond) STORE$_1$ M$_a$
    else STORE$_2$ M$_a$
    if (cond$_{1)}$ LOAD$_1$  M$_a$
    else LOAD$_2$ M$_a$
```

(b)

Figure 4.4: *Code fragments that have multiple true dependences.*

4.2.2 Synonym Generation and Communication

In this section we discuss methods of generating synonyms for predicted dependences and how these synonyms are used by loads and stores to perform speculative communication. In cloaking, stores initiate the communication by generating a synonym in reaction to the prediction of a dependence. The synonym has a dual role: (i) it identifies the specific instance of the dependence (or dependences in the case of multiple consumers), and (ii) it is used as a handle by the dependent instructions to locate the storage element through which the communication will take place.

The exact encoding of the synonym is not important. However, it is desirable for the naming scheme used to provide different synonyms for unrelated communication at any given point of time. This may require generating different synonyms for different instances of the same static dependence when these instances are simultaneously active (e.g., the values have not been consumed yet). We approach this issue by separating memory dependences into two categories: (1) those dependences whose instance lifetimes are distinct in the original program order, and (2) those dependences whose instance lifetimes overlap in the original program order. Figure 4.5 shows examples illustrating these two cases: parts (a) and (b) show dependences that fall into the first category, while parts (c) and (d) show dependences that fall under the second category. As shown in part (b) dependence lifetimes do not overlap when the loop of part (a) is executed, however, as shown in part (d), when the loop of part (c) is executed, dependence lifetimes do overlap. The key distinction between the two categories is that in the second category, in between the dependent store and load, other instances of the store may be encountered (as shown in part (d) of Figure 4.5, the "store a[11]" appears in between the dependence (store a[10], load a[10])). In either case, assigning a new synonym to every instance of the store instruction is straightforward and can be done in numerous ways (for example, using a global counter). The challenging part is having the instances of the corresponding loads locate the appropriate synonym also. For dependences of the first category the functionality is similar to that used for register renaming (ignoring the potential size of the synonym name space). All we need to do is record the last synonym associated with each dependence. This can be done when the synonym is created by associating the tag representing the dependence with the synonym. The load can then locate the synonym and the corresponding store by using the dependence tag supplied by the memory dependence predictor.

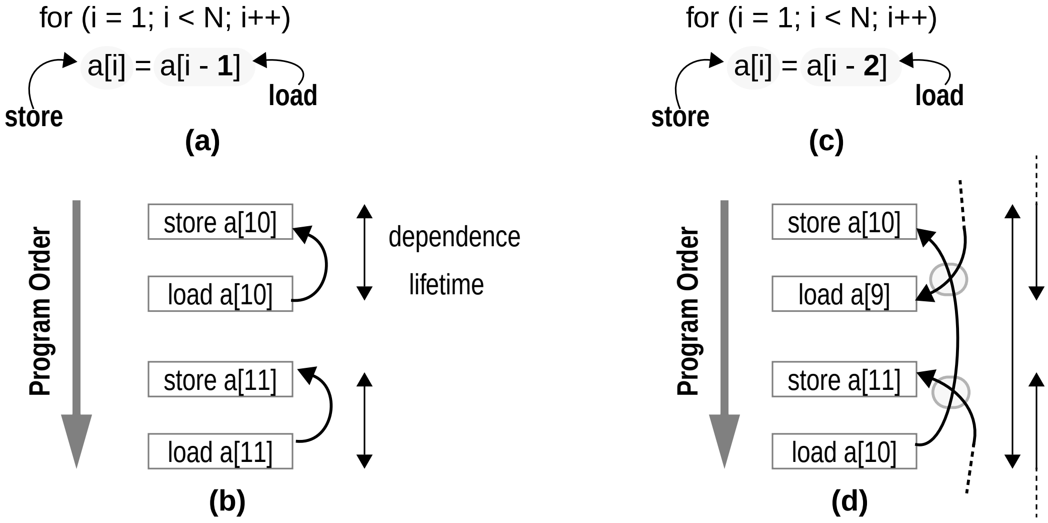


Figure 4.5: *Examples illustrating dependences whose lifetimes do not overlap (part (a)) or do overlap (part (b)). Dependences are marked with thick arrows. Dependence lifetimes are marked with thin arrows.*

For dependences that fall into the second category just recording the very last synonym assigned to the dependence is not sufficient. Instead, loads will have to determine which of all synonyms is the appropriate one. For example, in part (d) of Figure 4.5, "load a[10]" will have to locate the synonym assigned to "store a[10]" and not to "store a[11]". While support for these kind of dependences may be possible (i.e., providing a mechanism that allows loads to select among multiple instances of a store), we note that providing this support is not a requirement as it represents only a potential for increased performance. Accordingly, in this work we restrict our attention to synonym generation schemes that work correctly only for dependences whose instance lifetimes do not overlap (i.e., exhibit unit instance distances). However, even though we do not report these results, we note that we have experimented with a prediction mechanism that attempted to guess instance distances greater than one using an incremental approach (i.e., initially a dependence distance of 1 is tried and if that fails, a dependence distance of 2 is tried and so on). We found that while this mechanism was successful in handling cases where instance distances remained constant, it couldn't distinguish between incorrect instance distance predictions and dependence mispredictions. As a result, for virtually all programs prediction accuracy dropped significantly.

To perform the communication, physical storage has also to be provided for synonyms. The storage elements should provide space for the data value and an indication on whether the value is currently available. Finally, mapping synonyms to storage elements can be done in a variety of ways (e.g., using a direct mapped or a fully associative organization).

4.2.3 Verification

Because the communication that takes place in cloaking is based on dependence prediction, any values so obtained are speculative and have to be verified. This can be done by letting the dependent instructions also communicate via memory. The support required for invalidating and re-executing instructions that used incorrect data is no different than that required for memory dependence or value speculation [54]. Two options have been proposed to date: (i) squash or (ii) selective invalidation. In squash invalidation, all instructions after the mispeculation point are invalidated and re-executed. In selective invalidation only those instructions that used incorrect data are re-executed. While squash invalidation requires no more hardware than what is typically found in modern processors (it is also used to support control speculation) its performance penalty is relatively high. Selective invalidation on the other side offers relatively low performance penalty at the expense of added hardware cost and complexity. In fact, support for selective invalidation is in our opinion still in an experimental phase and whether such mechanisms are practically possible is still unknown.

4.2.4 Implementation Aspects

In this section we describe an implementation of the speculative memory cloaking technique and explain its operation by means of an example. Our goal is to demonstrate the feasibility of the required mechanisms and to provide insight about their complexity. A discussion of how the various components can be integrated in a typical pipeline is given in Section 4.6.6, where we discuss the exact mechanism we used to evaluate the performance impact of cloaking.

We partition the support structures in the following: (a) *dependence detection table* (DDT), (b) *dependence prediction and naming table* (DPNT), and (c) *synonym file* (SF). As we explained earlier, the DDT is used to detect dependences. An entry of this table consists of the following fields: (1) Data Address (ADDR), (2) Store PC (STPC) and (3) a valid bit. This information identifies the store that last updated the given word data address. The DPNT is used to identify, through prediction, those loads and stores that have dependences. It also provides the tags that are used to create synonyms for the dependences. An entry of this table comprises the following fields: (1) instruction address (PC), (2) dependence predictor (PRED), (3) dependence tag (DTAG), and (4) a valid bit. The instruction address identifies the load or the store this entry corresponds to. The purpose of the dependence predictor field is to provide an indication on whether a dependence exists. Finally, the dependence tag field is used to identify the dependences of this instruction. The SF is used to provide storage for synonyms. SF entries have the following fields: (1) name, (2) value, (3) full/empty bit, (4) valid bit. Based on the exact configuration used, some of the fields may not be required (e.g., we may not use a name field in a direct mapped SF) and some structures can be combined (e.g., we can merge the DPNT and the SF, or the register file and the SF).

The exact function and use of the support structures is best understood by means of an example. In the discussion that follows we use the working example of Figure 4.6 to demonstrate how an earlier detection of a dependence between a store and a load results in the streamlining of the inter-operation communication the next time the same dependence is encountered. In the discussion that follows we assume that the dynamic dependences result from the execution of the loop shown in part (a). Dynamically, a series of dependences will be observed between instances of the marked load and store. Each of the dynamic dependences will map to a different memory address (we assume that new space is allocated for each token).

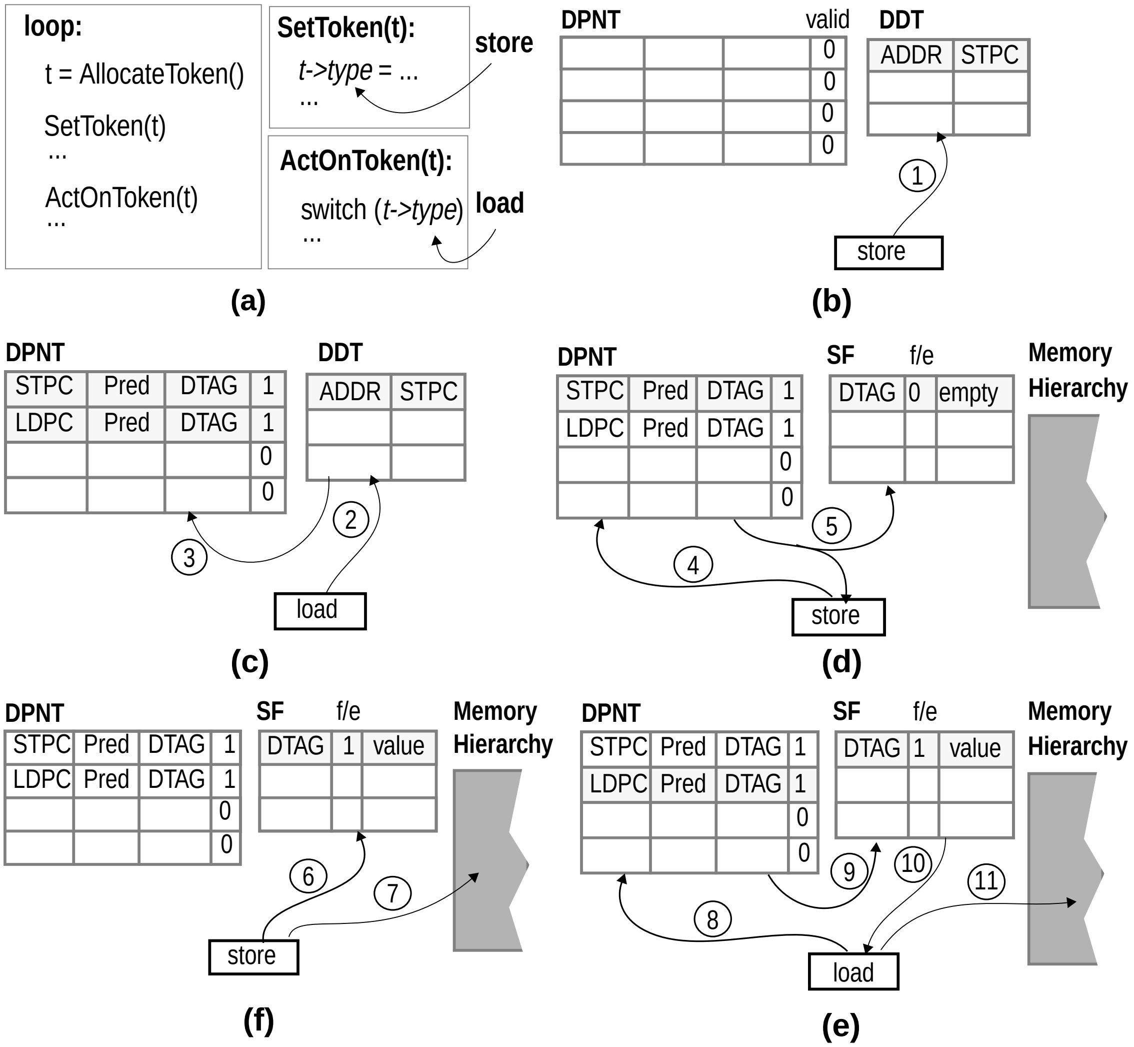


Figure 4.6: *Speculative Memory Cloaking: working example.*

In parts (b) and (c) we show the actions that lead to the detection of the dependence. In part (b), the first instance of the store executes and records in the DDT its PC and the data address it updated (action 1). Later on, in part (c), the first instance of the load using its data address probes the DDT (action 2) and determines that a dependence exists with the recorded the store. In reaction to this detection, two entries are allocated in the DPNT one for the load and one for the store (action 3). In addition, a tag is created for the dependence, and it is recorded in both entries. (Since the operation of the DDT has been described in steps 1 through 3, it is not shown in the remaining parts of the figure.)

In parts (d) through (f) the actions that lead to the cloaking of a later instance of the dependence recorded in part (c) are shown. Cloaking is initiated when, as shown in part (d), a later instance of the store enters the instruction window. The PC of the store is used to probe the DPNT for a matching entry (action (4)), and since one is found, its predictor is used to determine whether cloaking should occur. Assuming that the predictor indicates so, a synonym is generated based on the tag recorded in the DPNT entry (for the purposes of this discussion the tag of the DPNT and the synonym are the same), and it is used to allocate space in the SF (action 5). The full/empty bit of the SF entry is set to empty to indicate that the value is not yet available, whereas, the store also records the location of the SF entry since the actual data value, when it becomes available, will have to be written in the SF entry (part (e), action 6). Eventually, the store also accesses the traditional memory hierarchy (part (e), action 7).

When the next instance of the load enters the window (part (f)), as it was done previously with the store, its PC is used to probe the DPNT (action 8). After a match is found and a dependence status prediction is made, the tag recorded in the DPNT entry leads to the generation of the same synonym generated previously for the store. This synonym is used to access the appropriate SF entry (action 9) and to obtain the data left there by the store. At this point the load may use this data to execute speculatively (action 10). Later on, when the data address becomes available, the load accesses the traditional memory hierarchy to obtain the actual data value (action 11). This value is compared against the value read previously from the SF and appropriate action is taken if the two values differ. At this point we may also update the predictors in the DPNT entries for both the load and the store (to locate the DPNT entry for the store the SF entry will have to record the store's PC).

4.3 Speculative Memory Bypassing

With cloaking, values can flow quickly from stores to loads. However, in typical load/store architectures, stores and loads do not compute values. Loads and stores are simply used to pass the values that some other instructions produce to some other instructions that consume them. Which is to say that loads and stores are part of the memory interface acting as agents between the register and memory name spaces. It is interesting to consider why memory is used for the communication instead of registers. One reason is that the register name space is relatively small. When there are more live values than registers, memory is used. Another reason is programming conventions (for example, caller- and callee-saved registers). Another reason is that in some cases the compiler is unable to establish the dependence statically. Finally, because of the limited addressability of registers, memory has to be used to implement many commonly used data structures.

In this section we present speculative memory bypassing, a conceptually straightforward extension to speculative memory cloaking that aims at eliminating the overheads associated with performing memory communication via loads and stores. S*peculative memory bypassing* converts DEF-store-load-USE chains into DEF–USE chains whenever the load-store dependence is predicted and the DEF and USE instructions coexist in the instruction window. In this case, the value can speculatively flow directly from the actual producer (DEF) to the actual consumer (USE). This concept we illustrate in Figure 4.7 using the *DEF–store–load–USE* chain shown in part (a). Even though cloaking may allow the value to be speculatively communicated between the store and the load, the value will still have to travel through these two instructions before it can reach USE. However, as shown in part (b) with speculative memory bypassing, the value can be sent directly from DEF to USE. As was the case with cloaking, this communication is speculative and has to be verified via the traditional memory name space. This does not necessarily imply that we have to access the memory system. (For example, we might be able to establish that the value is correct by comparing the addresses of loads and stores. In this case, bypassing and cloaking may also help in reducing data cache bandwidth. However, we do not investigate this option.)

Speculative memory bypassing can be implemented as a straightforward extension to speculative memory cloaking. We explain the exact process using the working example of Figure 4.7, part (c). At step (1), instruction *DEF* is decoded and register renaming creates a new name, *TAG1*, for the target register *R1*. At

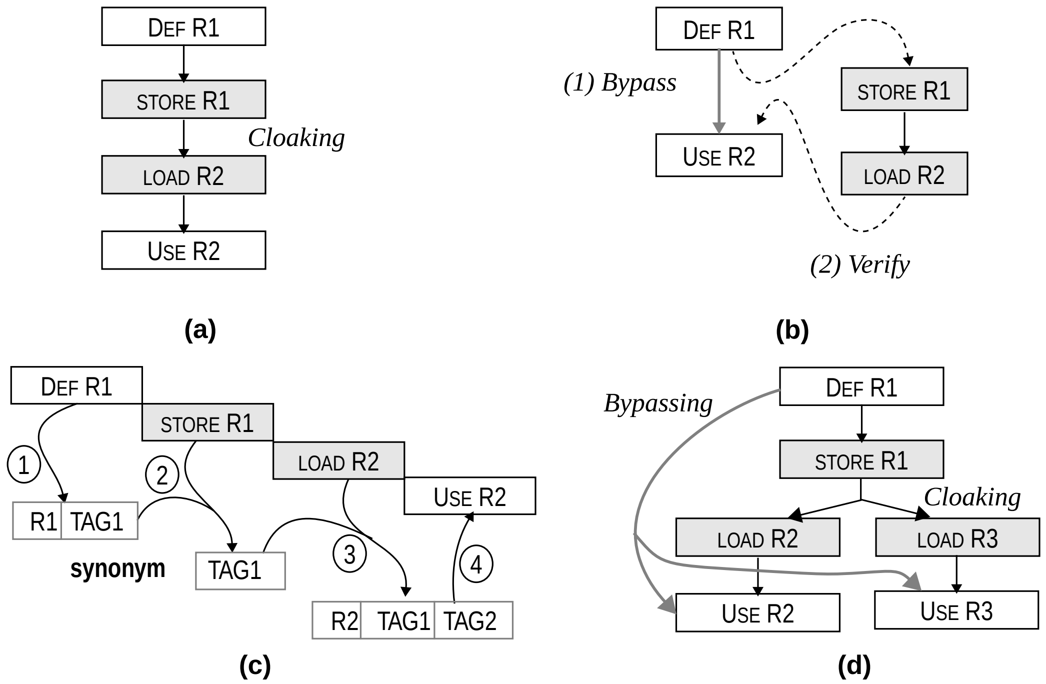


Figure 4.7: *Speculative Memory Bypassing. (a) Communication path followed when the traditional memory interface is used. (b) Communication path followed when speculative memory bypassing is in use. (c) How the store and the load are taken off the communication path. (d) Simultaneous bypassing of multiple RAW dependences.*

step (2), the store instruction is decoded and as part of register renaming determines the current name *TAG1* of its source register *R1*. In parallel, via the use of cloaking, a synonym is created for the memory communication. To perform bypassing, at this point we associate the synonym with the current name *TAG1* of the store's source register *R1* (this association can be done by recording the register in the synonym file). At step (3), the load instruction is decoded and register renaming creates a new name *TAG2* for the destination register *R2*. In parallel, via the use of cloaking, the load locates the synonym and hence determines the name *TAG1* of the store's source register *R1*. In doing so, the load has determined the storage (e.g., physical register or reservation station) where the actual producer *DEF* will place or has placed the value. This name is speculatively associated with the target of the load *R2*. This way, when at step (4) *USE* is decoded, it can determine that its source register *R2* has two names: one actual *TAG2* and one speculative *TAG1*. By using the speculative name *TAG1*, *DEF* can link directly to *use* and execute speculatively as soon as *DEF* produces its value. Later on, after the load has accessed memory, the integrity of the communication can be verified.

Note that speculative memory bypassing naturally extends for dependence chains that include more than one memory dependence; whenever a store detects that its source register has a speculative name, it can optimistically pass it via the synonym. However, we do not study such an extension in our evaluation. Moreover, speculative memory bypassing becomes more attractive when a store has multiple dependences as it may help in further reducing latency compared to cloaking when write-back bandwidth is limited. In such cases and when only cloaking is used, each of the loads that get a value from cloaking will need to individually propagate this value to their dependent instructions. When write-back bandwidth is limited (for example the store had dependences with four loads and there are only two write-back ports available) propagating the speculative value will be delayed. When bypassing is used, all dependent instructions will obtain a speculative value as soon as the actual producer executes.

Finally, bypassing can also be used to eliminate the need for an explicit synonym file at the expense of reduced coverage. In such a design, prediction will have to be restricted to only those dependences that are visible from within the instruction window. In this case, no synonym file is required as bypassing associates synonyms with pre-existing storage elements (i.e., physical registers or reservation stations).

4.4 Extending Cloaking and Bypassing to Support Data-Sharing

Having presented how cloaking and bypassing can be used to reduce the overheads associated with communicating values through the traditional memory name space, in this section we explain how these methods can be extended to also streamline data-sharing through memory. Data-sharing occurs when a memory value is accessed more than once. Data-sharing is possible even when the value was written by a store. The case that interests us here is when no producing store is visible through the dependence detection and prediction mechanisms. This is possible when: (1) the value was produced outside the scope the program (for example via a system call or the value is a constant), (2) the store that wrote the value has executed long ago escaping detection, and (3) when the store-load dependences are not stable enough to facilitate prediction.

Extending cloaking and bypassing to also support read-after-read dependences is straightforward. All we need to do is detect read-after-read dependences and treat some of the loads as producers of values. Detecting read-after-read dependences can be done by recording loads in the dependence detection table whenever no store is found at the same address. When subsequent loads access memory, they can also access the dependence detection table and identify the preceding load that also accessed the same memory address. At this point we can mark the preceding load as a candidate for read-after-read cloaking and bypassing (this can be done using an additional bit indicator in the DPNT entry). The next time that load is encountered, we can use the existing cloaking and bypassing mechanisms to associate the loaded value with a synonym. The only difference is that for bypassing purposes, loads should be treated as the actual producers of values in contrast to stores that write a value that some other instruction is producing. That is, in this case we convert a series of LOAD$_1$-USE$_1$...LOAD$_N$-USE$_N$ chains into a single LOAD$_1$-USE$_1$...USE$_N$ producer-consumer graph.

4.5 Related Work

Techniques related to cloaking and bypassing can be broadly classified into the following categories: (1) techniques to eliminate memory communication via register allocation, (2) techniques to reduce load latency via address or value prediction, and (3) techniques that attempt to link loads and stores similarly to what is done in cloaking.

4.5.1 Register Allocation Alternatives

An obvious alternative to cloaking is register allocation where instead of using a memory location to pass or hold a value a register may be used. Register allocating a value eliminates the need for load or store instructions and memory communication altogether. However, it is not always possible to register allocate a value for the following reasons: The register name space is typically limited in size (e.g., 32) and as a result, space may not be available to register allocate a value. Some dependences (RAW or RAR) may not be detectable statically either because ambiguous dependences exist or because dependence behavior is dynamic in nature. Registers also offer limited addressability. Other reasons why memory allocation may be used include calling and other programming conventions (e.g., dynamically linked libraries and objects).

In this context, a related technique is the *C Machine Stack Cache* proposed by Ditzel and McLellan in [19]. In their proposal, a small data cache is introduced on which stack allocated variables can be mapped. Using a combination of software (ISA extensions) and hardware, memory references to stack variables are redirected to this the stack cache (provided that the stack pointer value is available before the corresponding instruction is prefetched into the instruction cache).

To reduce the amount of memory communication that results from increased register pressure, a larger register name space may be used. While this approach is conceptually straightforward, it is not free of trade-offs and of implications. Increasing the set of architecturally visible registers requires changes in the ISA. Finding opcode space to accommodate wider register specifiers may not be possible without increasing the width of instructions. In the latter case, care must be taken to avoid performance degradation due to secondary, I-cache effects. Furthermore, an increased register file may lengthen the base cycle time of the processor and increase the amount of information that has to be check-pointed on context switch.

Better disambiguation methods (see Chapter 3 for pointers) may help in reducing the communication that results from ambiguous dependences. Dietz and Chi proposed *Cregs* a hybrid software/hardware approach to register allocation in the presence of ambiguous memory dependences. In their proposal, registers are augmented with memory address tags. This facilitates the register allocation of memory values in the presence of ambiguous memory dependences. Hardware is responsible for keeping these registers coherent with memory. The goal of this technique is avoiding accessing memory whenever no aliases exist. There is no provision to allow re-ordering of instructions that use CReg allocated values.

While inter-procedural register allocation [93, 49] may help in reducing the memory traffic caused by the various programming conventions, such approaches are also not trade-off-free and may not always be applicable. For example, an inter-procedural register allocator may have to rely on cloning (i.e., creating a specialized copy of the called function) and for this reason may have to balance cloning possibilities against increases in instruction-cache footprint. Inter-procedural register allocation may also be limited by imprecise call-graph information and by the use of dynamically linked objects and libraries. Compared to the aforementioned techniques, a potential advantage of cloaking and bypassing is that they need not be architecturally visible and do not require any changes to existing codes and architectural interfaces. Moreover, both techniques offer the possibility of capturing memory behavior that is not detectable statically.

4.5.2 Address Prediction Based Techniques

Numerous techniques that attempt to predict the data addresses of loads and stores have been proposed to reduce the access latency of loads both in hardware and in software [5,7,9,21,30,51,74]. Even though no attempt is made to establish explicit links between dependent instructions, these techniques may, as a side effect, reduce the latency of the communication of load-store or load-load dependences. These techniques may do so, provided that the data address accessed by the load is correctly predicted and that the source store or load has executed (i.e., both the data address and value are available). Cloaking may streamline the communication even if the access pattern defies prediction and does not require that the store address is known in the case of store-load dependences.

A closely related technique to the ones mentioned in the previous paragraph is *sum-addressed memory* or *SAM* [55]. While not a prediction technique, SAM can be used to reduce the latency associated with address calculation. In SAM, address calculation takes place in the decoder array of the data cache itself. Even in this case, cloaking and bypassing may offer additional latency reduction for the following reasons: (1) the synonym file access latency may be shorter compared to the 1st-level data cache latency due to size differences, (2) either of the dependent instructions may have not calculated their addresses when the data becomes available, and (3) the source instruction may have left the instruction window so that the dependent load can obtain its value at the time it enters the instruction window.

Finally, although not an address prediction technique the *Knapsack* [8] proposed by Austin, Vijaykumar and Sohi, provides a method of reducing load access latency to 0 cycles. In this approach a data cache, the knapsack, is introduced at the decode stage of the pipeline and is mapped onto a continuous portion of the memory address space. Zero access latency is achieved by accessing the knapsack in the decode stage using just the offset field of the opcode. By utilizing appropriate values for the base register, the offset field is often sufficient for locating the appropriate storage location in the knapsack.

4.5.3 Value Prediction

A technique closely related to the cloaking is *load value prediction* [54], a special case of value prediction [53]. In this technique, a prediction is made on the value that a load will read. As with cloaking, the prediction can be done early in the pipeline using, for example, the PC of the load instruction. This technique may effectively reduce the latency of memory communication independently of whether the corresponding load has a dependence or not. The success of this approach relies on the ability to track and predict the actual values. The fundamental difference between value speculation and cloaking is that cloaking does not directly predict the loaded value, rather it predicts its producer or another load that also accessed the same location. While load value prediction relies on regularity in the load value stream, cloaking relies on regularity in the memory dependence stream. As we will demonstrate in the evaluation section, dependence regularity is stronger than value regularity for a significant fraction of executed loads (Section 4.6.4.4). Even so, whenever value regularity is strong, value prediction may be advantageous for two reasons. First, the information required to do value prediction is local to the load instruction, while in cloaking we need to somehow associate the store and the load. Second, if the value is relatively constant then value prediction may supply a load value earlier than cloaking would. Widigen, Sowadksy and McGrath also describe a technique similar to load value prediction in [94].

4.5.4 Techniques Similar to Cloaking or Bypassing

Techniques similar to cloaking are “*memory renaming”*, proposed by Tyson and Austin [89] and *alias prediction* proposed by Lipasti [52]. “Memory renaming” is conceptually identical to cloaking with the exception that RAR dependences are not used. Other technical differences exist, for example, (1) dependence detection in “memory renaming” occurs either in the load/store queue or by propagating store PCs up in the memory hierarchy, (2) no mention on how to support loads with multiple dependences is discussed, and (3) communication takes place through the load/store queue. This technique —which should not be confused with the memory renaming [6] that takes place in a typical load/store queue or any other mechanism supporting multiple versions of the same memory address for the purposes of out-of-order execution in the presence of WAR dependences (e.g., [27])— relies on value prediction for those loads that do not exhibit detectable RAW dependences. As we demonstrate in the evaluation section, RAR dependence prediction can be used to capture loads that value prediction cannot. However, a significant fraction of loads with RAR dependences are also amenable to value prediction. Accordingly, in Section 4.6.6.4 we investigate possible combinations of cloaking/bypassing and value prediction.

Alias prediction can be viewed as a restricted form of cloaking for RAW dependences. In this technique, a prediction is made on whether a load will read a value from a preceding store that co-exists in the instruction window. The value is then located by predicting the store buffer location where the value is to be found.

Another related technique is *Instruction Reuse* [79]. In instruction reuse a load may obtain a value prior to address calculation provided that it is established that the same address will be accessed and that no store has written to that address since the last time the load was encountered (i.e., no RAW dependence has been observed). In this case the value so obtained is known to be correct (i.e., it is non-speculative).

A mechanism similar to speculative memory bypassing for store-to-load communication (RAW dependences) was recently studied by Jourdan, Ronen, Bekerman, Shomar and Yoaz [41]. In their proposal, bypassed loads do not necessarily have to access memory. This is achieved by comparing the addresses of the relevant store and load. If the addresses are the same then the value obtained through cloaking or bypassing is known to be correct.

We should note that an effect similar to speculative memory bypassing is possible using a pure software technique when ambiguous dependences are present. In such an approach, two instructions are allowed to communicate speculatively via a register while the communication is verified via memory using loads and stores. Both verification and mispeculation handling has to be done explicitly in the code itself. For example, a system that may perform this optimization is DAISY [20] where this optimization may be applied dynamically by rewriting part of the code during run-time.

A software guided approach to speculative memory cloaking was investigated by Reinman, Calder, Tullsen, Tyson and Austin [69]. In their approach, new instructions are introduced that allow the compiler to communicate speculative memory dependences to the hardware. Speculative memory dependences are communicated using load and store instructions with explicit tags. These tags are allocated in a new name space, separate to memory or registers. Profile information along with a number of heuristics is used to select the most stable memory dependences. Tags are also introduced to allow value prediction and dependence speculation to be used whenever deemed advantageous. Reinman and Calder also performed a comparative study of load value prediction, memory dependence speculation/synchronization and of a variation of the “memory renaming” technique of Tyson and Austin [68] (they also used the incremental algorithm of Chrysos and Emer [17] to generate synonyms and used a 4K DDT [61] instead of relying on the load/store queue or on propagating store PCs up in the memory hierarchy [89]). As per the proposal of [89, 61] they used only RAW prediction for the purposes of cloaking. Using a processor model that does not speculated on memory dependences as their base and assuming a selective invalidation mechanism, they concluded that value speculation offers the most benefits and suggested that it should be the first technique to be implemented in future generation processors.

Finally, we should note that we initially reported cloaking and bypassing for RAW dependences in [62].

4.6 Evaluation

In this section we present experimental evidence in support of the utility of the techniques we propose. We use a two step approach. Initially we investigate cloaking and bypassing ignoring timing considerations. This allows us to study cloaking and bypassing without having to be concerned with side-effects introduced by their interaction with other execution techniques. Once we have studied the potential of cloaking and bypassing we then consider how a particular implementation performs in a out-of-order dynamically scheduled processor environment. Finally, we compare this cloaking/bypassing mechanism with a value prediction mechanism and investigate two possible combinations of value prediction and cloaking/bypassing.

The rest of this section is organized as follows: The first step in using cloaking is building the dependence history necessary to predict subsequent dependence behavior. For this reason, in Section 4.6.1 we measure the fraction of memory dependences observed as a function of the memory dependence detection table size. In Section 4.6.2 we investigate an aggressive cloaking mechanism and study its accuracy. In Section 4.6.3

we investigate various confidence predictors in an effort to reduce mispeculations while keeping the fraction of loads that get a correct value from cloaking relatively high. Having shown that a relatively simple confidence predictor can be used to meet our goal, in Section 4.6.4 we present a characterization of the memory communication that is handled by a cloaking mechanism that offers relatively high accuracy. A description of the characteristics we consider along with a justification on why we do so is given at the beginning of that section. We continue the evaluation of cloaking and bypassing by considering what effect finite prediction structures have on its accuracy in Section 4.6.5. In Section 4.6.6, we measure the performance impact of a combined cloaking and bypassing mechanism on a fairly aggressive superscalar processor with out-of-order characteristics. We initially study how performance is affected by the use of squash and selective invalidation. We then compare the cloaking/bypassing mechanism with a straight-forward last-value load value predictor and conclude by studying two possible combinations of cloaking/bypassing and value prediction.

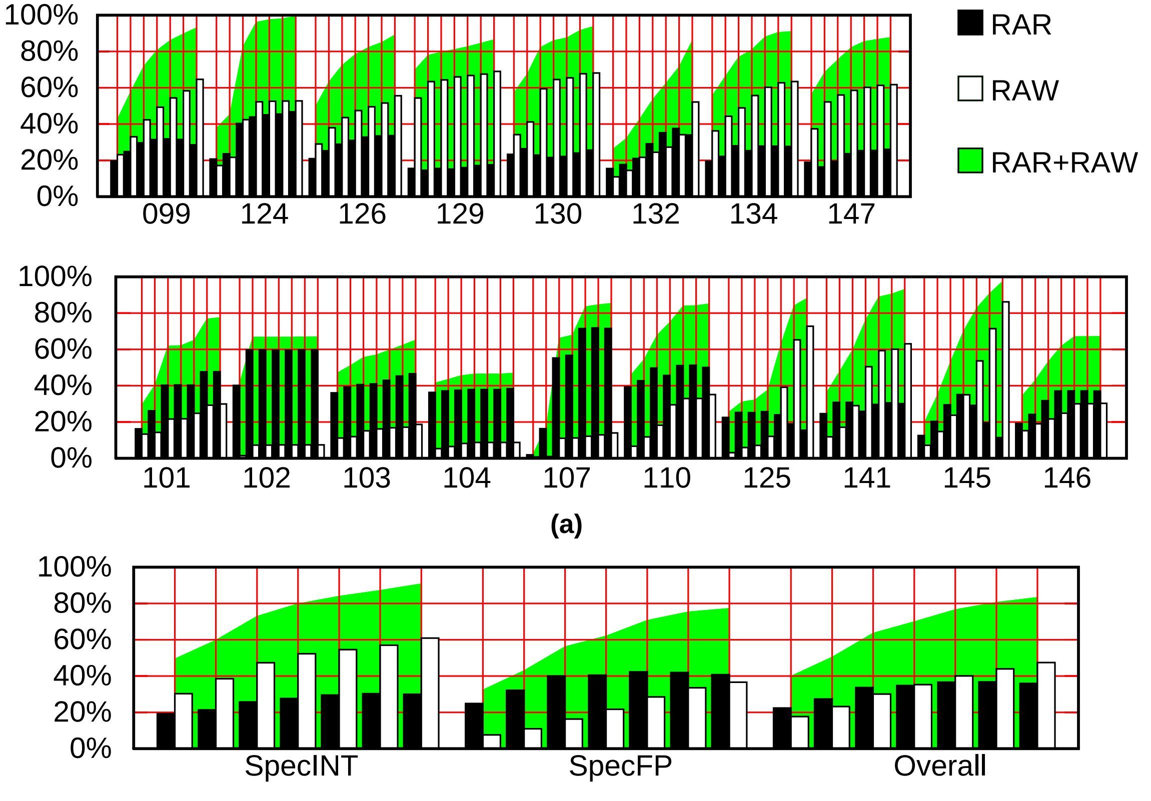
4.6.1 Memory Dependence Detection

In this section we measure the fraction of memory communication and data-sharing activity that is visible with various dependence detection table (DDT) sizes. Dependence detection is the first step in applying cloaking and bypassing and is used to build the dependence history required to predict future dependence behavior. The measurements presented in this section allow us to estimate the fraction of loads we can hope to handle with cloaking and bypassing. We have presented similar measurements in Chapter 2 where we measured the fraction of dependence activity that is visible from within various address window sizes: a fully associative, DDT of size *n* with LRU replacement is equivalent to an address window of the same size as per our definition given in Chapter 2. In this section we take a closer look at the lower end of the spectrum, considering dependence detection table sizes we consider feasible and reasonable in the context of modern and next generation processors. It should be noted that the results of this section provide only an indication of the fraction of loads cloaking and bypassing may handle. The actual fraction of loads that do get a correct value via these techniques may be larger or smaller as compared to the fraction of loads whose dependences is detected. It can be smaller as, for example, the mechanisms we simulate are not able to handle dependences with distances greater than one. It can be larger as the address distance of a dependence may fluctuate over time. As a result, some instances of the dependence may not be visible while cloaking and bypassing may still be applied as the result of an earlier detection of another instance of the same static dependence.

Figure 4.8 reports the fraction of dynamic (committed) loads that experience a RAW or a RAR dependence as a function of DDT size. Whenever a load experiences both a RAR and RAW dependence we only account for the RAW dependence. We do so as cloaking will give preference to RAW dependences in these cases. We consider detection table sizes in the range of 32 and up to 2K entries. In part (a), results are shown per program, while in part (b) we present averaged results for the integer, floating point and all programs. To aid in the interpretation of these results we present RAW and RAR fractions as separate bars. Moreover, the grey shaded area reports the sum of the two bars and represents the overall fraction of loads that have either a RAW or a RAR dependence detected.

Focusing first on the averaged results, we can observe that a large fraction of loads get their value via a dependence that is visible even with relatively small DDTs. For integer codes about 50% of all loads experience a RAW or a RAR dependence within the last 32 addresses accessed. This phenomenon is less pronounced for the floating point codes where about 30% of loads experience dependences within the same range. However, as we move toward larger DDTs, the fraction of loads with dependences rises, approaching roughly 90% (integer) and 80% (floating-point) for the 2K-entry DDT.

The relative fractions of RAR and RAW dependences, and for this reason their importance are dissimilar for the two classes of programs. While in the integer codes RAW dependences are almost twice as frequent



(a)

(b)

Figure 4.8: *Fraction of loads that have RAW or RAR dependences as a function of dependence detection table size. Range is 32 to 2K entries in steps that are powers of two. (a) Per program results. (b) Average over integer, floating point and all programs.*

as RAR dependences are, in the floating point codes the roles are almost reversed for the smaller DDT sizes. Nevertheless, as we consider larger DDTs, the fraction of RAW dependences increases in the floating point programs, and for a 2K entry table RAW dependences are almost as frequent as RAR ones. Interestingly, as we move toward larger DDT sizes, RAW dependences become increasingly frequent. The relative increase is higher for smaller sizes. While RAR dependence frequency also increases with the detection table size in the range of 32 to 512, virtually no increase is observed as we move toward sizes of 1K or higher. For the floating point programs we even observe a decrease in RAR dependence frequency between 1K and 2K table sizes (a decrease in RAR frequency with increased DDT size is also exhibited on a per program basis). This phenomenon is explained by the increased frequency of RAW dependences: Some of the RAR dependences are among loads that read a value written by a preceding but distant store. When smaller DDTs are used, the store is evicted from the DDT due to limited space and the RAW dependences escape detection.

Similar observations apply on per program basis. An increasing number of loads with RAW or RAR dependence is observed as we consider larger DDTs. Again the relative increase is higher in the range of 32-256 entries for most programs. RAR dependences are more frequent than RAW dependences for most floating programs, while the opposite is true for the integer programs. And finally, variation is exhibited in the frequency of RAR dependences as we move toward larger DDT sizes.

The results of this section suggest that a DDT of moderate size (e.g., 128 entries) can capture dependences for large fractions of loads (roughly 70% and 60% for the integer and floating-point programs respectively).

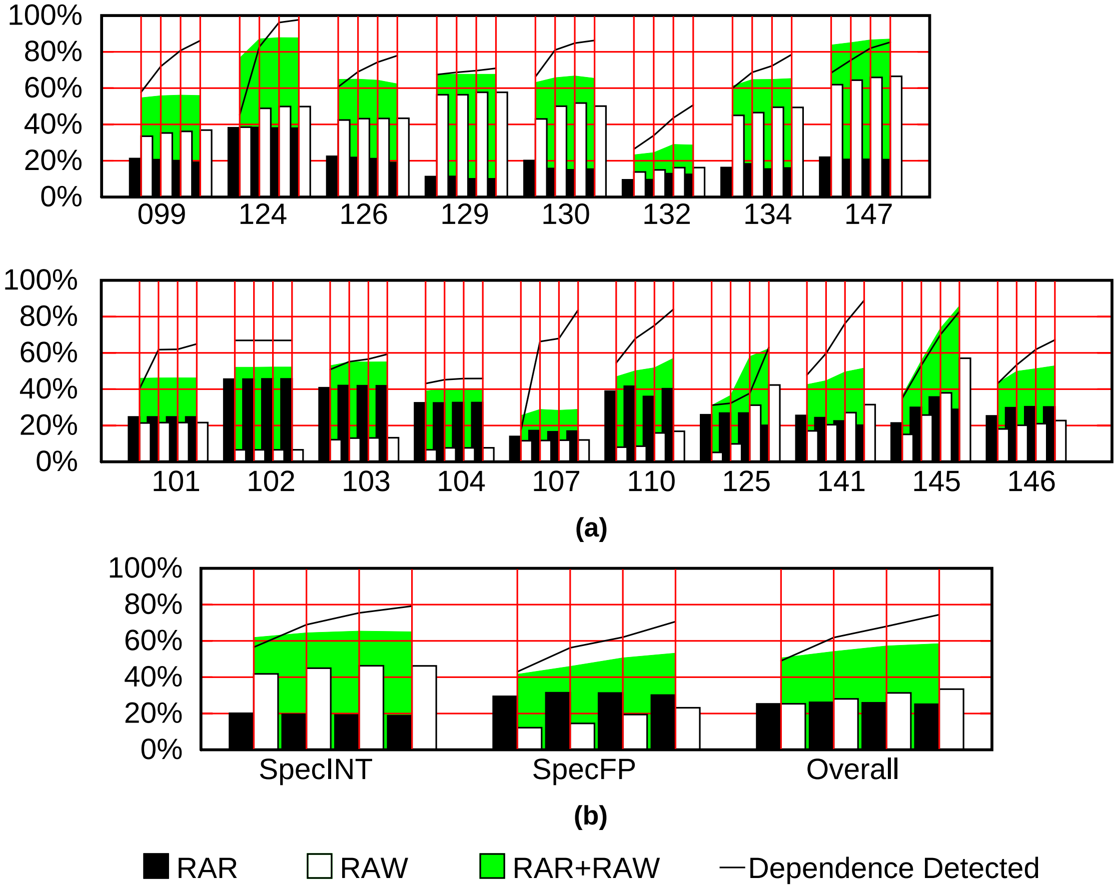
4.6.2 Cloaking Coverage And Mispeculation Rates

The results of the previous section provide a first indication that cloaking may be able to capture a significant fraction of all dynamic loads. However, detecting dependences is just the first step. The next step is predicting dependences and passing the values from producers to consumers. We use two metrics to measure the accuracy of cloaking: *coverage* and *mispeculation rate.* Both metrics represent a fraction of all executed loads. We define *coverage* as the fraction of loads that do get a correct value via cloaking. We define *mispeculation rate* as the fraction of loads that get an incorrect value from cloaking.

For the purposes of this study we assume infinite dependence prediction and naming tables (DPNT) and measure the fraction of loads whose value is correctly predicted for various DDT sizes. To get a rough upper bound on the fraction of correctly predicted load values, we use an non-adaptive single bit predictor. This predictor will always cause cloaking to be performed for all instances of the corresponding instruction once a dependence is detected. In Section 4.6.3 we will consider adaptive predictors to reduce mispeculations. Figure 4.9 shows the coverage obtained for DDTs of 64, 128, 256 and 512 entries. In part (a) we show per program measurements. Averaged measurements are shown in part (b). The measurements presented are: (1) the total fraction of loads that get a correct value (shaded area), (2) a breakdown of these loads depending on whether they got the value via a RAW (white bars) or a RAR (dark bars) dependence, and finally (3) the fraction of loads for which a dependence is detected (line).

Focusing first on the averaged results (part (b)), we can observe that the majority of loads that have a dependence do obtain a correct value via the cloaking mechanism. On the average, about 65% (integer) and 53% (floating-point) of all loads get a correct value via cloaking when a 512 entry DDT is used. The average for all programs is about 59%. Interestingly, cloaking sometimes supplies a correct value for loads that have no dependences detected (for example, 147.vortex and a 64-entry DDT) supporting our earlier observation that an earlier detection of a memory dependence may be used to perform cloaking even when subsequent instances of the same dependence escape detection. However, and as expected, not all loads that have dependence detected get a correct value from cloaking. The fraction of loads that do have a dependence detected and do not get a correct value from cloaking is negligible for the smaller DDTs. This fraction increases as we move toward larger DDTs. Furthermore, it can be seen that while floating programs experience a steady increase in cloaking coverage, the integer codes see virtually no increase in cloaking coverage as we move toward larger DDTs. More interestingly, a slight decrease in cloaking coverage is observed between the 256-entry and the 512-entry DDTs. These phenomena can be explained by taking a closer look at individual program behavior.

As it can be seen by the results of part (a), even on a per program basis, the majority of loads that have a dependence detected also get a correct value from cloaking. The exception is 107.mgrid. Two are the causes of this behavior: (1) RAR dependences that have non-unit instance distances (see discussion of Section 4.2.2), and (2) RAR dependences that not stable. RAR dependences with non-unit instance distances are mostly caused when small arrays are traversed multiple times by different pieces of the code. The cloaking mechanism we used in this experiment fails to correctly communicate values as it always attempts to link



(a)

(b)

Figure 4.9: *Cloaking coverage as a function of Dependence detection table size. Range is 64 to 512 entries. Shown is the fraction of loads that obtain a correct value via a RAW or a RAR dependence (white and black bars respectively). Also shown is the fraction of loads that get a correct value independently of the dependence type (shaded area). Finally, the fraction of loads for which a dependence is detected is shown (black line). (a) Per program measurements. (b) Averaged measurements.*

with the last instance of the source load. Also, some of the RAR dependences detected are not stable. This is best explained using the following example (the actual behavior of 107.mgrid is more complicated):

```
1: LOAD A[ 0 ]
   ...
2: LOOP i
3:LOAD A[ i ]
4:LOAD A[ i ]
```

During the first iteration of the loop, RAR dependences are detected among the load of line 1 and the loads of line 3 and 4. Our greedy, non-adaptive approach to building dependence sets and marking producers and consumers will mark the first load as the producer and the other two loads as consumers for cloaking purposes. However, as execution progresses, the initial RAR dependences are not observed anymore and cloaking fails even though memory dependences are still detected.

For most programs, cloaking coverage remains relatively unchanged as we consider larger DDTs. Beyond an 128-entry DDT, noticeable increases in coverage are observed only for 145.fpppp and 125.turb3d and to a lesser extent for 132.ijpeg, 110.applu, 141.apsi and 146.wave5. (For 132.ijpeg virtually no increase is observed beyond a 256 entry table.) One reason why cloaking coverage does not increase is that, in some cases, increasing the DDT size does not result in a considerable increase in the number of detected dependences. This is the case for 129.compress, 102.swim, 103.su2cor and 104.hydro2d. However, for the rest of the programs, more dependences are typically detected if we use a larger DDT. In those cases, the reason why these dependences do not result in an increase in cloaking coverage is that in their majority they do not exhibit unit dependence distances (i.e, their dependence lifetimes overlap, see discussion of Section 4.2.2). This observation is supported by the measurements presented in Figure 4.10. In this experiment we assumed perfect prediction of the source instruction's identity and measured whether the value read is the same as the one read or written by the last instance of the source instruction. The lower part of the bar represents the fraction of loads that read the same value as the one read or written by the last instance of the source instruction (which includes all dependence with unit instance distance). The top part of the bar (light grey) represents the fraction of loads that don't (non-unit dependence distance). As it can be seen by comparing with the data of Figure 4.9, there is high correlation between (1) the fraction of loads that do not get a correct value via cloaking but do have a dependence detected and (2) the fraction of loads that exhibit non-unit dependence distances. As we mentioned before, the cloaking mechanism used in this experiment is not designed to handle non-unit distances. In Figure 4.10, we can observe an anomaly for 107.mgrid where a decrease is observed in the fraction of loads that experience unit instance distances when we move from an 256-entry to a 512-entry DDT. This decrease is the result of loads that have multiple RAR dependences some of which are distant and less stable than the less distant ones. As we consider larger DDTs, the distant and less stable RAR dependences are used to make predictions. As we explain in the next paragraph similar behavior is observed in other programs with the non-perfect dependence predictor.

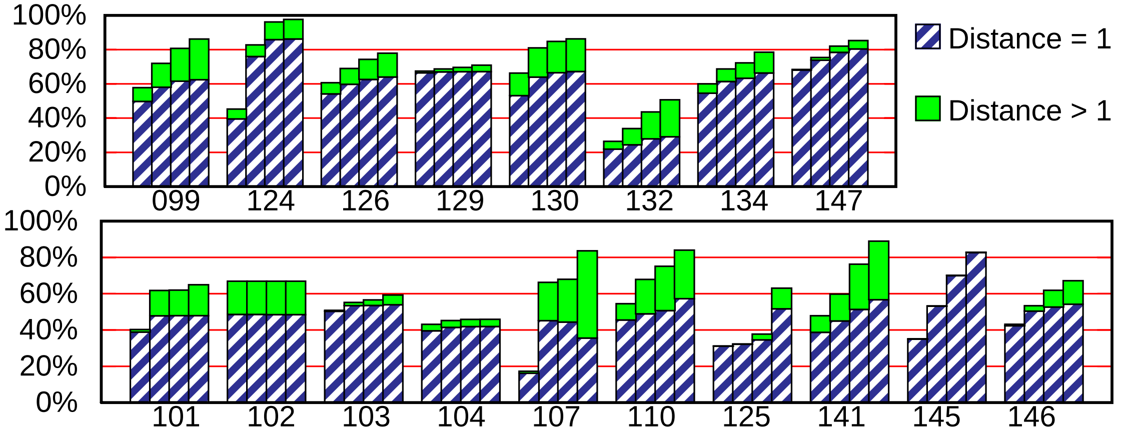


Figure 4.10: *Instance Distance Breakdown As A Function Of Dependence Detection Table Size. Assuming perfect dependence prediction, we measure the distance to the correct instance of the source instruction. Distance of 1 corresponds to the last instance of the source instruction. Results are show for dependence detection tables of 64, 128, 256 and 512 entries (left to right in that order).*

Returning to the results of Figure 4.9, we can observe that in some cases, a slight decrease in cloaking coverage is experienced as we increase the DDT size from 256 to 512 entries. These programs are 126.gcc, 130.li and 132.ijpeg. Two factors combine to cause this behavior: (1) the intrinsic dependence behavior of these programs, and (2) the use of a non-adaptive cloaking predictor. Some loads have multiple RAR dependences with the more distant ones being less predictable. As we consider larger DDTs, the distant and less predictable dependences take preference causing a decrease in coverage. A similar phenomenon is observed on some loads that have multiple RAW dependences that do not occur every time the corresponding store instructions are encountered. Consider for example a load that has a frequent dependence with store1 and a less frequent dependence with store2. Furthermore, assume that the dependence with store2 escapes detection with a smaller DDT but is detected when we use a larger DDT. If store2 appears often enough in between instances of the store1 and load instructions, it may occasionally overwrite the value of store1 (in synonym space) causing a decrease in cloaking coverage. Because in these experiments we used a non-adaptive dependence predictor, these less predictable dependences cannot be filtered out. However, we should note that the decrease in cloaking coverage is relatively small and for this reason it may not have a significant impact on performance.

While for most programs cloaking coverage is smaller than the fraction of loads that have dependences detected, for 124.m88ksim, 147.vortex and 125.turb3d cloaking coverage is significantly larger (for the smaller detection table sizes). This is possible, as the distance between the dependent instructions may vary over time. This is the case in 147.vortex for example. About 70% of loads instances that have no dependence detected but do get a correct value via cloaking in 147.vortex belong to the memcpy and ChkGetChunk functions. It seems that this program copies a large number of small objects, often multiple times. Each of these calls to memcpy is separated by a different amount of processing. What happens to each object is data dependent. Since the main memcpy loop is unrolled, different static instructions are used to copy different part of the data object being copied. Whenever the data object is small, each instance of these instructions touches a single data element. As a result, whenever multiple copies of the same structure are made, RAR dependences exist between the loads of subsequent calls to memcpy. Similarly, whenever a structure is copied in a chain fashion (i.e., one copy is made, then that copy is used to create another one and so on) RAW dependences exist between the stores of the previous call to memcpy and the loads of the current call. Function ChkGetChunk has a parameter that is an index to an array of object pointers. At entry point this index is used to de-reference the corresponding object and a different action is taken depending on the data found. Whenever this function is called for the same object, RAR dependences exist between the loads that follow the object pointers and access the object's elements. It should be noted that register allocating the corresponding data is not straightforward.

So far we have focused on the fraction of loads that do get a correct value via cloaking. However, since we use prediction, it is to be expected that some loads may get an incorrect value. Ultimately, the impact cloaking will have on performance will be determined by the latency reduction obtained when cloaking is successful and by the penalty paid to undo the effects of erroneous speculation. The two factors represent a trade-off. While on one side we would like to use a predictor that has the highest possible coverage, when performance is considered, care must be taken to maintain the net penalty of mispeculation relatively low. Figure 4.11 reports the mispeculation rates for the cloaking mechanism used in this section. It can be seen that mispeculation rates in most cases are high. For 107.mgrid and 132.ijpeg the mispeculation rate even exceeds cloaking coverage. It can also be seen that most of the mispeculations correspond to RAR dependences. Moreover, mispeculations typically increase with the size of the DDT. There two reasons why: (1) more dependences are exposed which are not necessarily regular, and (2) in the case of RAR dependences, using a larger DDT often results in marking different loads as producers of values for cloaking purposes. In the latter case, the dependences formed are not necessarily as regular as the ones formed with a smaller DDT.

Overall, the high mispeculation rate observed is the result of the non-adaptive nature of the predictor used. In the next section we consider alternative, adaptive predictors that aim at reducing the mispeculation rate while maintain the benefits of high cloaking coverage.

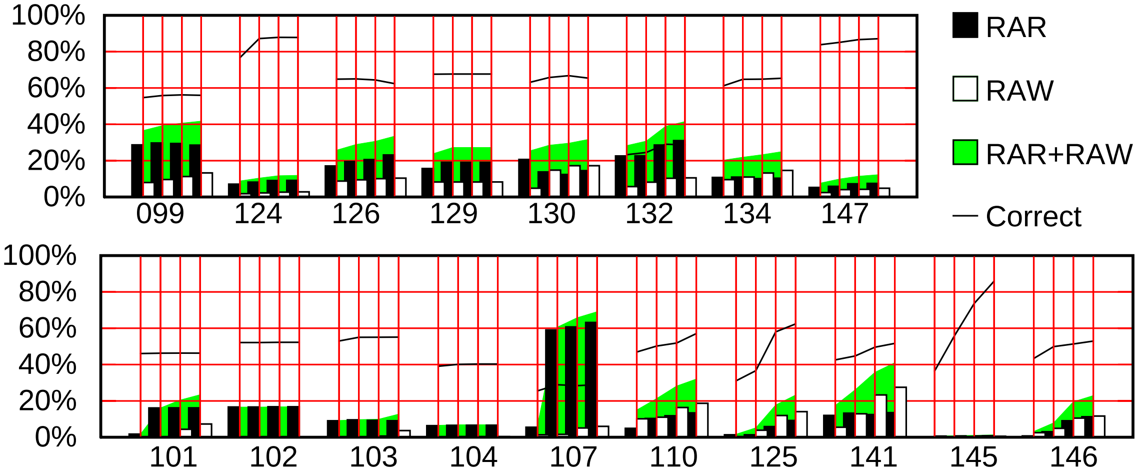


Figure 4.11: *Cloaking mispeculation rates for the non-adaptive dependence predictor as a function of detection table size. Range shown is 64 to 512 entries. Grey area represents the total mispeculation rate. The black and white bars represent the mispeculations that correspond to RAR and RAW dependences respectively. Also shown is the percentage of correctly predicted loads (black line).*

4.6.3 Using Adaptive Predictors to Improve Cloaking Accuracy

In the previous section we used a non-adaptive dependence predictor to obtain a rough estimate on the coverage we can hope to obtain with cloaking and bypassing. As expected we have seen that a non-adaptive predictor results in relatively high mispeculation rates. In this section we consider a number of alternative adaptive predictors that aim at maintaining high cloaking coverage while reducing the mispeculation rate. As we have seen in the previous section, a significant fraction of mispeculations correspond to dependences with non-unit instance distances. Another source of mispeculations is loads which have multiple dependences that exhibit different levels of regularity. To filter out these cases we considered a number of adaptive predictors including the commonly used 1 and 2-bit saturating counter-based predictors. These predictors operate as follows: A threshold value is set to decide whether cloaking should be performed or not. Whenever cloaking is not performed, the counter is incremented (bias toward cloaking being performed) when a dependence is detected. Whenever cloaking is performed the counter is adjusted to reflect whether the value obtained was correct or not. While these predictors perform better than the non-adaptive one, they suffer from oscillations on those loads that exhibit detectable dependences that cannot be handled by the cloaking mechanism in use. On these loads, the predictors oscillate between having cloaking being performed, determining that the value so obtained was incorrect, moving back to a state where cloaking is not performed and once a dependence is detected moving back to the state where cloaking is again performed. One solution to this problem is to decrement the counter by a higher amount whenever cloaking is erroneous or to use a relatively high threshold value (both approaches were suggested by Tyson and Austin in [89]). However, either approach results in a relatively high decrease in cloaking coverage. To filter out these loads while maintaining high coverage we experimented with the predictor shown in Figure 4.12. This predictor differs from the counter based ones in that cloaking is always performed when the predictor is in a state different than the initial state. However, the predicted value is propagated to the consumers of the load only if the predictor is in the “Use” state.

Figure 4.13, reports the cloaking coverage (fraction of all executed loads that get a correct value via cloaking) obtained with a number of different predictors. In these experiments we have used a DDT of 128 entries since, as we have seen in the previous section, cloaking coverage remains relatively unchanged when

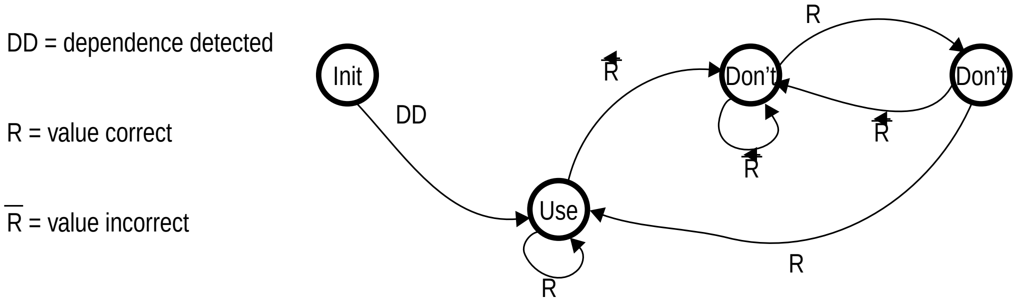


Figure 4.12: *An adaptive cloaking predictor.*

larger detection tables are considered. We report results for the following four predictors: 1-bit counter, 2-bit counter with threshold of 1, 2-bit counter with a threshold of 2 and the 2-bit predictor of Figure 4.12 (shown in that order starting from the second bar from left). To aid in the interpretation of these results we also include the 1-bit non-adaptive predictor we used in the previous section (left-most bar). As it can be seen, little variation in cloaking coverage is observed with all the adaptive predictors shown. Virtually no variation is observed for the floating point programs. Some variation is observed for the integer programs with the decrease in cloaking coverage being noticeable only for 099.go, 130.li and 126.gcc. This observation suggests that some dependences in these programs exhibit transient behavior. For example, some dependences either do not occur every time the corresponding instructions are encountered or exhibit variance in their instance distances. The adaptive predictors filter out these dependences and this results in a decreased cloaking coverage. However, it should be expected that for the same reason we should also observe a decrease in mispeculation rates. In most programs, the 2-bit counter based predictors perform slightly better than the adaptive predictor of Figure 4.12. However, the difference is relatively small and for two programs, 147.vortex and 125.turb3d the latter predictor performs better. Moreover, as we will show next, the predictor of Figure 4.12 offers significantly lower mispeculation rates compared to the counter-based predictors. These results suggest that any of the predictors studied may be used to meet one of our two goals, that of maintaining a relatively high cloaking coverage.

Figure 4.14 shows the mispeculation rates for the same predictors. The grey bar reports the overall mispeculation rate, while the black and white bars report the mispeculations that correspond to RAR and RAW dependences, respectively. We use a logarithmic scale for the Y-axis as the mispeculation range is rather large (the mispeculation rates observed with the last predictor can be seen in linear scale in Figure 4.15). As it can be seen, even though the counter based predictors do reduce the mispeculation rate compared to the non-adaptive 1-bit predictor, the reduction is more pronounced when the predictor of Figure 4.12 is used. For most programs the mispeculation rate drops below 1% (for 145.fpppp it drops to 0.03% — not visible in the figure). The only programs for which the mispeculation rate is above 1% are 099.go, 124.m88ksim, 130.li, 126.gcc and 110.applu. Some of these mispeculations correspond to dependences that are transient. Another source of mispeculations are RAW dependences on callee-saved registers of recursive functions. For example, roughly 1% (absolute fraction over all loads) of the mispeculations in gcc correspond to callee-saved register save-restore traffic from functions rtx_cost and canon_reg.

We can also observe that for the integer codes, RAR mispeculations are frequent and in some cases even more frequent than RAW dependences. However, we have seen (Figure 4.13) that most of the correct predictions in integer codes correspond to RAW dependences. This observation suggests that RAW dependences are more regular than RAR ones for these programs. It also provides us with a straightforward way of trading some of the cloaking coverage for a more than half reduction in mispeculation rates. Depending on the relative cost of mispeculations it may be advantageous to not include RAR dependences in the cloaking mechanism. For the floating point programs, RAR dependences are either the sole source of mispecula-

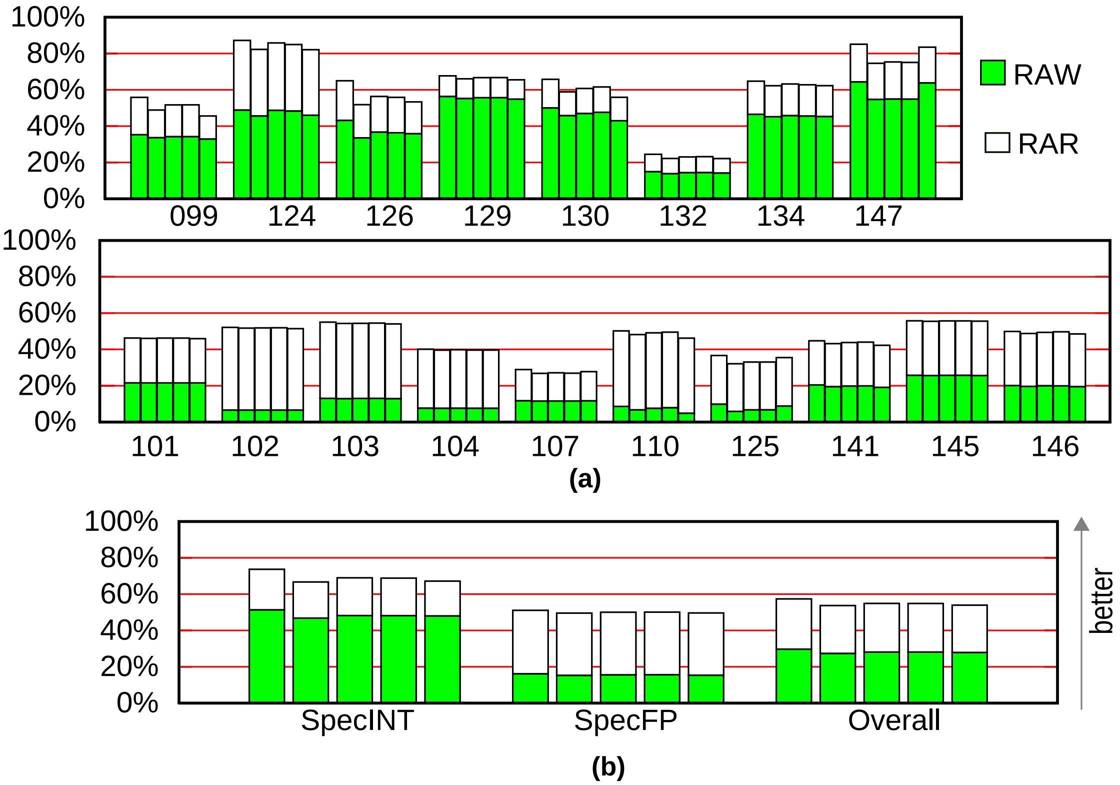


Figure 4.13: *Cloaking coverage with various predictors. Percentages are reported over all loads. (a) Per program results. (b) Averaged results. Five predictors are shown. From left to right these are: 1-bit non-adaptive, 1-bit adaptive, 2-bit saturating counter with threshold 1, 2-bit saturating counter with threshold 2 and the predictor shown in Figure 4.12.*

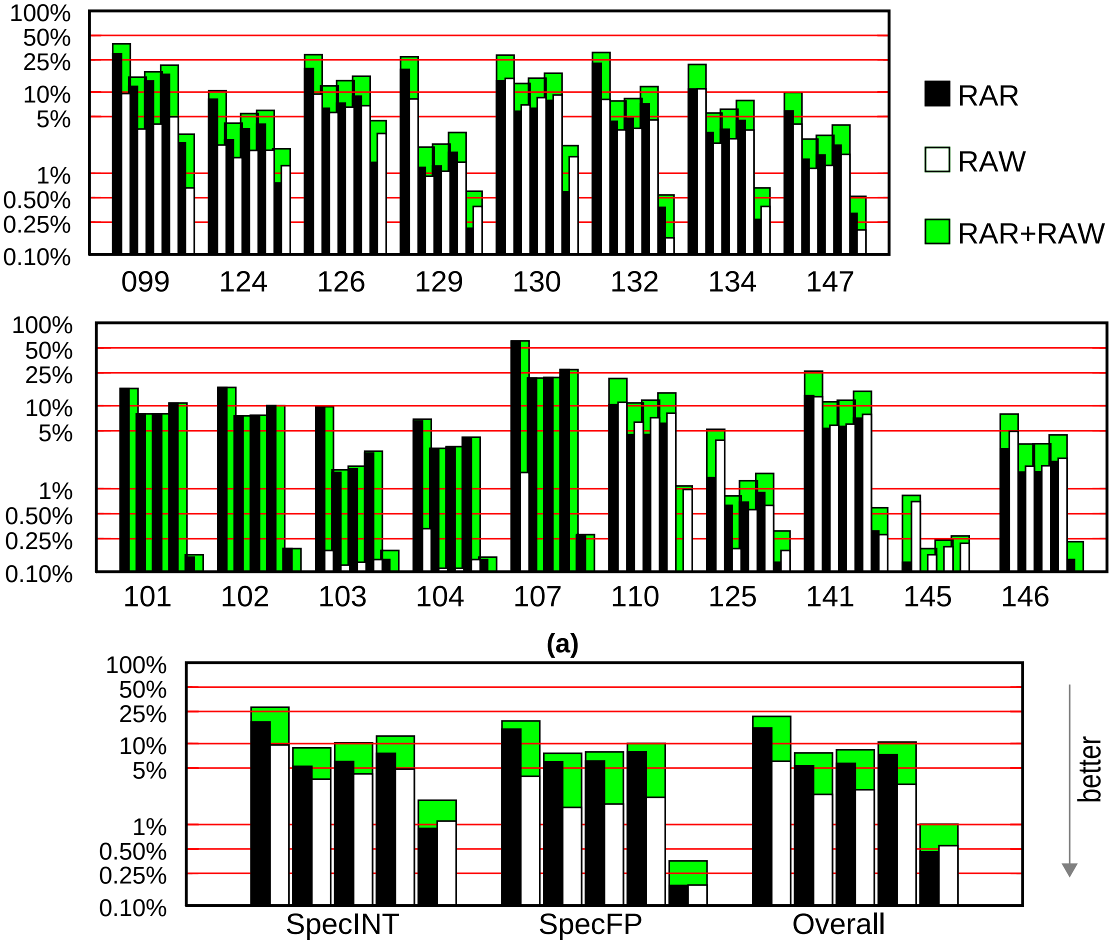
tions or they cause as many misspeculations as RAW dependences do. However, it should be noted that for most floating point programs RAR dependences are also responsible for most of the loads that are correctly communicated (cloaking coverage, as seen in Figure 4.13).

When all programs are considered, the predictor of Figure 4.12 reduces mispeculations by almost an order of magnitude when compared to the other predictors. The mispeculation rate is 2%, 0.35% and 1.01% for the integer, floating and all program respectively. From that, 1.1%, 0.17% and 0.54% (percentage of loads) corresponds to RAW dependences.

Since the predictor of Figure 4.12 drastically reduces the observed mispeculation rate with little or no reduction in cloaking coverage, we will focus on cloaking mechanisms that use this predictor in the rest of the evaluation.

4.6.4 Characteristics of the Memory Values that are Handled by Cloaking

With a cloaking mechanism that offers relatively high coverage and low mispeculation rates, it is now interesting to attempt to gain an understanding of what kind of loads are correctly handled by this mechanism. We first present a breakdown of loads that get values from cloaking in terms of the address space and the base register used (Sections 4.6.4.1 and 4.6.4.2 respectively). We do so as an obvious alternative to



(a)

(b)

Figure 4.14: *Cloaking misspeculation rates for various predictors. (a) Per program results. (b) Averaged results. The same predictors as in Figure 4.13 are used. The Y-axis is logarithmic.*

cloaking is register allocation where a value is placed into a register so that instructions can reference it directly. The address space and base register breakdowns provide indications on why the particular values were not register allocated or whether they could be allocated if more registers were available (values accessed via the global pointer should be easy to register allocate). The next characteristic we measure is address locality (Section 4.6.4.3). Address locality provides an indication on whether the particular load tends to accesses the same memory location over time. We do so as an alternative to cloaking would be to attempt to directly predict the addresses that loads and stores access and use this prediction to streamline access to those locations. The address locality results provide an additional indication on whether such an approach may offer better coverage. We then measure the value locality of loads handled by the cloaking mechanism and the coverage obtained through a last-value load value prediction mechanism (Section 4.6.4.4). As we explained in the related work section, value prediction may also be used to allow loads to obtain their value early. We also measure the dynamic instruction distance among the source and sink instructions that cloaking uses (Section 4.6.4.5). These results provide an upper bound on the fraction of loads that can be handled with bypassing and provide an indication of the effective scope of a realistic cloaking mechanism. Finally, we study how cloaking accuracy changes if a different input data set is used (Section 4.6.4.6). Our goal is to provide an indication that memory dependence behavior is mostly a property of the program and not of the data processed by the program.

4.6.4.1 *Address Space Breakdown*

In this section we present a breakdown of the memory traffic handled by cloaking in terms of the address space through which the values are communicated. Figure 4.15 shows a breakdown of loads that get a value from cloaking in terms of the address segment that the load accesses. Part (a) shows those loads that get a correct value while, part (b) shows those loads that get an incorrect value. Note that in part (b), the Y-axis range is different for the integer and the floating programs. It can be seen that while most of the values belong to the data and stack segments, a noticeable fraction of heap values are also correctly handled by cloaking. (Note that the few heap accesses observed for 103.su2cor and 104.hydro2d are mostly the result of the implementation of the fortran built-in functions which are translated to C in our environment.) While heap values contribute significantly to cloaking coverage for some programs, they are also responsible for a disproportionate fraction of the mispeculation rate.

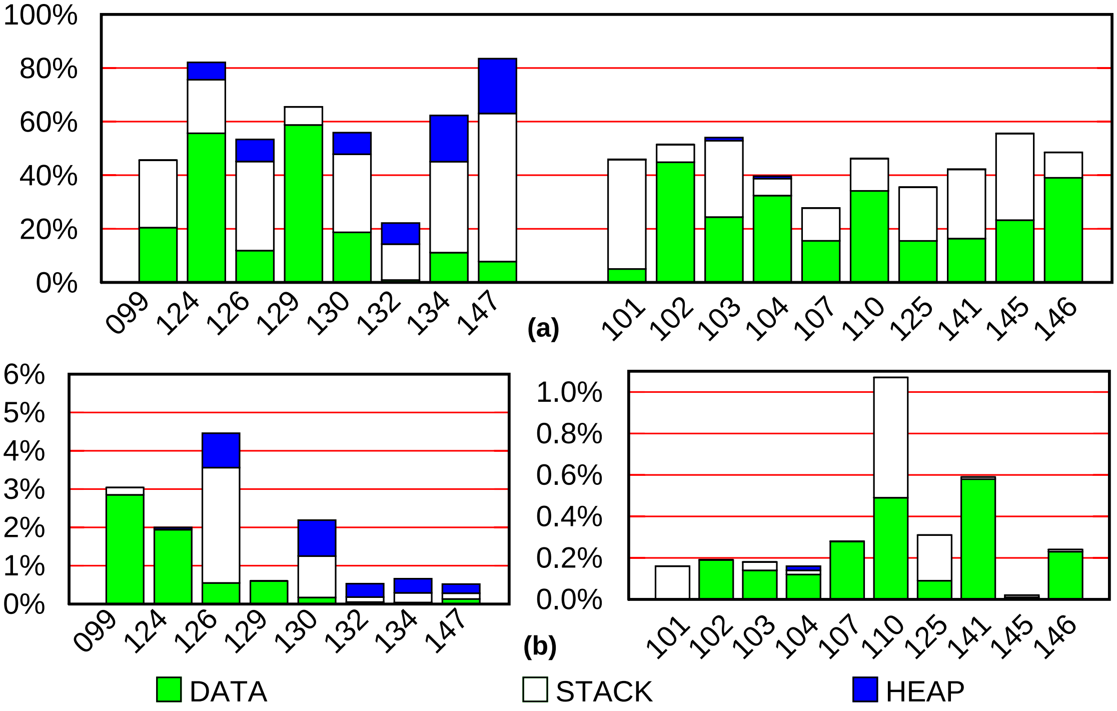


Figure 4.15: *Breakdown of loads that get a value via cloaking in terms of the address segment that is accessed. (a) Loads that get a correct value via cloaking. (b) Loads that get an incorrect value via cloaking. Y-axis range varies per graph.*

4.6.4.2 *Base Register Breakdown*

Figure 4.16 presents a breakdown of cloaking communication in terms of the base register used by the corresponding loads. Loads are separated into three classes based on whether they use the stack pointer, global pointer or any other register as their base register (shown from left to right in that order). Part (a) shows the loads that get a correct value, while part (b) shows the loads that get an incorrect value. It can be seen that for most of the integer programs, loads that use the stack pointer directly or a register other than the global pointer are responsible for most of the correctly communicated values. For the floating point programs most of the correctly communicated loads do not use the stack or the global pointer loads. The only program where global pointer loads are responsible for most of the correctly communicated values is 129.compress. Most (but not all) of these accesses correspond to global variables that are not register allocated and that are accessed repeatedly from within loops. Register pressure is one reason why these variables are not register allocated. In this case, providing a larger register set would help in removing these loads and stores. However, about 10% of all executed loads get a correct value from cloaking and do not use the global pointer. This observation suggests that even if the global pointer loads were eliminated cloaking may still be of use as it would cover about 25% of the remaining loads (global pointer loads account for about 60% of all loads).

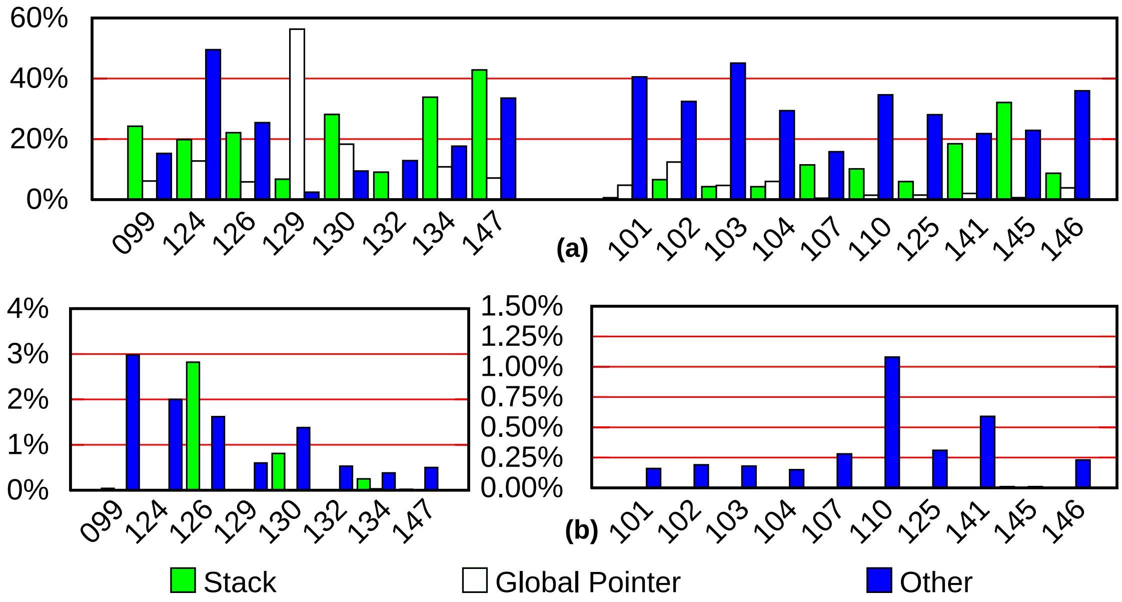


Figure 4.16: *Load breakdown in terms of the base register used. From left to right the categories are: sp (stack pointer), gp (global pointer) and other. (a) Correctly communicated loads. (b) Incorrectly communicated loads.*

A clear trend is demonstrated by mispeculations as shown in Figure 4.15. As it can be seen from part (b), virtually all of the mispeculated loads use a register other than the stack or global pointers. Exceptions are 126.gcc, 130.li and 134.perl where stack loads are also a significant source of mispeculations. The stack loads that are mispeculated in these three programs correspond, in their vast majority, to callee-saved register restore loads found in recursive functions. These loads experience RAW dependences with the save register store instructions found at entry to the corresponding recursive function. These loads are communicated correctly when recursion terminates (i.e., at the leafs of the calling-graph), while they may get incorrect values when further recursive calls are made.

The results of Figure 4.16 demonstrate that stack and global pointer loads exhibit more stable dependence behavior than loads that use other registers. Even so, the mispeculations that correspond to the latter loads are relatively few while their contribution to correctly communicated values is by comparison relatively high. Depending on the penalty of mispeculation we could avoid using cloaking for loads that do not use the stack or global pointers in order to minimize mispeculations (for most programs mispeculations will be virtually non-existent). (Tyson and Austin suggest such a confidence prediction in [89].) However, such an approach will also lead to a significant reduction in the fraction of loads that get a correct value via cloaking.

4.6.4.3 Address Locality Measurements

We next measure the address locality of the loads that get a correct value via cloaking. We define *address locality* as the probability that a load instruction accesses the same address as it did the last time it was encountered. We present these address locality measurements not only because they offer additional insight on the type of loads that are correctly handled by cloaking but also because they provide an indication on whether an address prediction based memory communication streamlining scheme would offer significant advantages over the cloaking scheme we are investigating. The results are shown in Figure 4.17. The left bar represents the fraction of all loads that exhibit locality while the right bar represents the fraction of loads that get a correct value via cloaking. To aid in the interpretation of these measurements we breakdown the loads that exhibit address locality into three categories depending on whether they also experience a RAW, RAR or no dependence. We can observe that while many of the loads that cloaking captures also exhibit address locality, a significant fraction of them does not exhibit address locality. We can also observe that with the exception of 145.fpppp, there are very few loads that exhibit address locality and do not also experience a dependence. (145.fpppp uses many statically allocated variables of aggregate type that are used extensively in repeated calculations. The vast majority of these loads experience dependences. These dependences however escape detection as the detection table used in this experiment is relatively small. Had we used a larger detection table, most of these loads would have been captured by the cloaking mechanism also (see Section 4.6.2).) Moreover, it is often the case that cloaking captures loads that do not exhibit address locality. These results, suggest that cloaking may offer better coverage than a straightforward last-address prediction scheme while not requiring explicit tracking of the addresses that are accessed by loads and stores.

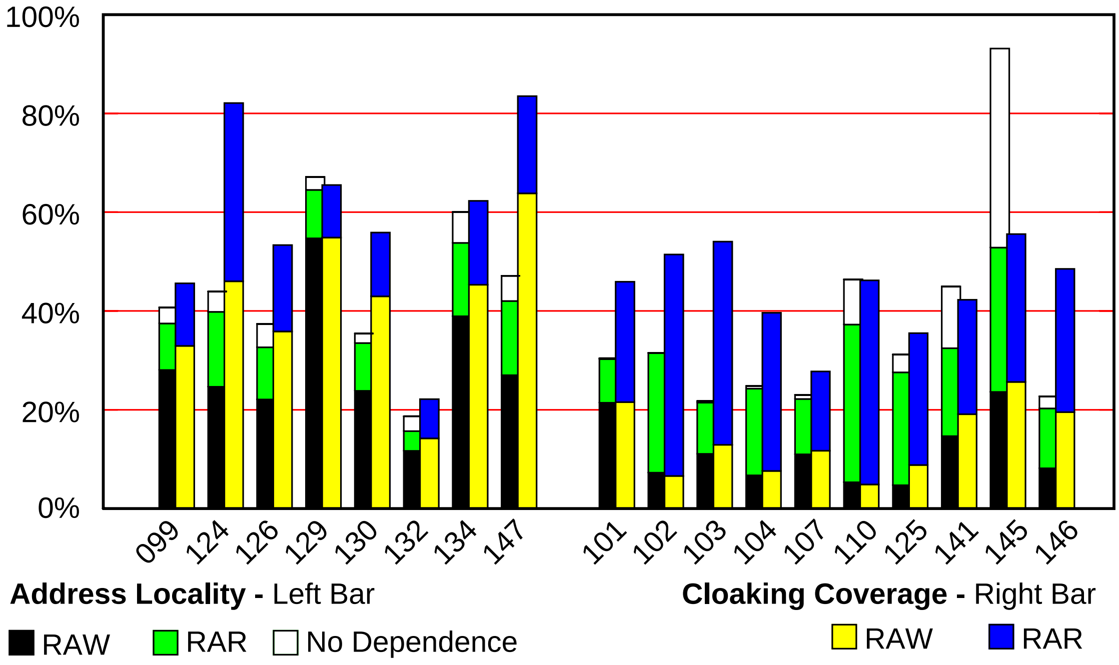


Figure 4.17: *Address Locality breakdown. Left bar reports address locality. Right bar reports cloaking coverage.*

4.6.4.4 Value Locality and Value Prediction Measurements

In this section we measure the value locality exhibited by the loads that get a value from the cloaking mechanism. Value locality and how it is distributed among loads that have dependences is an interesting metric in this context, as value prediction can also be used to allow loads to obtain their value early, possibly earlier than cloaking would allow. Figure 4.18 shows a breakdown of loads that exhibit value locality alongside with a breakdown of loads that get a correct value via cloaking. The loads that exhibit value locality are separated into three categories depending on whether they have a dependence detected and what type this dependence is. The categories are: RAW, RAR and no-dependence. We can observe that for most programs cloaking coverage is higher than the percentage of loads that exhibit value locality. More loads exhibit value locality only for 132.ijpeg, 104.hydro2d, 110.applu and 125.turb3d. When the dependence status of loads is taken into consideration, we can observe that cloaking covers more of the loads that experience dependences. This phenomenon is more pronounced for those loads that experience RAW dependences where cloaking coverage is sometimes twice the fraction of loads that exhibit value locality. However, we can also observe that a noticeable fraction of loads with value locality do get a correct value via cloaking. This observation suggests a potential synergy of the two techniques. We do investigate this possibility. Before doing so however, we take a look at a real load value predictor.

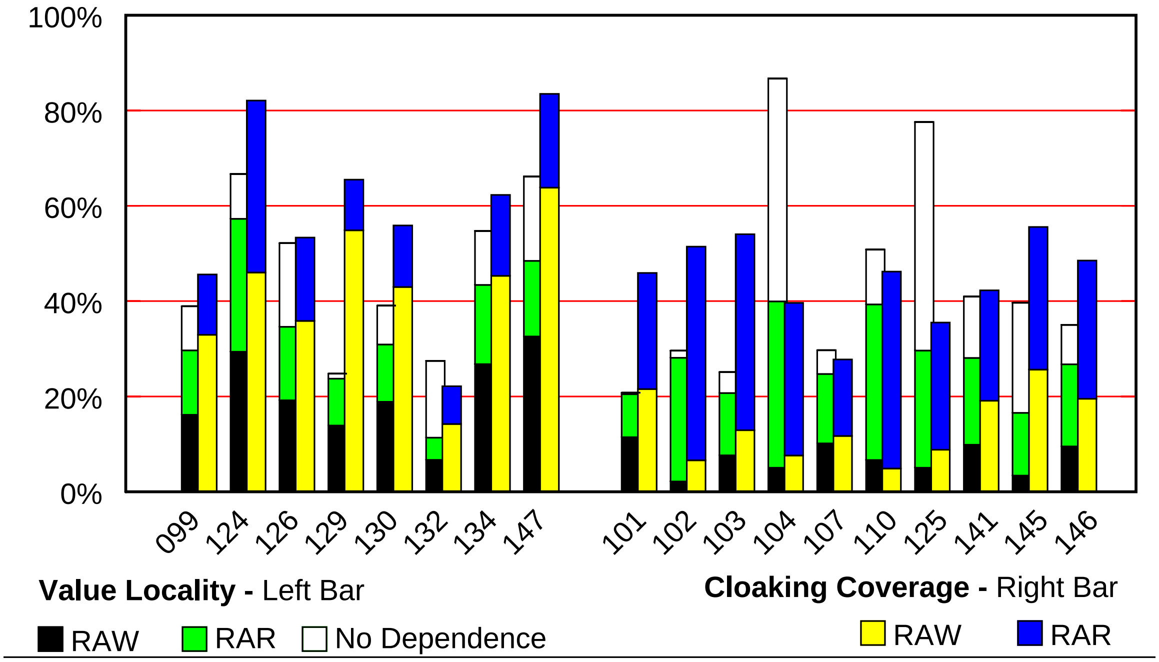


Figure 4.18: *Value Locality breakdown. Left bar: loads that exhibit value locality separated into those that have a RAW, a RAR or no dependence. Right bar: loads that get a correct value via cloaking separated into those that have a RAW or a RAR dependence.*

Value locality provides just an indication of the fraction of the loads that may get a correct value via a value predictor. It directly measures those loads that would get a correct value from a value predictor which has infinite entries and uses a simple last-value with no hysteresis predictor (see [74]). To better understand how cloaking compares with value prediction, we next compare a finite value predictor with a cloaking mechanism that uses finite prediction and detection structures. For this experiment we simulate a value predictor with 16K entries that uses a last-value prediction scheme augmented with a per load confidence predictor. The confidence predictor we use is the one presented in Figure 4.12. The cloaking mechanism we use has a 16K DPNT, an 128-entry DDT and a 2K synonym file. All structures are assumed to be fully-associative. Figure 4.19 reports the coverage (part (a)) and mispeculation rates (part (b)) observed with these two mechanisms. The value prediction results are shown by the black left bar, while the cloaking prediction results are shown by the right grey bar. As it can be seen from the results of part (a), the cloaking mechanism offers better coverage than the value prediction mechanism for most programs. Value prediction offers better coverage for three floating point programs: 104.hydro2d, 125.turb3d and 110.applu. Similar trends are seen in the mispeculations observed. As it can be seen in part (b), the cloaking mechanism experiences less mispeculations than the value prediction mechanism for all programs except 129.compress and 146.wave5. In those two programs the mispeculation rates are relatively small and furthermore, cloaking exhibits significantly higher coverage than value prediction does.

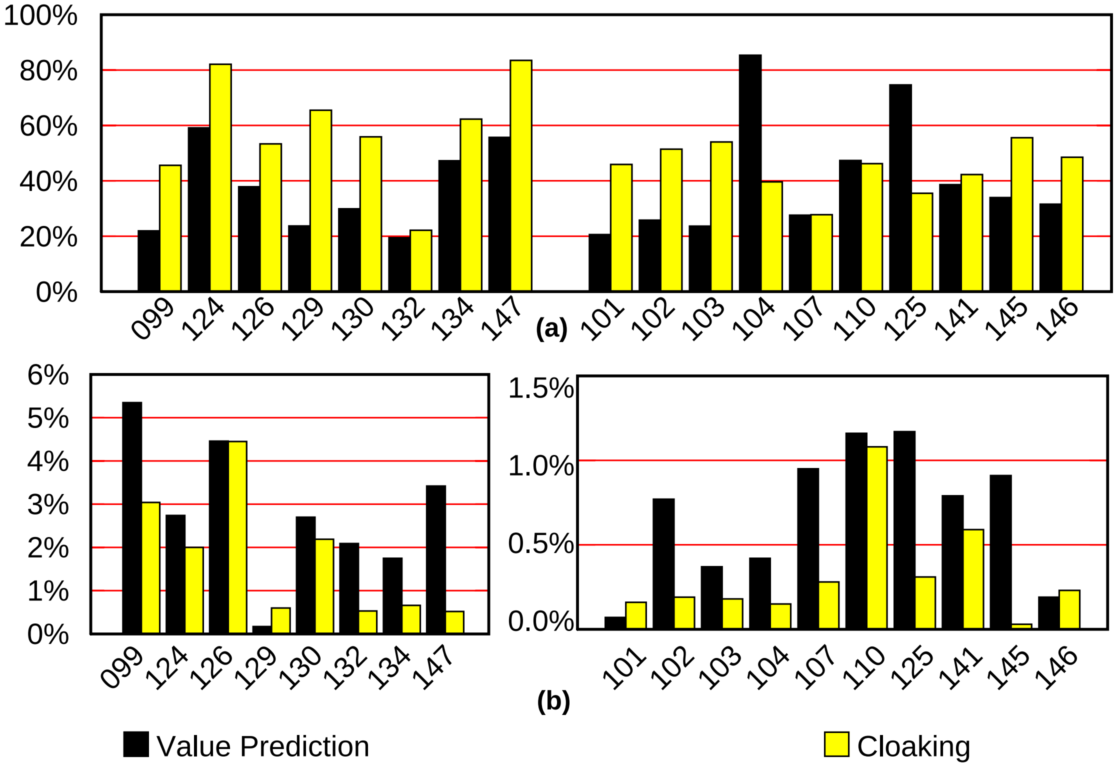


Figure 4.19: *Comparing a value predictor with cloaking. (a) Coverage: loads that get a correct value. (b) Mispeculation rates: loads that get an incorrect value.*

Even though these results suggest that cloaking offers better prediction than value prediction (for the specific predictors) it should be noted that these results should be used simply as an indication. Not only different or larger predictors may improve value prediction (and cloaking) but also, the performance impact of each technique can only be judged when timing is taken into account. Even so, these results suggest that cloaking provides a way of obtain load values early for a significant fraction of loads that do not exhibit value locality (mostly loads that have RAW dependences). This observation hints to a potential synergy between the two techniques. To better understand how value prediction and cloaking/bypassing relate, we measured the fraction of loads that get a correct value from cloaking/bypassing but not from value prediction and vice versa. The results are shown in Table 4.1. To aid in the interpretation of these results, we also present a breakdown of the values obtained via cloaking/bypassing in terms of the dependence type. We can observe that indeed for most programs, value prediction captures some loads that cloaking/bypassing does not and vice versa. For most programs, the fraction of loads correctly predicted only via cloaking/bypassing is higher than the fraction of loads correctly predicted only via the value predictor. The two exceptions are 104.hydro2d and 125.turb3d. Most of the loads captured only by cloaking get a value via a RAW dependence. However, a noticeable fraction of the loads correctly predicted only via cloaking/bypassing get a value via a RAR dependence. For some programs (most floating point programs and 124.m88ksim and 132.ijpeg), this fraction is close to or exceeds the fraction of loads that get a value via a RAW dependence. Motivated by the observations of this section, in Section 4.6.6 we investigate possible combinations of value prediction and cloaking/bypassing and find that the two techniques can be used to complement each other, leading to further performance improvements.

	Cloaking/Bypassing			**VP**		**Cloaking/Bypassing**			**VP**
	RAW	**RAR**	**Total**			**RAW**	**RAR**	**Total**	
099	23.43%	5.75%	29.18%	5.29%	***101***	10.22%	15.35%	25.58%	0.24%
124	14.23%	10.62%	24.85%	1.88%	***102***	6.43%	19.98%	26.41%	0.37%
126	18.15%	5.89%	24.04%	8.01%	***103***	7.18%	25.89%	33.08%	2.67%
129	41.18%	0.99%	42.18%	0.22%	***104***	3.02%	1.29%	4.31%	49.94%
130	31.08%	1.08%	32.17%	6.14%	***107***	2.34%	0.43%	2.77%	2.60%
132	8.67%	5.25%	13.93%	11.24%	***110***	3.18%	8.29%	11.46%	12.60%
134	21.72%	1.57%	23.29%	7.82%	***125***	2.27%	0.55%	2.82%	41.94%
147	29.52%	3.33%	32.85%	5.03%	***141***	8.85%	4.47%	13.34%	9.67%
					145	22.46%	17.87%	40.34%	18.17%
					146	10.08%	12.84%	22.92%	5.94%

Table 4.1: *Fraction of loads that get a correct value from cloaking/bypassing and not from a value predictor (“Cloaking/Bypassing” columns) and vice versa (“VP” columns).*

4.6.4.5 Dynamic Instruction Distance Distribution

In this section we measure the distance in dynamic instructions between the loads that get a correct value via cloaking and the source instruction that supplied that value. This information is useful in obtaining an upper bound on the fraction of loads that can also benefit from bypassing. As we noted in Section 4.3, bypassing can be applied only when the source store or load and the consuming load co-exist in the instruction window. Figure 4.1 reports the fraction of loads that get a correct value via cloaking as a function of dynamic instruction distance. The range shown is 4 to 16K instructions and samples are taken at distances that are powers of four. Part (a) reports the per program measurements while part (b) reports averaged measurements for the integer, floating point and all programs.

As it can be seen by the results of part (b), for 50% of all correctly communicated loads the source of the value is within 64 instructions. This result provides an upper bound on the fraction of loads that could also benefit from bypassing in a 64-instruction window processor. This percentage rises to roughly 75% when we consider distances of up to 256 instructions. It can also be seen that about 15% of all correct values correspond to dependences that span more than 1K instructions. For some programs (e.g., 129.compress and 107.mgrid) about 10% of all correct values correspond to dependences that even exceed 16K instructions.

This result suggests that even the relatively small detection table we used is capable of capturing memory communication that spans large regions of the dynamic instruction stream.

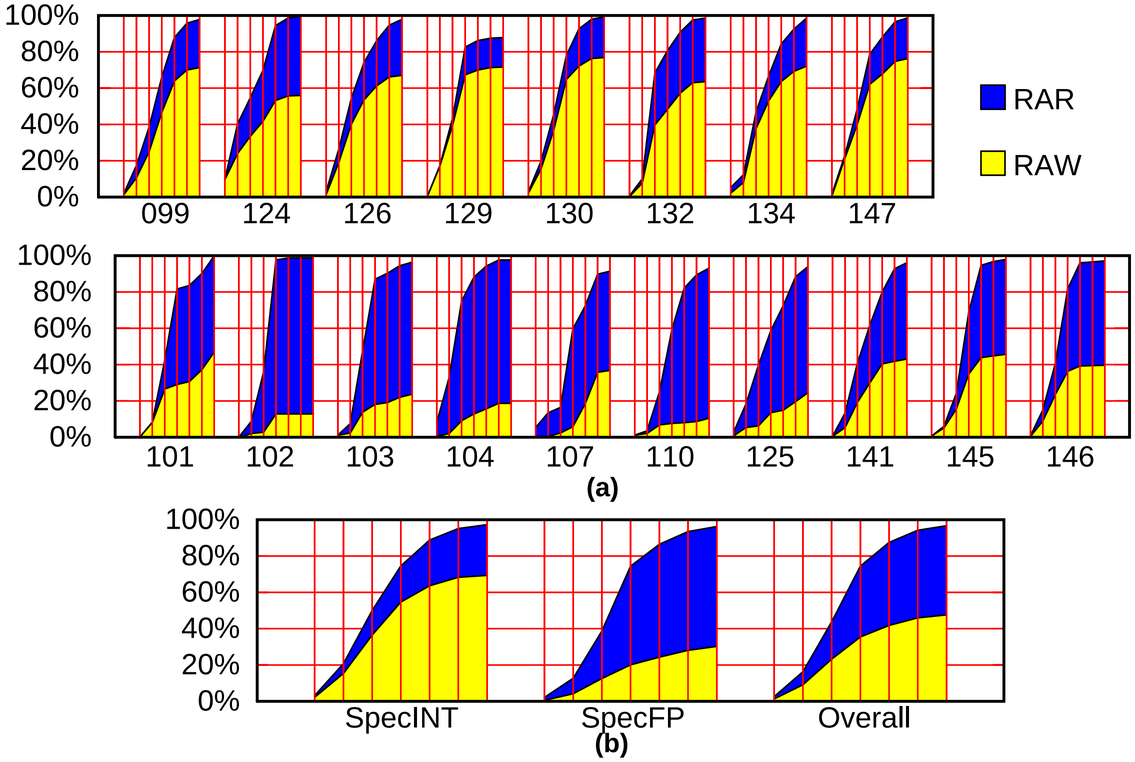


Figure 4.1: *Cumulative dynamic instruction distance distribution between source instruction and loads that get a correct value via cloaking. Range is 4 to 16K instructions. Samples are taken at the following distances: 4, 16, 64, 256, 1K, 4K and 16K (powers of 4). (a) Per program measurements. (b) Averaged measurements.*

4.6.4.6 Input Data Set Sensitivity Analysis

In this section we investigate the sensitivity of cloaking to a change of the input data set and demonstrate that, for most of the programs studied, cloaking coverage and mispeculation rates do not change significantly. For this experiment we use the a cloaking mechanism with an 128-entry DDT and infinite prediction structures and measure the cloaking coverage and mispeculation rates observed with the second input data set detailed in Chapter 1. For most programs, this data set produces significantly longer instruction counts. Figure 4.2 reports the results of this experiment. Part (a) reports cloaking coverage while part (b) reports mispeculation rates. Two bars are shown per program. The left dark bar corresponds to the alternate data set while the right light gray bar corresponds to the data set we have been using so far. Focusing first on the coverage results, it can be observed that little variation is experienced when the alternate data set is used. Only, 146.wave5 demonstrates a relatively large difference as its coverage increases by 16% when the alternate input data set is used (since mispeculations also decrease, any performance results obtained for 146.wave5 with the default data set will probably be pessimistic). A similar trend is exhibited by mispeculation rates (part (b)). However, the variation here is greater. For example, the mispeculation rate for 102.swim rises from 0.24% to roughly 0.9% while it drops from about 2% to 0.8% for 124.m88ksim. While large, when viewed relatively, these variations are small in absolute terms. Overall, even though some variation is observed in cloaking coverage and mispeculation rates, for most programs this variation is relatively small. This result provides an indication that the dependence relationships captured by cloaking are mostly data independent for these programs.

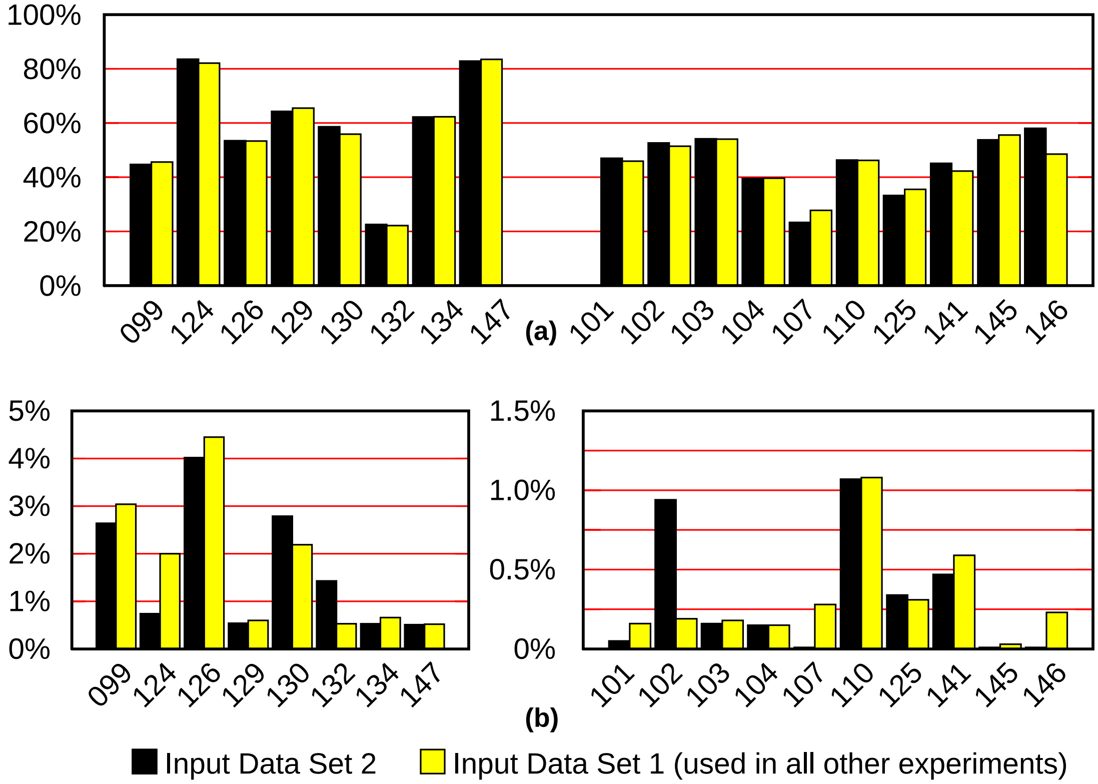


Figure 4.2: Comparing cloaking accuracy with a different input data set. Left bar: alternate input data set (see Chapter 1, input data set 2). Right bar: default input data set. Part (a): cloaking coverage. Part (b): mispeculation rates.

4.6.5 Effects of Finite Prediction Structures

So far we have been assuming infinite prediction structures. In this section we study what effect finite prediction structures have on the accuracy of cloaking. We first investigate prediction and naming tables of various sizes and of varying associativity.

4.6.5.1 Sensitivity to the Number of DPNT Entries

Figure 4.3 reports the cloaking coverage (part (a)) and misprediction rates (part (b)) for prediction and naming tables that range from 256 to 8K entries with LRU replacement. Each entry consists of a 32-bit PC, a two-bit confidence automaton and a 16-bit synonym. While, it may be possible to reduce the number of bits required per entry (for example by recording partial PC information or by using shorter synonym fields) we do not investigate this possibility. In Figure 4.2, the right-most dark bar per program corresponds to the infinite prediction tables we have been using thus far. It can be seen that with few exceptions, little variation is exhibited even if we use a relatively small prediction and naming table of 256 entries. For virtually all programs a DPNT of 1K entries results in cloaking accuracy that is very close to that observed with the infinite table. Exceptions are 099.go, 126.gcc, 147.vortex and 145.fpppp which require an 8K-entry DPNT to virtually match the accuracy of an infinite prediction table. This phenomenon is explained by the relatively larger instruction working set exhibited by these four programs as compared to the rest of the programs we studied.

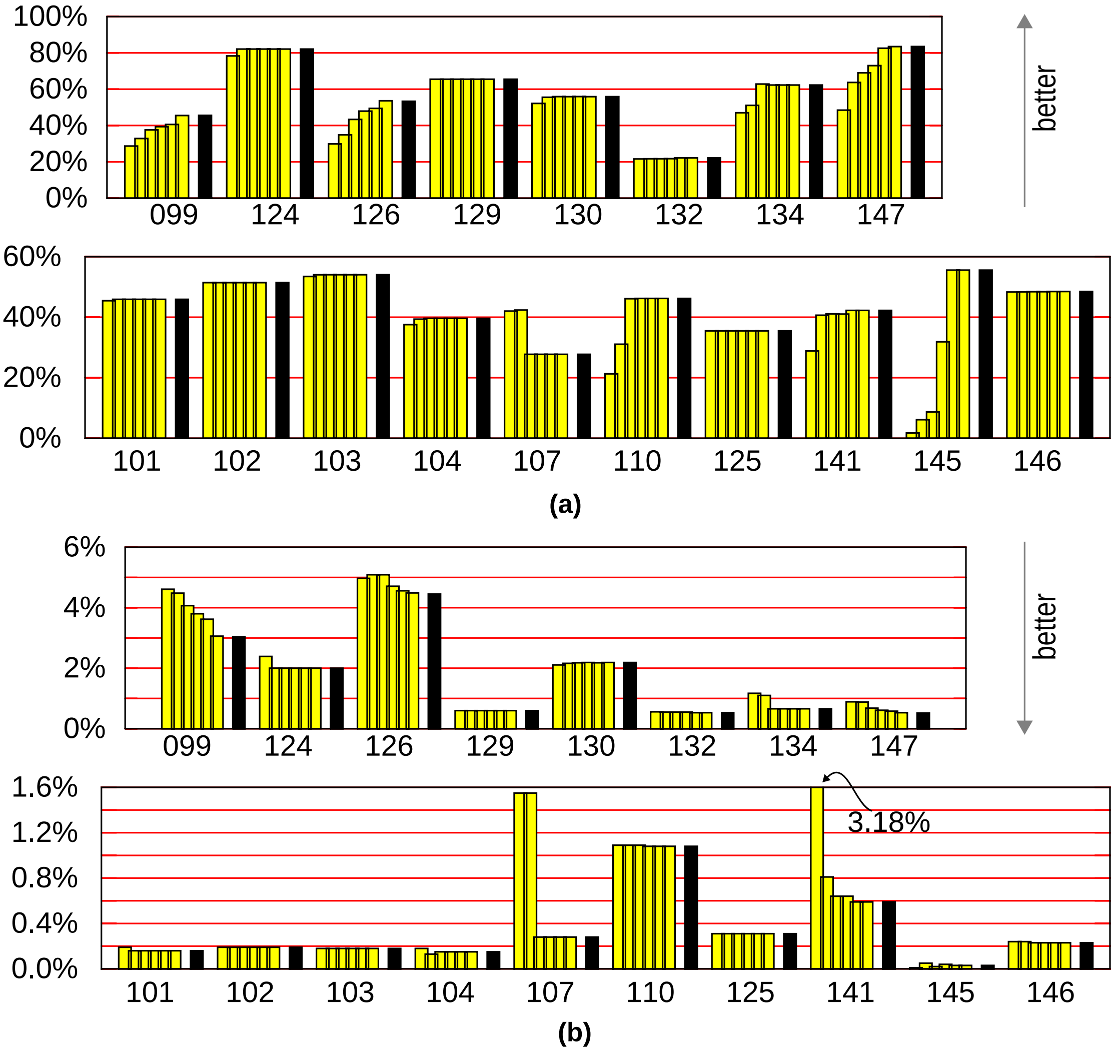


Figure 4.3: *Cloaking coverage and misspeculation rates as a function of the dependence prediction and naming table size. Part (a): cloaking coverage. Part (b): misspeculation rate. Range shown is 256 to 8K entries in steps that are powers of two. Last bar corresponds to an infinite table.*

Overall, an increase of the DPNT results in better cloaking characteristics. However, in some cases increasing the number of entries may reduce cloaking coverage and increase the mispeculation rate. For example, this is the case for 134.perl when we move from a table of 1K entries to one that has 2K entries. Generally, these negative effects are barely noticeable. Only in the case of 107.mgrid a significant decrease in cloaking coverage can be observed. Specifically, increasing the number of entries from 512 to 1K or more results in a 14% decrease in cloaking coverage (percentage of dynamic loads). This anomaly is observed on loads that have multiple RAR dependences which in the case of 107.mgrid, can be found mostly in the nested loops of the resid routine. The dependence behavior of these loads is similar to the behavior of the loads in the loop shown on page 110. As explained in Section 4.6.2, our greedy, non-adaptive approach to building dependence sets and marking producers and consumers will mark the first load as the producer and the other two loads as consumers for cloaking purposes. However, as execution progresses, the initial RAR dependences are not observed anymore and cloaking fails. If however, the first load gets evicted from the prediction table (which is more probable when the table is smaller) then the RAR dependence between the two loop loads may result in marking the load of line 3 as a producer and in the subsequent application of cloaking. This anomalous behavior may be eliminated by utilizing adaptive schemes in marking producers and consumers. For example, we could use a scheme that periodically flushes the prediction and naming table to achieve such an effect. However, given that for all programs except 107.mgrid this anomaly is barely noticeable and since our goal is to demonstrate the potential of cloaking and bypassing while understanding the issues involved, we do not investigate this issue further.

4.6.5.2 Sensitivity to the Associativity of the DPNT

Figure 4.4 shows how cloaking coverage and mispeculation rates vary as we change the DPNT's associativity from full to 1-way, 2-way, 4-way and 8-way. The DPNT used in this experiment is 8K entries. Reported is the *absolute* difference in the fraction of dynamic loads that get a value from cloaking (e.g., if coverage drops from 40% to 38% we report a 2% reduction). As it can be seen, for most programs, even a direct mapped table (1-way) results in similar cloaking accuracy as a fully-associative one. The only programs for which some differences are observed are 099.go, 126.gcc, 147.vortex and 145.fpppp. For those programs, associativity affects mostly cloaking coverage. While some variation in mispeculation rates is also observed, the absolute differences are relatively minimal (below 1%). These variations are caused by the larger instruction working set exhibited by these four programs. Even so, the differences in cloaking coverage are not extreme and drop to near 2%, for all programs except 145.fpppp when a 4-way set associative table is used.

4.6.5.3 Synonym File Size Sensitivity Analysis

We finally measure how cloaking accuracy varies as a function of the synonym file size. For this experiment we assume an 8K entry 2-way set associative DPNT and investigate synonym files with 32 to up 2K entries in steps that are powers of 2. We used fully-associative synonym files. Figure 4.5 reports the absolute difference in cloaking coverage compared to a synonym file of infinite size. It can be seen that most programs are relatively insensitive to variations in the synonym file capacity. An 1K synonym file yields cloaking accuracies that are virtually identical to a synonym file of infinite size. We have also experimented with the associativity of the synonym file. However, we omit these results as virtually no change was observed when we reduced the associativity of a synonym file with 1K entries.

4.6.6 Performance Impact

In this section we evaluate the performance impact of a combined cloaking and bypassing mechanism. We do so by simulating a dynamically scheduled ILP processor, measuring its performance with and without a cloaking/bypassing mechanism. The rest of this section is organized as follows. In Section 4.6.6.1 we describe the processor configuration and the cloaking/bypassing mechanism we simulated. In Sections 4.6.6.2 through 4.6.6.4 we report our findings. In Section 4.6.6.2 we measure how performance varies when cloaking/bypassing is used for various mispeculation handling models, and report: (1) the mispeculation rates observed, and (2) the fraction of loads that get a value via our mechanism along with a breakdown in terms of where that value came from (cloaking or bypassing). In Section 4.6.6.3, we compare a cloaking/bypassing and a last-value based load value prediction mechanisms. Finally, in Section 4.6.6.4 we consider two different combinations of cloaking/bypassing and last-value load value prediction.

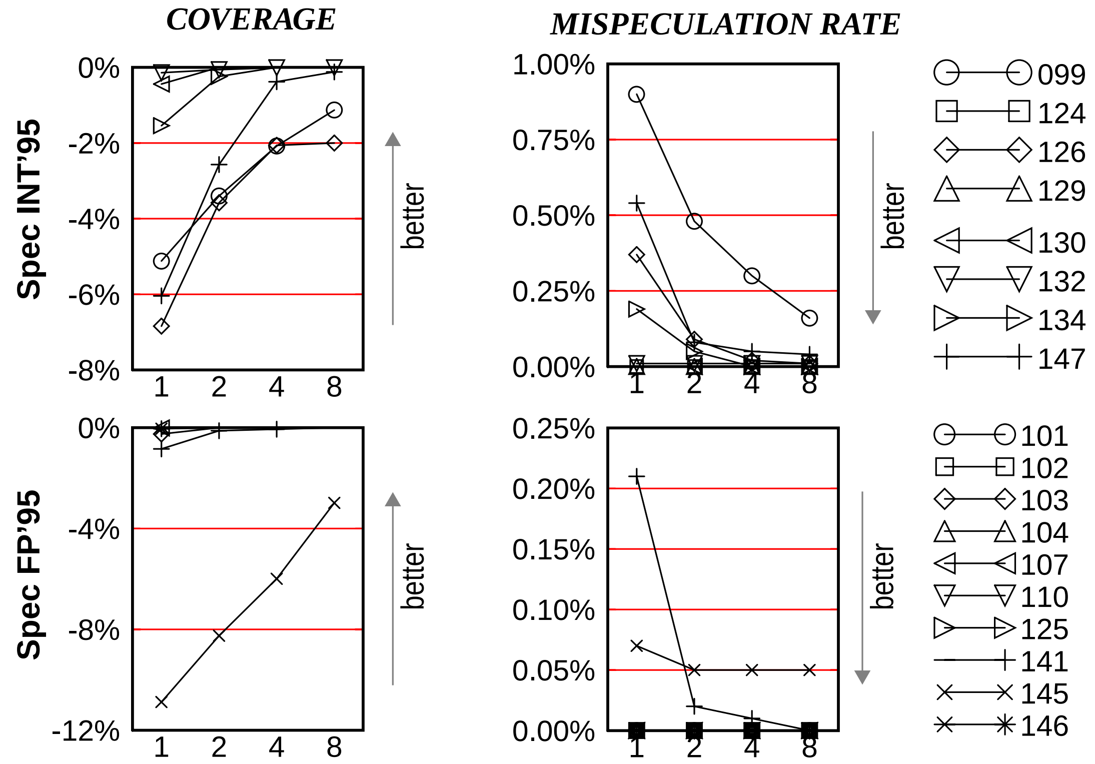


Figure 4.4: *Variations in cloaking accuracy as a function of the prediction and naming table associativity. Shown (Y axis) is the absolute difference (i.e., fraction of all loads executed) compared to a fully associative table of the same size. Number of entries is 8k and associativities simulated are: 1, 2, 4 and 8 (X axis). Note that the Y-axis range varies.*

4.6.6.1 Configuration Parameters

The base processor is capable of executing up to 8 instructions per cycle and is equipped with a 128-entry instruction window and a 128-entry load/store scheduler. It takes at least one cycle after a load has calculated its address to go through the load/store scheduler which implements *naive dependence speculation* (see Section 3.3). That is: (1) a load may access memory even when there are preceding stores that have yet to calculate their address, (2) a load will wait for preceding stores that are known to write to the same memory location (we have shown in Chapter 3 that dependence mispeculations rarely occur in such an environment). A detailed description of the rest of the configuration parameters was given in Chapter 1.

The cloaking/bypassing mechanism used comprises: (1) an 128-entry fully-associative DDT with word granularity (a single store can be recorded per word aligned address), (2) an 8K, 2-way set-associative DPNT, and finally, (3) an 1K, 2-way set associative synonym file. Figure 4.6 illustrates how the various components of the cloaking/bypassing mechanism are integrated in the processor's pipeline. Detection of dependences occurs when loads or stores commit by accessing the DDT. Synonym file updates and dependence prediction and naming table updates also occur at commit time. Dependence predictions are initiated as soon as instructions enter the decode stage. Loads and stores that are predicted as producers of synonyms associate the actual producer of the desired value with the predicted synonym by allocating an entry in the *synonym rename table* (SRT) for the purposes of speculative memory bypassing. That is, SRT entries associate synonyms with physical registers. Loads that are predicted as consumers of synonyms inspect both the

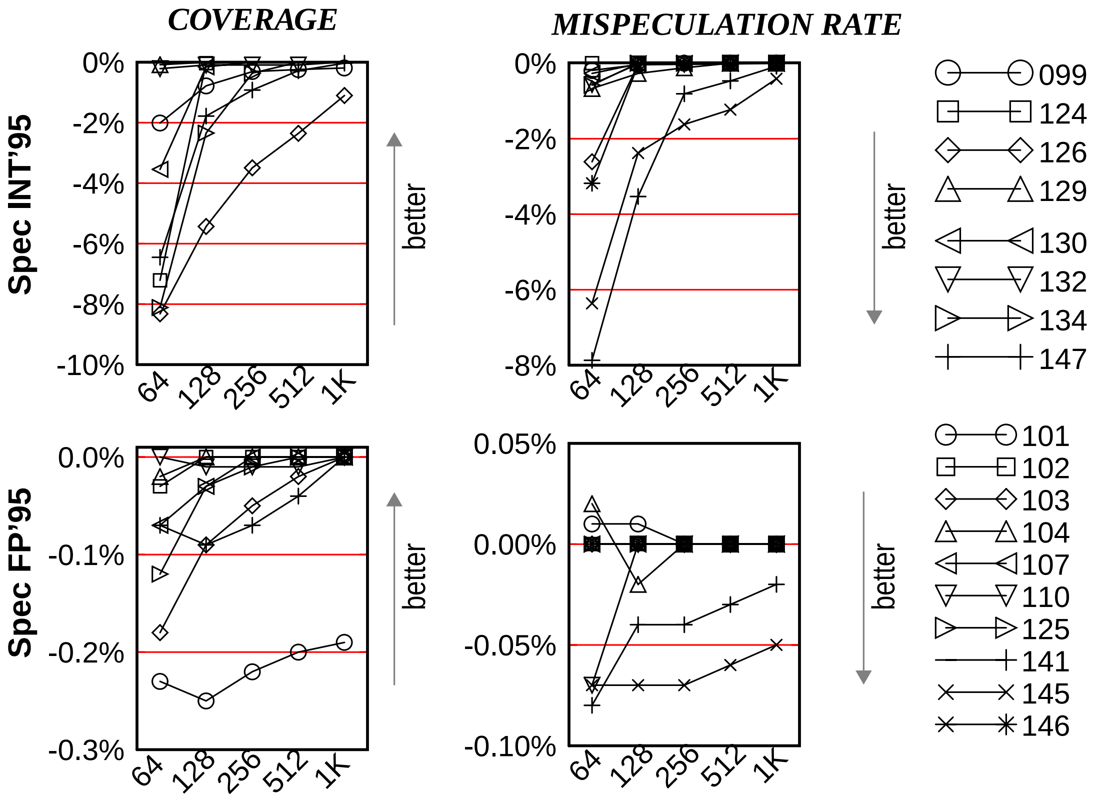


Figure 4.5: Cloaking coverage and mispeculation rates as a function of the synonym file size. Part (a): cloaking coverage. Part (b): mispeculation rate. Range shown is 32 to 2K entries in steps that are powers of two. Last bar corresponds to an infinite table. The prediction table used has 8K entries and is 4-way set associative.

SRT and the synonym file in parallel to determine the current location of the appropriate synonym. If an entry is found in the SRT, the synonym resides in the physical register file (or in a reservation station) as the corresponding load or store has yet to commit. Otherwise, the synonym is to be found in the synonym file. At most 8 predictions can be made per cycle and at most 8 instructions can be scheduled for cloaking or bypassing per cycle. Finally, no data type information is used for cloaking/bypassing purposes.

There are final piece of the cloaking/bypassing mechanism is responsible for: (1) verifying speculatively communicated values, and (2) recovering from mispeculations. For the purposes of this evaluation we have experimented with three mispeculation recovery mechanisms. The first is an oracle mechanism that avoids mispeculations completely. This is achieve by not speculating on a load value whenever that would result in a mispeculation. While this mechanism is impractical, its use allows us to obtain an upper bound on the performance benefits that might be possible for the given configuration. The second mechanism uses *selective invalidation* in order to re-execute only those instructions that used incorrect data. The selective invalidation mechanism we simulated works by: (1) first re-executing the instruction that was mispeculated, and (2) then sending the correct result to all its dependent instructions which will re-execute if the result is different than the one received earlier [52, 72, 89]. For loads and stores we used a variation of the selective invalidation scheme described in [72] where timestamps are associated with each store and propagated to consuming loads. The last recovery mechanism we simulated is the one typically used in modern processors to support control speculation. This mechanism invalidates all instructions starting from the one that was mispecu-

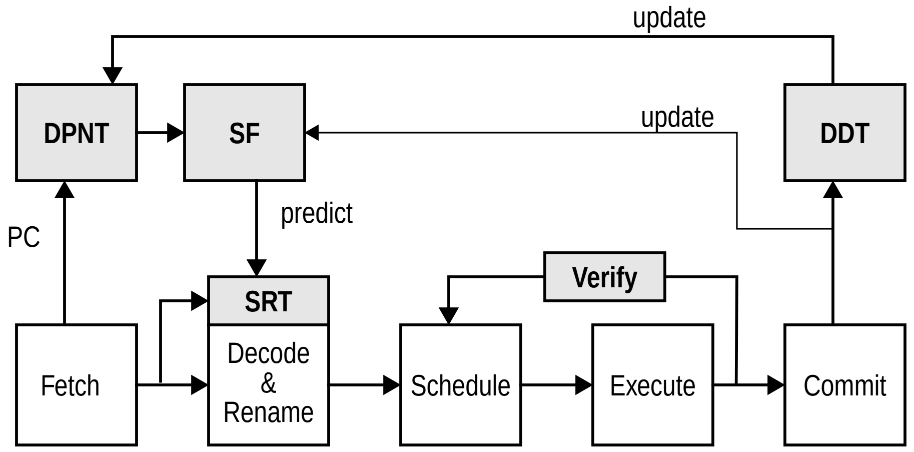


Figure 4.6: *A out-of-order processor pipeline with a cloaking/bypassing mechanism*

lated. These instructions have to be re-fetched from scratch. We will refer to this mechanism as *squash invalidation*.

A challenge shared by most techniques that try to speculate instruction outcomes is how quickly it can be established that the values so obtained are correct, an action to which we will refer to as *data speculation resolution*. Furthermore, as also reported in [80], care must be taken to avoid destructive interference with other prediction techniques, especially branch prediction. For the purposes of this study we assume the ability to resolve all speculation in a register dependence chain as soon as it is established that its input values are correct. Whether such a mechanism can be built is still an open question. The exact data speculation resolution mechanism we simulated, data speculation resolution proceeds in two steps: (1) when an instruction is issued, a determination is made on whether it is data speculative, and (2) when an instructions writes back a non-speculative result, a determination is made on which instructions become non-data speculative. We first explain how we determine whether an instruction is data speculative at issue time: two flags are kept per instruction indicating whether the instruction is currently data speculative. The first flag indicates whether the instruction is data speculative via a register dependence and the second whether it is data speculative via memory. An instruction is data speculative when at least one of these flags is set. Initially, all instructions are marked as data speculative as they pass through the decode stage. When an instruction is issued, a determination is made on whether its register data speculative flag should be cleared. The flag is not cleared under the following conditions: (1) the instruction is using a value obtained via prediction (i.e., cloaking/bypassing or value prediction), or (2) the instruction is using a result produced using a data speculative value. For (2) to be possible, all we have to do is mark an instruction as data speculative when it receives a data speculative result. The required information is already there in a typical scheduler. For memory, we do not rely on dependences to determine whether a value is data speculative. The reason is that memory dependences may change over time as a side-effect of value speculating on base register operands. Instead of attempting to track memory dependences, we use an incremental approach similar to what is done for control dependences: The memory data speculative flags of loads and stores are cleared using a global pointer. This global pointer points to the earliest in program order store or load that has either used a data speculative value (i.e., its register data speculative flag is set) or has not yet calculated its address. Every cycle, the global pointer is updated by scanning forward in the load/store queue, clearing the memory data speculative flags of loads and stores until one is found that meets the aforementioned criteria. With this scheme a load is marked as non-value speculative only when all preceding loads and stores have calculated their addresses and posted their data using non-speculative values. We now explain how, when a non-speculative value is written back, the data speculative flags of instructions are cleared for values obtained through register dependences. When a non-speculative result is written back, all instructions that directly use this result via a register dependence, are marked as non-data speculative. This takes care of register dependence chains of unit depth. For longer dependence chains, we maintain a bit vector per instruction marking the source register dependences it has. These bits are set at decode time. Each bit is connected to a line indicating whether the corresponding instruction is data speculative or not. These signals are generated by merging together (logical OR) the register and memory data speculative flags of the corresponding instruction. Every cycle, these signals are used to determine whether the instruction is still data speculative.

Finally, in order to avoid interfering with branch prediction we disallow control resolution on branches with value speculative inputs.

4.6.6.2 Performance with a Cloaking/Bypassing Mechanism

Figure 4.7 reports how performance varies when cloaking/bypassing is used. Reported are the speedup or slowdown with respect to the base processor that uses no cloaking/bypassing. Part (a) shows the speedups obtained with the oracle (left bar) and the selective (right bar) invalidation mechanisms. Part (b) reports performance variation with the squash invalidation mechanism. While performance improvements are observed for all programs studied with either the oracle or the selective invalidation mechanism, performance rarely improves when the squash invalidation mechanism is in place. In fact, for most of the programs the combination of cloaking/bypassing and squash invalidation often leads to performance degradation. As with any speculation technique, in cloaking/bypassing care must be taken to balance the benefits of correct speculation against the net penalty of erroneous speculation. This result suggests that while cloaking/bypassing may improve performance when correct, the performance improvements so obtained cannot offset the loss incurred if all instructions following a mispeculation have to be re-fetched. To overcome this limitation we may seek to improve the accuracy of the cloaking predictor, or, as the results of part (a) suggest, use an invalidation mechanism that reduces mispeculation penalty.

Focusing on part (a), we can observe that if it was possible to avoid all mispeculations performance improvements are observed (oracle bar). Moreover, the selective invalidation mechanism offers performance that is virtually identical to that of the oracle disambiguation mechanism. As shown, performance improvements vary significantly from as low as 1.8% for 132.ijpeg to as high as 11% for 147.vortex. It should be noted that often the selective invalidation mechanism leads to better performance than the oracle mechanism. There are two reasons why this is so. The first has to do with our simulation methodology. The oracle mechanism as simulated, does not permit any cloaking/bypassing on instructions that will later get squashed due to branch mispeculation. However, speculating on these instructions may positively affect performance as they may, for example, prefetch data cache blocks that are actually needed. The second reason has to do with the nature of the computations that may get speculated. Consider the following sequence of two instructions: “load r1; and r1, 1”, where r1 is a register and the load may receive a value from cloaking. The oracle mechanism decides on whether to speculate or not by looking at the load’s value. If the value from cloaking/bypassing is different than the one that will be read from memory, speculation is inhibited. However, after the “and” instruction only the last bit of the value is relevant. Provided that the value from cloaking/bypassing is correct at least in that bit, any instructions that use the result of the and instruction will be correctly speculated. The selective invalidation mechanism benefits from those cases.

Figure 4.8 reports additional characteristics of the cloaking/bypassing traffic that are useful in interpreting the performance results. In part (a), shown are the data mispeculation rates observed for the selective (left bar) and the squash (right bar) invalidation mechanisms. As performance is affected by the instructions that use the speculative values (the consumers of loads that get a value from cloaking/bypassing) we report mispeculation rates over *all committed* instructions and moreover count a mispeculation for every instruction (not only loads) that used erroneous data. In these measurements, we do not include data mispeculations on

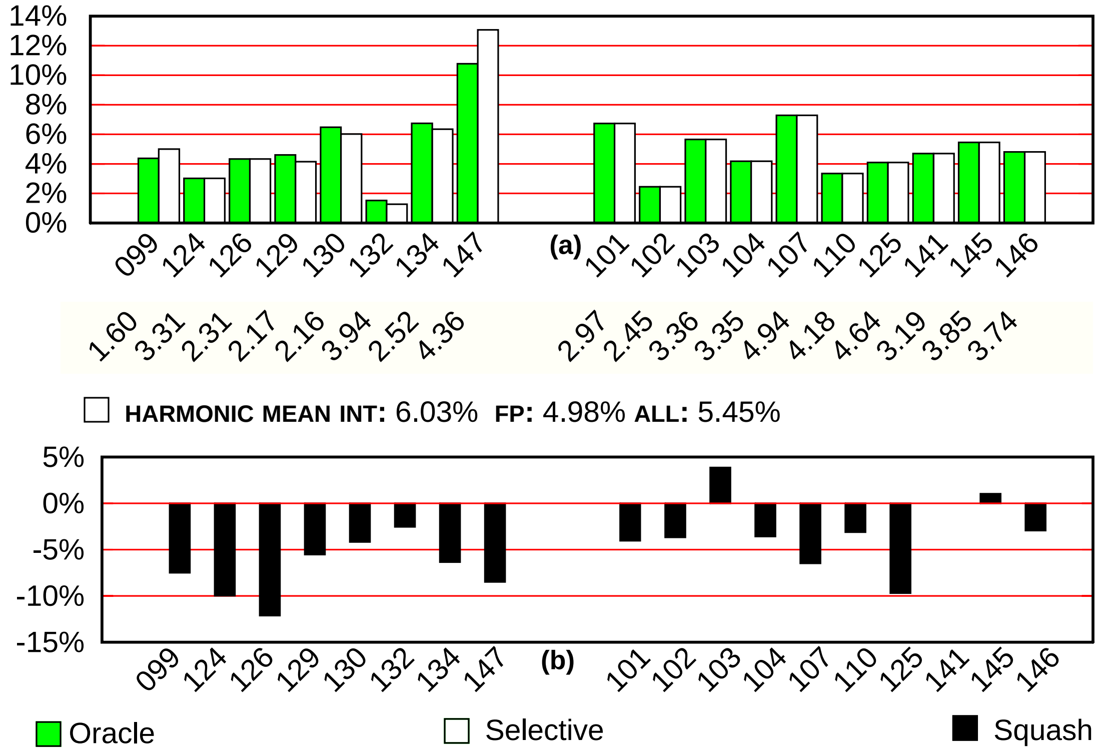


Figure 4.7: *Relative performance of cloaking/bypassing for various mispeculation handling mechanisms. (a) Oracle and Selective invalidation. (b) Squash invalidation. Performance changes are relative to a processor that uses no cloaking/bypassing. IPCs for the base case are reported in the gray strip underneath the graph of part (a). An 8K-entry 2-way set associative DPNT table and an 1K fully associative synonym file are used.*

instructions that are squashed as the result of control mispeculation. It can be seen that selective invalidation is very effective in handling relatively high mispeculation rates (as much as 7% for 134.perl). It can also be seen that with squash invalidation mispeculation rates are much smaller. This is to be expected as a squash invalidation throws all instructions following the earlier mispeculation.

Part (b) of Figure 4.8 shows a breakdown of the fraction of committed loads that were identified as potential consumers by cloaking. Loads are classified into those that were given a value when the prediction was made (synonym or register file) and to those that had to use bypassing (actual producer was in the window and had not yet committed). We can observe that there is not a strong correlation between cloaking/bypassing coverage and performance. Consider for example 099.go and 124.m88ksim. 099.go sees a 5% improvement by cloaking/bypassing while 124.m88ksim sees only a 3% improvement. In contrast, 40% of loads in 099.go use cloaking/bypassing while nearly 80% of loads do so in 124.m88ksim. This result demonstrates that ultimately, performance is affected by whether the consumers of loads can issue earlier if they are given a speculative value and by whether they are part of the critical path of the computation. Even when we correctly speculate a load value, no benefits may be possible if no consumers are visible or if the consumers have other dependences pending. Performance improvements may not also be possible if the processor has to wait anyhow for some other part of the computation to complete.

Comparing the cloaking coverage results of part (b) with those of our preceding trace-driven evaluation we can observe that they are very close. This result suggests that delaying dependence detection and predic-

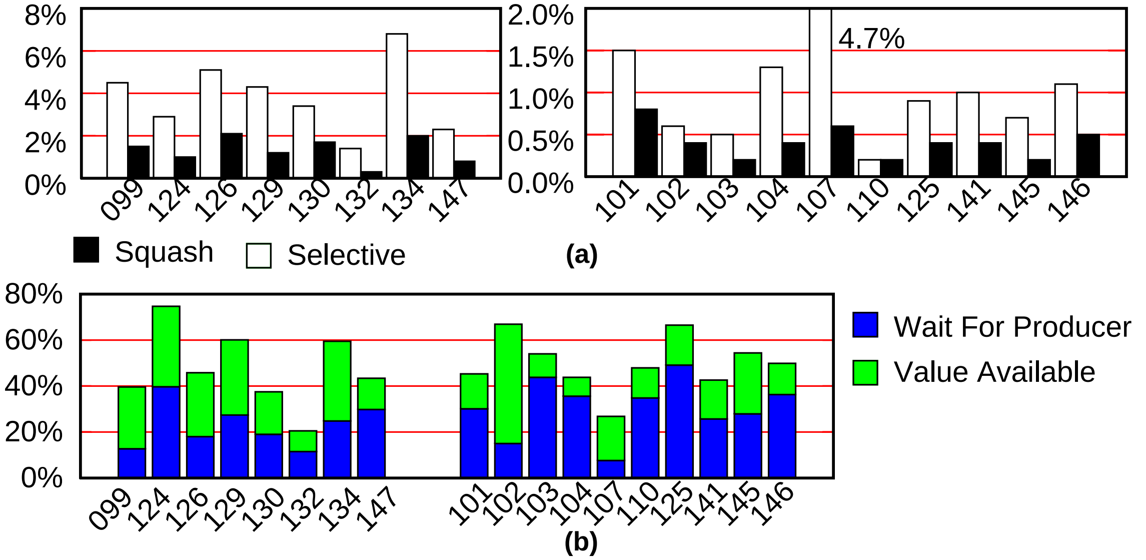


Figure 4.8: Characteristics of cloaking/bypassing communication. (a) Mispeculations observed for the squash and selective invalidation schemes. Rates are over all committed instructions. (b) Fraction of committed loads that get a value via cloaking/bypassing. The upper part of the bar reports loads that get a value at decode time. The lower part reports loads that find that the actual producer of the value is in the window and has not yet produced a value. These loads benefit from bypassing.

tion updates until instructions are committed does not severely impact cloaking coverage. In fact, in some cases (e.g., 145.fpppp) cloaking coverage improves as mispeculations do not affect the predictor immediately (however, mispeculations also increase in this case). As the results of part (b) indicate, for most programs at least half of the speculated loads obtain a value using bypassing. This is to be expected as our base processor has a relatively large instruction window and a fairly aggressive front-end. As we have reported in Section 4.6.4.5, most of the loads that may get a value from cloaking do so over relatively short dynamic instruction distances.

4.6.6.3 Comparing Cloaking/Bypassing and Value Prediction

We next compare cloaking/bypassing with a simple, last-value with no hysteresis load value predictor. For this experiment we use a 16K-entry fully-associative value predictor. Each entry of this predictor also contains the confidence automaton of Figure 4.12 (we experimented with other confidence mechanisms and found that this automaton yields superior accuracy for value prediction also). The accuracy characteristics of this value predictor are almost identical to those reported in Figure 4.19. In part (a) we report performance improvements over the base configuration that does not use cloaking/bypassing or load value prediction. Included are results for cloaking/bypassing mechanisms with 8K and 16K 2-way set-associative DPNTs. Furthermore, we used the selective invalidation mechanism outlined in the beginning of this section. We can observe that increasing the size of the DPNT improves performance for some of the programs. It can be seen that for most programs the cloaking/bypassing mechanism offers better performance than the particular value predictor. However, 132.ijpeg, 104.hydro2d and 125.turb3d, see greater improvements from value prediction. In both techniques the performance tradeoffs are similar. Performance may improve *only when* instructions that depend upon loads execute earlier using a correctly speculated value. Performance may be negatively affected when erroneous speculation causes instructions to execute multiple times consuming resources that could be used more productively. So, in either case we have to balance between the benefits of correct speculation and the impact of erroneous speculation. As we have seen cloaking/bypassing offers, in most cases, equal or better accuracy than the particular load value predictor. This is one reason why cloaking/bypassing results in better performance. Another reason is how quickly the speculative load values become available. In load value prediction, the predicted value is always available early in the pipeline, while, in cloaking/bypassing getting a speculative value may be delayed until the actual producer creates it. While getting a predicted value earlier may be beneficial when the value is correct (dependent instructions may execute earlier) it is not so when the value is incorrect as the speculatively executed instructions consume resources that could be used more productively.

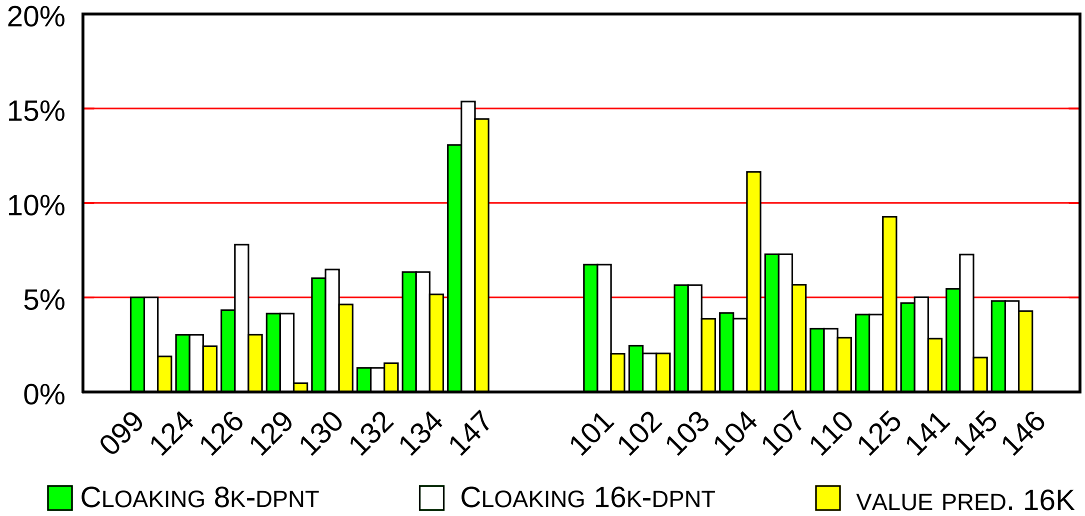


Figure 4.9: *Comparing a load value prediction mechanism with a cloaking mechanism.*

4.6.6.4 Combining Cloaking/Bypassing and Value Prediction

As we have discussed in Section 4.6.4.4, it may be beneficial to combine value prediction and cloaking/bypassing. In this section we evaluate two alternatives, the first uses cloaking/bypassing for RAW dependences and value prediction for other loads. Combining RAW memory dependence prediction with value prediction was suggested by Tyson and Austin [89]. Here, we extent that mechanism by using bypassing for RAW dependences. In the other mechanism we utilize both RAW and RAR prediction for cloaking/bypassing. Preference is given to value prediction. Cloaking/bypassing is used when value prediction fails either by mispeculating a load or by not speculating at all. Performance results are given in Figure 4.10, where the two aforementioned combinations of value prediction and cloaking/bypassing are marked as "CLOAK/VP" and "VP/CLOAK". With the exception of 104.hydro2d and 125.turb3d, all three mechanisms that use cloaking/bypassing resulted in similar performance improvements. For most programs, using a combination of cloaking/bypassing and value prediction results in slightly better performance. A notable exception is 099.go, whose performance improvements are reduced with either of the combined mechanisms. Comparing the two combined mechanisms, we can observe that the mechanism that attempts to use value prediction first (VP/CLOAK) and that also utilizes RAR dependence prediction performs slightly better for most programs. For 126.gcc, 104.hydro2d and 107.mgrid the mechanism that does not use any RAR dependence prediction resulted in slightly better performance. However, overall the differences are minor and cannot be used as an indication of the superiority of any of the mechanisms studied over the rest of them.

The results of these experiments support our earlier observation of a synergy between cloaking/bypassing and value prediction. Unfortunately, we have seen that the performance improvements by combining the

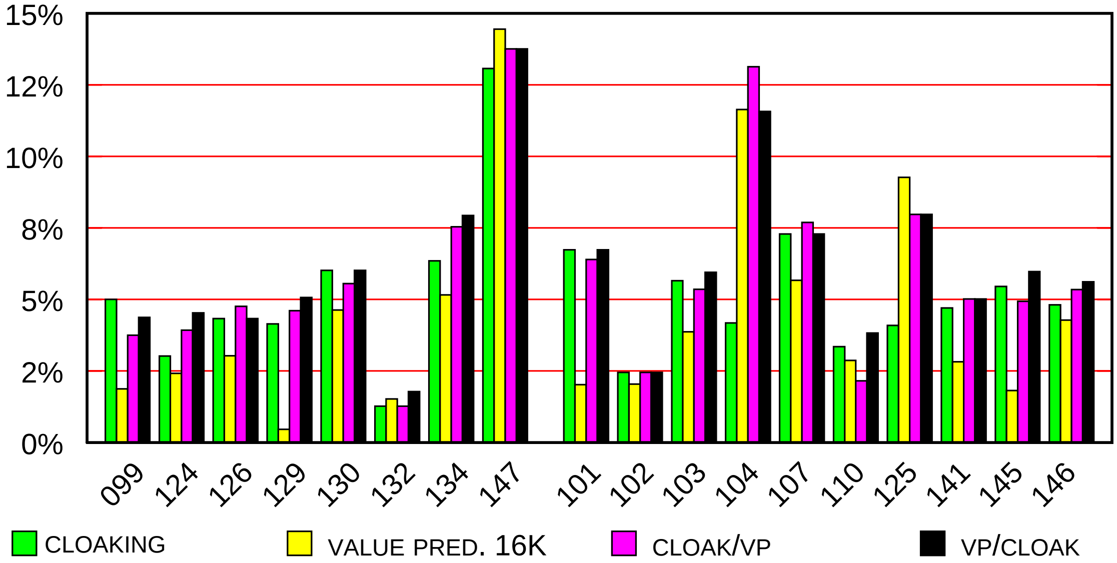


Figure 4.10: *Performance improvements obtained by combining load value prediction and cloaking/ bypassing. All cloaking/bypassing mechanisms use a 8K DPNT.*

two techniques obtained are relatively minor. Further, investigation is required to determine whether performance is limited by the program itself (i.e., whether some other part of the program becomes the critical path when the techniques are combined) or whether it is the limited bandwidth of the processor model we used that inhibits any further improvements. Such an investigation is beyond the scope of this work.

4.7 Summary

In this chapter we identified and were motivated by two prevalent uses of memory: inter-operation communication (passing a value from a store to a load) and data-sharing (two or more loads access a common memory location). We reviewed how these actions are specified using the traditional address-based memory interface and identified a set of potential drawbacks: calculating an address (fetching input registers and performing an addition), disambiguation (determining whether a preceding, yet uncompleted store will provide the value and where the value is going to be placed) and finally, memory system access. Noticing that using run-time calculated addresses is not the only way of specifying these actions we were concerned with methods of explicitly specifying memory communication and data-sharing and of exploiting this information to streamline loading values from memory. We proposed using memory dependence prediction to identify the memory communication and data-sharing patterns dynamically, annotating the corresponding instructions. We used memory dependence prediction as the basis of speculative memory cloaking, a novel technique where we create a new, albeit speculative name space through which loads and stores can access values without having to incur some of the overheads associated with address calculation, disambiguation and memory system access. Furthermore, observing that loads and stores act simply as agents passing values that other instructions create or use we proposed speculative memory bypassing where the instructions that produce or consume memory values via stores and loads are linked directly. Both techniques are speculative in nature and for that require that the values so obtained are verified through the traditional memory name space also. For this reason, these techniques can only reduce the latency associated with reading memory values.

Evaluating cloaking and bypassing we have seen that most of the executed loads read a value that was written by a recent store or read by a recent load. Moreover, we have demonstrated that the majority of the corresponding dependences can be captured with relatively small structures that can hold the last 128 memory locations accessed. Furthermore, we have demonstrated that this history information can be used to predict future dependence behavior with relatively high accuracy and showed that a cloaking mechanism that utilizes this prediction can provide correct values for about 70% and 50% of all loads executed in the integer and the floating point program respectively, while maintaining mispeculation rates of roughly 2% and 0.4% respectively. We have identified that important sources of mispeculations are: (i) the save and restore stores and loads respectively of recursive functions, and (ii) loads that have multiple RAR dependences some distant and less regular and some in close proximity that are fairly regular. We also demonstrated that while RAW dependences are more frequent for the integer codes, RAR dependences are more frequent in floating point codes. Moreover, we showed that even through the use of a small dependence detection structure, we can capture and predict dependences that span across much larger distances in terms of dynamic instructions (some of the dependences captured were between instructions more than 16K instructions apart).

We also investigated the address and value locality characteristics of the loads that get a value via cloaking. We showed that cloaking captures most of the address locality found in the programs studied and while also capturing some loads that do not exhibit any address locality. We also looked at the value locality characteristics of the loads that get correct values from cloaking and found high correlation between the type of dependences experienced by a load and its value locality characteristics. We saw that loads that experience RAW dependences exhibit relatively low value locality while the loads that experience RAR exhibit relatively high value locality. For most programs, few other loads exhibit value locality. Moreover, we found that cloaking offers superior accuracy when compared to a simple, yet realistic last-value based value prediction mechanism. However, we also found that a synergy exists between value prediction and cloaking/bypassing.

Moreover, we investigated the performance impact, a combined implementation of cloaking and bypassing has on a dynamically scheduled ILP processor. To do so we used a fairly aggressive processor with an 128-entry instruction window that performs naive memory dependence speculation as our base. We found that performance was extremely sensitive to the mispeculation recovery mechanism used. When squash invalidation was used, performance improved slightly only for two programs. In fact, performance degradation varying from as low as 2.5% to as much as 12% was observed for the rest of the programs. However, when combined with an implementation of selective invalidation, cloaking/bypassing offered performance improvements for all programs studied. The observed speedups were in the range of 1.5% to 15.4% with the harmonic mean being 4.1% (16K, 2-way set associative DPNT, 1K 2-way set-associative synonym file).

Finally, we compared our cloaking/bypassing mechanism to a straightforward last-value, load value predictor and found that cloaking/bypassing offered superior performance for all programs except 104.hydro2d. We also considered two mechanisms that combined load value prediction and cloaking/bypassing and found the although some benefits were possible, in most cases those benefits were relatively small for the processor model we used. However, these mechanisms may perform better under different processor models. For example, they may perform better: (1) when memory latency is higher as now the potential benefits of obtaining a memory value as early as possible are increased, and (2) when the instruction window is increased as in this case value prediction may supply a memory value long before cloaking or bypassing could.

While conceptually exciting, our results suggest that support for selective invalidation and fast data speculation resolution has to materialize before the techniques presented can be of practical use (under similar assumptions about the processor configuration and memory access latency).

Chapter 5

Transient Value Cache

Highly parallel execution can benefit from both low memory latency and from the ability to perform multiple memory accesses in parallel. The techniques we have presented so far aimed at providing low memory latency by either scheduling loads as soon as possible (Chapter 3) or by creating a speculative name space through which loads can obtain a value without incurring the overheads associated with the traditional memory address space (Chapter 4).

In this chapter, we present the *Transient Value Cache* (TVC for short), a method of supporting multiple memory access per cycle. The TVC is a novel memory hierarchy component that combines a memory dependence status predictor and a relatively small, narrow data cache. The basic approach is illustrated in Figure 5.1. The TVC records in its data cache component the *n*th most recent accessed memory locations as a traditional data cache would. However, contrary to what is done in a traditional memory hierarchy, the data cache component does not always appear in series with the rest of the memory hierarchy. Instead, its placement is decided using a memory dependence prediction as follows: When a load is ready to access memory, a prediction is made on whether the memory location it will access is resident in the TVC, or viewed differently whether the load has a RAW or a RAR dependence with a recent store or load respectively. If so, the load is sent only to the TVC, in which case the TVC appears *in-series* with the L1 data cache. Otherwise the load is sent to both the TVC and the rest of the memory hierarchy, in which case the TVC appears *in-parallel* with the L1 data cache. Provided that prediction accuracy is high, the potential benefits of the TVC approach are: (1) the loads that hit in the TVC are hidden from the rest of the memory hierarchy, freeing up L1 data cache ports to be used by other loads, (2) the latency of loads that are unlikely to hit in the TVC remains unchanged. High prediction accuracy is essential as the TVC may result in increased load latency when prediction incorrectly indicates that a load will find its data in the TVC. We also discuss a possible WAW dependence status prediction extension, where the TVC is also used to hide from the rest of the memory hierarchy those store accesses that are likely to be overwritten quickly, precluding potential problems with writeback traffic contention while reducing L1 port requirements (see discussion in Section 5.2).

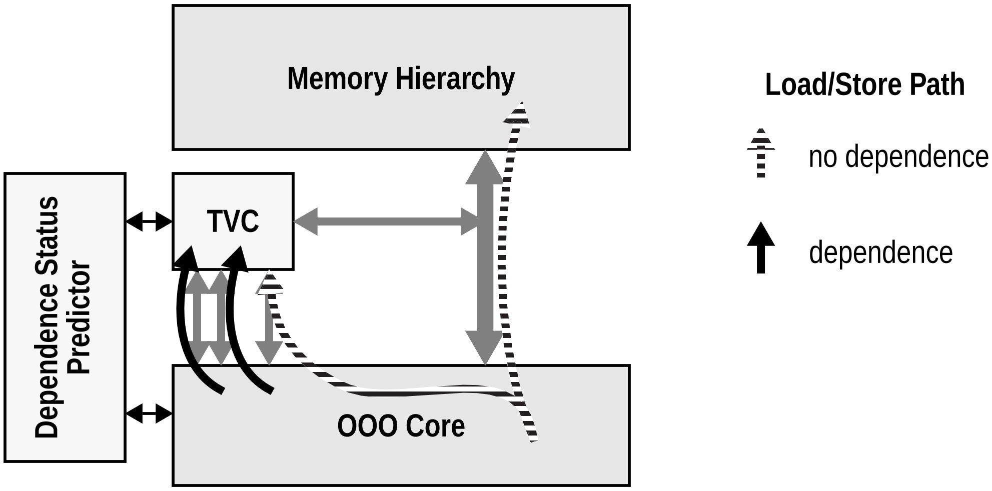


Figure 5.1: *The Transient Value Cache*

For the TVC to be effective, programs must exhibit a significant fraction of loads with short-distance RAW or RAR dependences. Furthermore, the existence of those dependences must be predictable with high accuracy. Moreover, if we decide to use the WAW extension, the aforementioned characteristics should be exhibited by the WAW dependence stream of programs. We have presented evidence that such behavior exists in Chapter 2. In Section 5.1, we take a closer look at the short-distance RAW, RAR and WAW memory dependence behavior of programs. We use the results of this analysis to motivate the TVC approach which we describe in Section 5.2. In Section 5.3, we review related work. In Section 5.4 we present experimental evidence in support of the benefits of the TVC approach. Finally, in Section 5.5 we summarize our findings.

We should warn the reader that the evaluation presented in this chapter is preliminary. For this reason, the results presented should be interpreted merely as an indication of the potential of the TVC approach. Further investigation is required to determine how a TVC affects performance.

5.1 Short-Distance Memory Dependence Measurements

The TVC method is best motivated by examining the short-distance memory dependence behavior of programs. For this reason, we first present an empirical study of the short-distance memory dependence behavior of loads and stores. Specifically, to get an estimate of the fraction of loads or stores that the TVC can potentially hide from the rest of the memory hierarchy we measure: (1) the fraction of loads that have a RAW or RAR dependence, and (2) the fraction of stores that have a WAW dependence with a succeeding store. We measure both fractions as a function of the number of memory locations that can be stored in a word-wide, fully-associative data cache. We have presented similar measurements in Chapter 2, where we were concerned with the amount of resources required to capture a desired level of memory dependence activity. The analysis of this section differs in that we take a closer look at the lower end of the spectrum and consider address window sizes that seem more appropriate from a practical perspective. Figure 5.2, reports, in part (a), the fraction of loads that would find the data they need in the data cache, and in part (b), the fraction of stores that would be killed in the data cache. The size of the data cache is varied from 32 to up to 8K entries, and samples are taken at sizes that are powers of two.

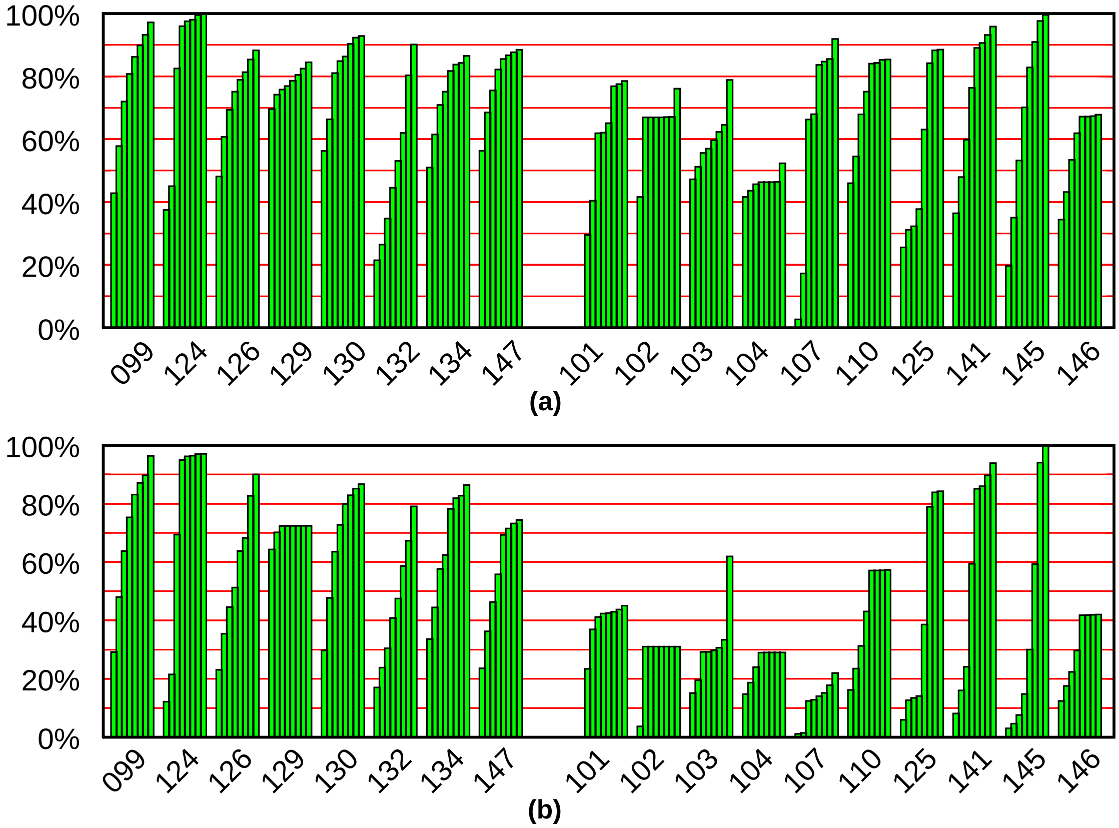


Figure 5.2: *(a) Loads with a RAW or RAR dependence within the last n unique, most recently accessed memory locations. (b) Stores that have a WAW dependence within the next n unique memory locations accessed. Range of n shown is 32 to 8K in steps that are powers of two.*

Focusing first on the loads (part (a)), we can observe that most programs exhibit relatively large fractions of loads that read a recently written or read memory location. Even when we keep a record of just the last 32 accessed memory locations, we can often capture around 40% or more of all load accesses. As it can be seen by the results of part (b), a significant fraction of stores also get killed within the limits of a relatively small data cache. However, the fraction of stores that gets killed within a data cache of a given size is typically smaller than the fraction of loads that would find their data in a data cache of the same size. As we also noted in Chapter 2, the WAW dependence behavior of floating point programs is different from that of integer programs. In the integer programs, the fraction of stores with WAW dependences rises sharply with the size of the data cache, exceeding 65% with a data cache of 8K. In contrast, the increase in stores with WAW dependences is not as sharp for floating point programs. Moreover, for most floating point programs, the fraction of stores with WAW dependences is typically lower compared to the integer programs. For example, in 104.hydro2d the fraction of stores with WAW dependences is near 30% even when an 8K data cache is used.

The results of this section suggest that a data cache that can hold a relatively small number of recently accessed or written memory locations, can potentially service significant fractions of loads and stores. For example, a data cache capable of recording 128 memory locations can provide values for 62% (70.2% for the integer codes and 56.3% for the floating point codes) of all loads on the average. In the same data cache, 38% of all stores (55% for integer codes and 23% for floating point codes), will get killed on the average.

Motivated by these empirical observations, in the next section we present the transient value cache which exploits this behavior to reduce the bandwidth requirements imposed on a traditional memory hierarchy.

5.2 The Transient Value Cache

To goal of the TVC approach is to provide support for multiple memory requests per cycle, reducing the port requirements imposed on the L1 data cache. Traditionally, support for multiple memory requests was provided by multi-porting (e.g., replication) or by partitioning (e.g., banking) of the L1 data cache. However, it is likely that L1 data cache sizes will have to increase in future generation processors to compensate for increased working sets and relatively slower main memories or other levels of memory hierarchy. Furthermore, it is likely that in future processors, wire propagation delays may limit the amount of resources reachable within a clock cycle thus making larger caches relatively slower [98]. Under these constraints, we argue that it is desirable to service as many memory requests as possible using relatively small storage structures. The reasons are that: (1) relatively small structures may occupy less area and for that may be faster (e.g., [98]), and (2) the cost of replicating a small structure for the purposes of multi-porting may not be prohibitive.

As the results of the previous section suggest, for the programs we studied, behavior is such that even relatively small data caches could service significant fractions of loads and stores (a discussion on whether this observation will apply to future workloads is delayed until later in this section). The TVC approach is best motivated if we first consider how a traditional memory hierarchy could be extended to exploit the aforementioned program behavior. In a traditional memory hierarchy, the various data cache levels are placed serially. The straightforward way of exploiting the phenomena identified in the previous section, is to introduce an additional level of caching between the L1 data cache and the processor. This is shown in Figure 5.3, where the newly introduced cache is referred to as the L0 data cache. The advantage of this organization is that now, loads that will hit in the L0 data cache may benefit from the additional ports provided. Similarly, stores that get killed in the L0 are hidden from the L1 data cache. In either case, the processor may benefit from the ability to issue multiple requests at the same time, while the bandwidth requirements on the L1 data cache are reduced. Unfortunately, the L0 organization has also potential disadvantages. The first is that the latency of loads that do not hit in the L0 is now increased by the time required to inspect the L0. For this reason, care must be taken to balance between the benefits obtained when loads hit in the L0 and the increases in load latency when loads do not hit in the L0. This trade-off applies to all hierarchies that place caching levels serially. However, in today's memory hierarchies the absolute and relative increases in latency between adjacent caching levels is relatively high, while the miss rates are typically low. For relatively small L0 caches, and as the results of the previous section suggest, miss rates will be relatively higher, while it is likely that the latency of the L0 will be comparable, if not equal to that of the L1. Under these new conditions the benefits of additional ports may not offset the effects of increased load latency for those loads that miss in the L0. An additional concern with the L0 organization is how it may interact with a typical out-of-order instruction scheduler and specifically, when instructions that use a value loaded from memory, can be scheduled for execution. If the scheduler can wait until a determination is made on whether the source load hits in the L0, there is no concern. However, if the scheduler must optimistically schedule the dependent operations in parallel with accessing the L0 or earlier [84], then a high L0 miss rate will translate in frequent replays in the pipeline which may degrade performance.

The goal of the TVC approach is to provide the benefits of servicing a significant fraction of loads and stores using a relatively small data cache, while avoiding the aforementioned problems associated with introducing an additional level of caching between the processor and the L1 data cache. The TVC is nothing more than a small data cache coupled with a memory dependence status predictor. The TVC differs from a typical data cache in that it does not always appear in series with the next level of the memory hierarchy as is the case with traditional memory hierarchies. Instead, the TVC uses memory dependence status

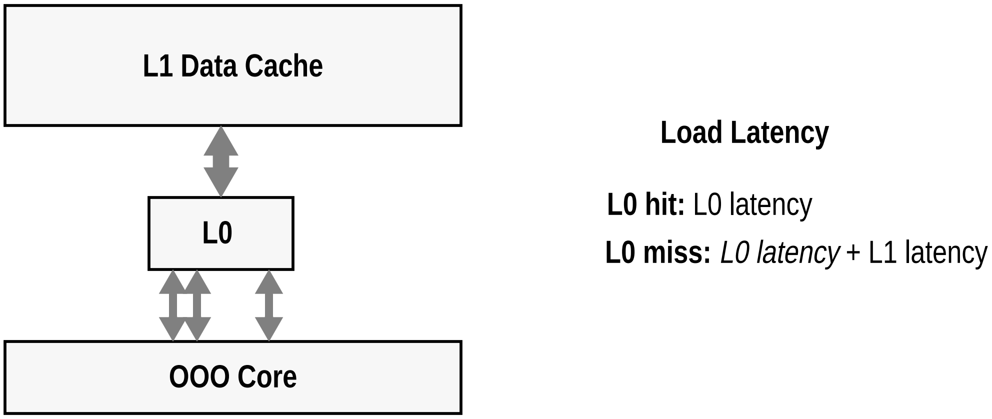


Figure 5.3: *Incorporating an additional level of caching between the L1 data cache and the processor.*

prediction to place itself either in-series or in-parallel with the rest of the memory hierarchy on a per access basis. Before a load is performed, a prediction is made whether it will access data written or read by a recent load or store. If so, the access is directed first to the TVC, otherwise it is sent to both the TVC and the L1 data cache in parallel. Provided that the dependence status of an instruction is correctly predicted the benefits are two-fold: (1) accesses that find the data they need in the TVC do not consume L1 ports which may be used by other memory instructions, (2) accesses that are not likely to find their data in the TVC do not pay the latency of having to first go through the TVC. High prediction accuracy is essential as when prediction is incorrect one of the following two scenarios apply: (1) load latency will increase by the time required to first go through the TVC and then to the L1 cache, or (2) a load will be incorrectly exposed to the L1 unnecessarily consuming an L1 data port. In the first scenario, the TVC is incorrectly placed in series with the L1 data cache. However, this is no different than what would have happened with the L0 organization of Figure 5.3. In the second scenario, the TVC is incorrectly placed in parallel with the L1. This case represents a lost opportunity for L1 data port bandwidth reduction.

We can also extend the TVC with a WAW dependence status predictor, in which case a prediction is made on whether a store will be killed in the TVC before it is evicted to the L1 data cache. The potential benefits in this case are indirect. The L1 port scheduler may use this information to better utilize idle L1 port cycles by writing those stored values that are more likely to not get killed in the small data cache. However, in the case of incorrect WAW prediction, the TVC may not hide some of the stores that would have been killed with the organization of Figure 5.3.

The operation of the TVC is summarized in Figure 5.4 where we show how dependence status prediction is used to steer loads and stores. Loads that are likely to have RAW or RAR dependences with recent stores or loads respectively, are initially sent only to the TVC. Such loads are directed to the data cache only if we miss in the TVC (part (a)). In the latter case we do bring the data in the TVC. Other loads have to access both the TVC and the data cache in parallel (part (b)) since the most recent value may be only in the TVC. If the WAW extension is used, then stores that are likely to be killed soon are initially sent only to the TVC in hope that they will be killed in it before they are forced to go the data cache (part (c) — in an L0 organization all stores will first get exposed only to the L0 if a write-back policy is used). Other stores are sent to both caches to keep them coherent (part (d)). If a dirty block in the TVC needs to be replaced, its contents will have to be written to the data cache.

The TVC requires a memory dependence status predictor. The memory dependence predictors we use in our evaluation associate dependence status information with static loads and stores via their PCs. To collect

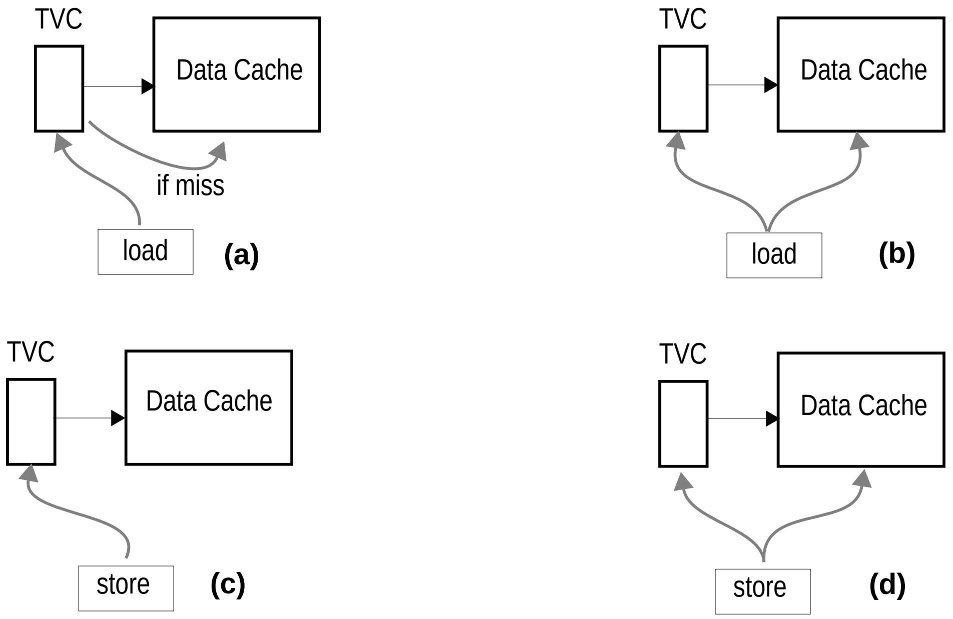


Figure 5.4: *Transient value cache operation. Loads: (a) RAW or RAR dependence predicted, (b) no RAW or RAR dependence predicted. Stores: (c) WAW dependence predicted, (d) no WAW dependence predicted.*

this information, a method of detecting the *existence* of memory dependences is required. Since we are interested in the dependence status of instructions, the exact dependences are not important, only whether dependences exist. In the case of loads, detecting whether the load has a RAW or a RAR dependence within the limits of the TVC is equivalent to whether the load has found the data it needs in the TVC. This observation suggests that the TVC itself may also serve as a detection mechanisms for RAW and RAR dependences. Detecting whether a store is overwritten by a another store is not as simple. The reason is that at the time the WAW dependence occurs we want to associate the event with the store that wrote the data being overwritten. For this reason, detecting and predicting WAW dependences requires to also record the PC of the store along with the data in the TVC itself.

We should also note that in a shared memory multiprocessor environment and subject to the consistency model in use, we may have to expose all memory operations to the coherence mechanism. However, the issues are no different than those applying to any memory hierarchy.

We conclude the discussion of this section by commenting on the following two issues: (1) whether the underlying phenomena that the TVC exploits will exist in future workloads, and (2) how the TVC might interact with the cloaking/bypassing methods we presented in Chapter 3.

We motivated the TVC by observing that the behavior of the programs we studied is such that even relatively small data caches could service significant fractions of loads and stores; many loads and stores read or write on memory address that was recently accessed by another load or store. A valid concern however, is whether the aforementioned observations will or will not apply for future workloads. Unfortunately, there is no way of providing a definite proof for either possibility. However, we do offer two qualitative arguments: one that suggests why short-distance memory dependences may still represent a reasonable fraction of all memory traffic in some future workloads and one that suggests that the TVC approach may at least not harm performance when short-distance memory dependences are not exhibited. As the results of Chapter 4 suggest much of the short-distance memory traffic can be attributed to inter-operation communication through memory or to repeatedly reading a memory value. These phenomena are partially the result of how programs operate and of how this operation is expressed. Often times, programs perform calculations whose results are quickly used for further processing. Such behavior gives rise to short distance inter-operation communication. Moreover, programs often use constants or values that do not change for some period of time. Such behavior gives rise to short-distance RAR memory dependences. Finally, the use of structured programming techniques, of object-oriented programming techniques and of dynamic linking often results in increased memory traffic (passing parameters, saving and restoring registers on method/procedure calls). It will take a program that scans through large data structures without reusing any of its memory stored results or memory stored constants for short distance RAW and RAR dependences to be infrequent. Finally, even when short-distance RAR and RAW dependences are not frequent the TVC being an adaptive structure may at least succeed in avoiding increasing latency for loads. Although, the TVC will not provide any benefits for such programs, it is highly probable that it will also not cause any harm.

Finally, it is also interesting to consider how the TVC interacts with the cloaking/bypassing mechanism presented in Chapter 4. Recall, that cloaking/bypassing utilizes RAW and RAR dependence prediction to supply memory values early in the pipeline. The loads that cloaking/bypassing is capable of handling are those that have dependences detected via a DDT of moderate size. These loads, in their majority are the loads that the TVC aims to hide from the rest of the memory hierarchy. The potential exists to combine the two techniques, sharing a common prediction structure. In this case, the TVC and cloaking/bypassing may be used to both reduce the latency for a large fraction of loads and to also hide them from the rest of the memory hierarchy reducing the bandwidth requirements imposed on the L1 data cache.

5.3 Related Work

A plethora of cache related techniques and studies has been reported. In this section we review work that specifically targets supporting multiple, simultaneous requests at the L1 data cache-processor interface, and related studies. A variation of the TVC approach, in which only RAW memory dependence status prediction was used for loads, was presented in [62].

A number of previous studies have focused on the bandwidth requirements of highly-parallel processing. Sohi and Franklin [83] argued for the need of high-bandwidth memory systems in order to support the data bandwidth demands of future, wide-issue processors. They proposed a number of organizations that can be used to support multiple, simultaneous load and store requests. Techniques suggested include lockup-free caches [48] and multi-ported or multi-bank caches. Another technique to support multiple memory requests per cycle is *time division multiplexing* or *virtual multi-porting.* In this technique, which is, for example, used in the IBM Power2 [76] and in the Alpha 21264 [11], multi-porting is achieved by performing multiple (two in the specific implementations) data cache accesses serially within a single clock cycle. Another technique, utilizes multiple copies of the L1 data cache. This technique is used for example in the 21164 Alpha processor [45].

Wilson, Olukotun and Rosenblum [96, 95] studied how performance varied with the number of available data cache ports, and found that multiple cache ports can have a significant impact on performance. They also suggested a number of organizations that utilize a very small data cache, the *Line Buffer* which is placed in series with the L1 data cache [96]. They demonstrated that a large fraction of memory accesses can be serviced within this small data cache. The assumption of the study is that the line buffer, being relatively small, does not impact load latency even when the data needed is not found there. The TVC aims at offering most of the benefits of the Line Buffer approach even when this assumption is not valid.

Rivers, Tyson, Davidson and Austin propose the *Locality-Based Interleaved Cache* (LBIC) [70], that employs a line buffer per data cache bank to reduce the negative effects of bank conflicts. In their design, accesses to the same memory block can be serviced simultaneously via the use of a line buffer. They also study the effects of multi-porting on the performance of a very aggressive dynamically-scheduled ILP processor and find that true multi-porting can lead to significant performance improvements.

Kin, Gupta and Mangione-Smith also suggested introducing a relatively small data cache, termed the *Filter Cache*, in between the L1-processor interface [46]. The Filter Cache uses the L0 organization of Figure 5.3 as it aims to reduce power at the expense of some performance degradation.

In this work we were motivated by the large fraction of memory accesses that can be serviced with a relatively small data cache. A plethora of previous studies have also looked at the memory reference behavior of programs, with the focus being the optimization of the memory hierarchy. The most relevant to this work studies are the following: McNiven and Davidson [59] analyzed memory reference behavior and suggested using compiler hints to identify values that are killed in order to reduce the traffic between adjacent levels of the memory hierarchy. Huang and Shen studied the minimal bandwidth requirements of current processors taking into account instruction issue rate, memory capacity and memory bandwidth. They also formalized the notion of an efficient memory system, were the smallest possible storage structure is used to meet the data requirements of program execution [35, 36]. They demonstrated that relatively small structures can in principle be used to meet the bandwidth requirements of typical programs.

The TVC approach exploits the high-levels of locality exhibited in the memory dependence status of loads and stores. Abraham, Sugumar, Windheiser, Rau and Gupta, have shown that there is high correlation between misses and the static load and store instructions that cause them [2]. They proposed a number of compiler optimizations to reduce the performance impact of those misses. Tyson, Farrens, Matthews and Plezkun exploited this phenomenon to predict whether the data fetched by a memory instruction will be used again [88]. This prediction was used to increase data cache efficiency by not caching data with low locality. The TVC is orthogonal to either technique as it exploits the high correlation between instructions and miss behavior to hide loads from higher levels of the memory hierarchy.

5.4 Evaluation

In this section we provide experimental evidence in support of the potential of the TVC method. To do so, we assume a memory dependence status predictor of infinite size and measure its accuracy as a function of the size, associativity and data block size of the data cache used. These results, indicate the fraction of loads that would be serviced by the TVC and the fraction of stores that will get killed in it. Furthermore, they also indicate the fraction of loads that will see an increased latency by having to first go through the TVC. We also compare the TVC to L0 organizations that utilize the same data cache components. We should warn the reader that these results should only be interpreted as indications of the potential of the TVC method. The evaluation presented has several limitations, the most important of which are: (1) we do not study the impact the TVC has on performance, and (2) we do not compare with other existing techniques of improving cache port efficiency.

Our focus in on how the TVC interacts with load instructions. This study is presented in Section 5.4.1. Initially, we measure the effectiveness of the TVC approach for various sizes of the data cache components and compare its characteristics to an L0 organization of the same size. In Section 5.4.1.1, we study the effects of using data cache components of reduced associativity. Finally, in Section 5.4.1.2 we study the effects of increasing the block size of the data cache component. In Section 5.4.2, we study the prediction accuracy of a WAW memory dependence status predictor. In this study, we assume fully-associative data cache components.

5.4.1 Load Dependence Status Prediction Accuracy

Figure 5.5 reports the memory dependence status prediction accuracy on loads as a function of the data cache entries used. In these experiments we assumed a fully-associative, word-wide data cache component and varied its size from 32 to 8K entries, in power of two steps. Furthermore, we assumed a dependence status predictor that associated a 2-bit saturating counter with each load and store. A threshold value of two was used. That is, a load should have a dependence detected twice before a positive dependence status prediction could be made. No limit was placed in the number of predictor entries. However, as we have seen in Chapter 2, the working set of loads with dependences, even when the data cache is 64K words, is less than 4K for all programs except 145.fpppp. This result implies that a predictor with at most 4K entries should yield prediction accuracies very close to those possible with the predictor used in this study. Part (a) of the figure reports the fraction of loads that have their dependence status correctly predicted. We can observe that with the exception 099.go and for data cache sizes of 128 entries or more, prediction accuracy is above 90%. For most floating point programs, accuracy is above 95% independently of data cache size. We can also observe that while, typically increasing the size of the data cache improves accuracy, in some cases accuracy may decrease. The reason is that when the data cache size is increased, more dependences get exposed. However, these dependences are not always well behaved. In some cases, the dependences correspond to loads that traverse through structures that only partially fit in the data cache, or to loads that exhibit control flow dependent behavior.

Part (b) of the figure, reports the fraction of loads that are correctly predicted and do find the data they need in the TVC. These are the loads that will be hidden from the rest of the hierarchy thus freeing up L1 data cache port resources to be used by other loads or stores. We can observe that in absolute terms, a large fraction of loads is correctly identified. Moreover, when compared with the hit rate that would be possible with a traditional organization (part (a) of Figure 5.2), we can observe that the vast majority of loads that would indeed find their data in a data cache of the given size are predicted correctly. This is clearly shown by the results of part (d), where we report the loads that are predicted correctly and hit in the TVC as a fraction of the loads that would have hit in a data cache of the same size. With the exception of 132.ijpeg, more then 90% of the loads that would have hit in a traditional organization will do so using the TVC approach.

As we discussed in Section 5.2, the TVC may increase the latency of some loads. This happens when dependence status prediction incorrectly indicates that the load will find the data it needs in the TVC cache. Part (c) of Figure 5.5 reports this fraction. We can observe that in absolute terms, few loads will see a latency increase by the introduction of the TVC. However, for very small data cache sizes, this fraction is noticeable. For example, about 8% of loads in 130.li will observe a latency increase when a 32-entry data cache component is used. Fortunately, as we consider larger data cache sizes the percentage of loads that will be penalized by the use of the TVC drops. In particular, when an 128-entry data cache is used, this percentage drops below 5% for all programs except 099.go. We should however note, that had we used the traditional organization of Figure 5.3, the fraction of loads that would be penalized would have been significantly higher. In fact, all loads that would miss in the L0 data cache would have been penalized. As we have discussed in Section 5.1, roughly 30% and 45% of loads would miss on the average in a data cache of 128 words for the integer and floating point codes respectively.

Table 5.1 compares the TVC approach with a traditionally organized L0 data cache (organization of Figure 5.3). The comparison is done in terms of the load hit rates and of the fraction of loads that will observe a latency increase (miss rate in the case of the L0) compared to an organization that uses neither an L0 nor a TVC. In this comparison we limit our attention to data cache sizes of 128, 256, and 512 word entries (or equivalently, 512, 1K, and 2K bytes). Moreover, we assume fully-associative word-wide data cache components for both the TVC and the L0. Under the TVCHIT% columns we report the fraction of loads that will get hidden in the TVC (these are the loads for which dependence status prediction correctly predicts that

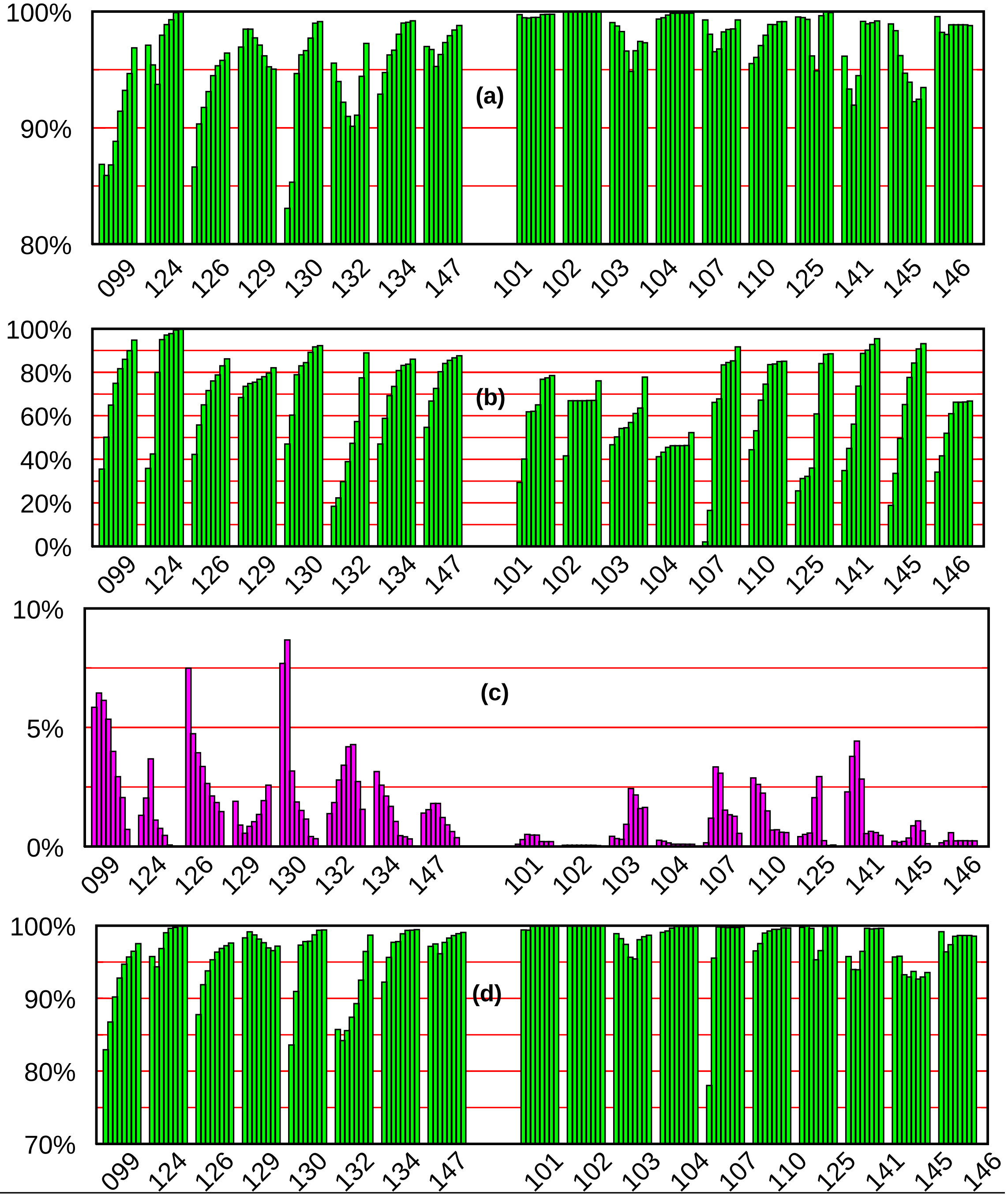


Figure 5.5: *Load dependence status accuracy. (a) Correctly predicted loads, (b) loads that are correctly predicted and hit in the TVC, (c) incorrectly predicted loads that do not hit in the TVC. (d) loads that hit in the TVC as a fraction of the loads that would have hit in a data cache of the same size.*

they will find the data they need in the TVC). That is, the TVCHIT% indicates the positive impact the introduction of the TVC may have. Under the TVCLI% columns we report the fraction of loads that will observe a latency increase (these are the loads for which dependence status prediction incorrectly predicts that they

	TVCHIT%	TVCLI%	L0HIT%	L0LI%		TVCHIT%	TVCLI%	L0HIT%	L0LI%
099	64.9%	6.1%	+7.0%	+28.0%	*101*	61.8%	0.5%	+0.05%	+38.1%
	74.9%	5.3%	+5.8%	+19.2%		62.0%	0.4%	+0.03%	+37.9%
	81.6%	3.9%	4.5%	+13.7%		65.0%	0.4%	+0.03%	+34.9%
124	79.9%	3.6%	2.5%	+17.4%	*102*	66.9%	0.05%	0%	+33.0%
	95.0%	1.1%	0.9%	+4.0%		66.9%	0.05%	0%	+33.0%
	97.1%	0.7%	0.3%	+2.5%		66.9%	0.05%	0%	+33.0%
126	65.0%	3.9%	4.3%	+30.6%	*103*	54.1%	0.3%	+1.4%	+44.4%
	71.6%	3.3%	3.5%	+24.8%		54.5%	0.9%	+2.4%	+43.0%
	76.0%	2.6%	2.8%	+21.1%		56.9%	2.4%	+2.7%	+40.3%
129	74.8%	0.5%	0.9%	+24.2%	*104*	45.4%	0.1%	+0.1%	+54.3%
	75.4%	0.8%	1.4%	+23.1%		46.2%	0.1%	+0.05%	+53.6%
	76.7%	1.0%	1.8%	+21.3%		46.2%	0.1%	+0.05%	+53.6%
130	78.8%	3.1%	2.1%	+18.9%	*107*	66.1%	3.3%	+0.1%	+33.7%
	82.9%	1.8%	1.8%	+15.1%		67.7%	3%	+0.1%	+32.0%
	84.4%	1.5%	1.8%	+13.6%		83.4%	1.5%	+0.2%	+16.3%
132	29.7%	2.7%	5.0%	+65.2%	*110*	67.2%	2.2%	+0.6%	+32.1%
	38.9%	3.4%	5.6%	+55.4%		74.5%	1.4%	+0.5%	+24.8%
	47.3%	4.1%	5.6%	+46.9%		83.5%	0.6%	+0.4%	+16.0%
134	69.2%	2.1%	1.6%	+29.1%	*125*	32.1%	0.5%	+0.1%	+67.7%
	73.4%	1.6%	1.6%	+24.8%		35.9%	2.0%	+1.7%	62.2%
	80.7%	1.0%	0.9%	+18.3%		60.8%	2.9%	+2.1%	+36.9%
147	72.5%	1.8%	2.9%	+24.4%	*141*	56.1%	4.4%	+3.6%	40.1%
	80.2%	1.8%	1.8%	+17.8%		73.6%	2.8%	+2.6%	+23.6%
	84.0%	1.2%	+1.4%	+14.4%		88.6%	0.5%	+0.3%	11.0%
Mean INT	66.9%	3.0%	+3.3%	+29.8%	*145*	49.6%	0.2%	+3.5%	+46.8%
	74.1%	2.4%	+2.8%	+23.1%		65.1%	0.3%	+4.9%	+29.8%
	78.5%	2.0%	+2.4%	+19.0%		77.6%	0.8%	+5.2%	+17.1%
Mean All	60.4%	2.0%	+2.1%	+37.5%	*146*	51.9%	0.5%	+1.3%	+46.6%
	66.7%	1.7%	+2.0%	+31.3%		60.9%	0.2%	+0.9%	+38.1%
	73.5%	1.5%	+1.7%	+24.7%		66.2%	0.2%	+0.9%	+32.8%
					Mean FP	55.2%	1.2%	+1.1%	+43.7%
						60.8%	1.2%	+1.3%	+37.8%
						69.5%	1.0%	+1.2%	+29.2%

Table 5.1: *Comparing a TVC with a L0 data cache of the same size. Data cache sizes shown are 128, 256 and 512 words (top to bottom)*

will find the data they need in the TVC). The TVCLI% indicates the negative impact the introduction of the TVC may have. The average memory latency is given by the following formula:

$$\mathit{TVCHIT\%} \times \mathit{TVC}_{lat} + \boldsymbol{\mathit{TVCLI\%} \times (\mathit{TVC}_{lat} + \mathit{MEM}_{lat})} + (1 - \mathit{TVCHIT\%} - \mathit{TVCLI\%}) \times \mathit{MEM}_{lat}$$

Where $\mathrm{TVC}_{\mathrm{lat}}$ is the latency of the TVC and $\mathrm{MEM}_{\mathrm{lat}}$ is the latency observed when accessing the rest of the memory hierarchy starting from the L1 data cache. Note that in the above formula, the latencies of the TVC and the L1 data cache add only when dependence status prediction incorrectly predicts that a load has a dependence visible from within the TVC (TVCLI% component). The L0HIT% columns report the *additional* fraction of loads that would hit in the L0 cache (compared to the TVC) and the L0LI% columns report the *additional* fraction of loads that would miss in the L0 and thus observe a latency increase. That is the L0HIT% rate is calculated by subtracting the TVCHIT% rate from the actual hit rate of the L0, while the L0LI% rate is calculated by subtracting the TVCLI% rate from the actual miss rate of the L0. (Note that the loads that miss in the TVC are not necessarily the same as the ones that miss in the L0.) The L0-HIT% column can be interpreted as the fraction of loads that will incorrectly not get hidden in the TVC (i.e., dependence status prediction incorrectly indicates that these loads will not find the data they need in the TVC), while the L0-LI% column can be interpreted as the fraction of loads that will correctly not get penalized when the TVC is used (i.e., dependence status prediction correctly indicates that these loads will not find the data they need in the TVC). The average memory latency in this case is as follows:

$$(\mathit{TVCHIT}\% + \mathit{L0HIT}\%) \times \mathit{L0}_{lat} + \boldsymbol{(\mathit{TVCLI}\% + \mathit{L0LI}\%)} \times \boldsymbol{(\mathit{L0}_{lat} + \mathit{MEM}_{lat})}$$

Where $\mathrm{L0}_{\mathrm{lat}}$ is the latency through the L0 data cache. Note that in this formula the latencies of the L0 and the L1 data cache add whenever loads do not have dependences that are visible through the L0 (TVCLI% + L0LI%). We make the following observations: (1) very few loads that hit in the L0 do not hit in the TVC, (2) very few loads will observe a latency increase with the TVC, and (3) a relatively large number of loads will miss in the L0 and thus observe a latency increase by the introduction of the L0 data cache. These results, suggest that the TVC can capture most of the loads that would hit in a traditionally organized L0 data cache, while avoiding the vast majority of load misses that would otherwise lead to increased load latencies.

The results of this section suggest that a relatively small data cache (e.g., 128 words) coupled with memory dependence predictor could be used to offer the benefits of servicing a large fraction of loads (i.e., 66.9% and 55.2% for the integer and floating-point programs respectively) without consuming L1 cache ports, while increasing load latency for 3.3% and 1.1% of all loads on the average for the integer and floating point codes respectively. In the worst case observed, the latency of 7.5% of all load was increased (099.go). For a traditional organization, that would place an 128-word data cache in series with the L1, 29.7% and 43.7% of all loads will have observed increased latency on the average and for the integer and floating point codes respectively.

5.4.1.1 Effects of Associativity on Prediction Accuracy

In this section we vary the associativity of the data cache component and measure how memory dependence status accuracy is affected. For the purposes of this study, we restrict our attention to a data cache component of 128 words. Table 5.2 compares a TVC and a L0 cache that are, from top to bottom: (1) direct mapped, (2) 2-way set associative, and (3) 4-way set associative. The same metrics as in Table 5.1 are used for this comparison.

We can observe that for smaller associativities, and as expected, the fraction of loads that can get hidden in a small data cache drops. In this environment the effectiveness of both the TVC and the L0 caches is reduced. The L0 caches exhibit much higher miss rates, which would translate in an increased number of loads getting penalized by the introduction of an additional caching level. In the case of the TVC caches, prediction accuracy also drops compared to a fully-associative data cache component. Fortunately, increasing the associativity results in higher accuracy. Compared to an L0, the TVC still captures the vast majority of loads that would hit in the L0. However, the differences are now higher. Moreover, the fraction of loads that the TVC incorrectly tries to hide from the L1 (TVC-LI% column) is also much higher, especially with

	TVCHIT%	TVCLI%	L0HIT%	L0LI%		TVCHIT%	TVCLI%	L0HIT%	L0LI%
099	50.8%	10.8%	+10.6%	+38.6%	*101*	30.2%	1.6%	+1.3%	+68.5%
	57.6%	9.0%	+9.1%	+33.3%		41.7%	2.4%	+1.5%	+56.8%
	61.4%	7.7%	+8.1%	+30.5%		49.1%	3.2%	+2.6%	+48.4%
128	61.3%	10.0%	+6.9%	+31.7%	*102*	41.7%	0.3%	+0.1%	+58.2%
	66.9%	10.4%	+7.3%	+25.8%		50.7%	0.5%	+0.0%	+49.2%
	72.5%	8.9%	+5.3%	+22.2%		64.3%	0.1%	+0.0%	+35.7%
126	55.2%	7.3%	+6.1%	+38.7%	*103*	44.8%	1.7%	+1.9%	+53.3%
	60.5%	5.6%	+5.4%	+34.2%		51.7%	1.1%	+1.6%	+46.7%
	62.9%	4.7%	+4.8%	+32.2%		52.9%	0.6%	+1.5%	+45.6%
129	59.0%	4.9%	+2.3%	+38.7%	*104*	34.6%	1.6%	+0.6%	+64.8%
	71.5%	2.5%	+1.3%	+27.2%		40.5%	1.5%	+0.5%	+59.0%
	73.9%	1.1%	+0.9%	+25.1%		45.1%	0.5%	+0.1%	+54.8%
130	62.1%	10.0%	+6.7%	+31.1%	*107*	42.2%	8.2%	+6.1%	+51.7%
	72.8%	7.1%	+3.5%	+23.7%		50.6%	8.9%	+5.0%	+44.5%
	77.0%	4.6%	+2.5%	+20.5%		61.9%	5.8%	+1.1%	+37.0%
132	26.8%	2.7%	+3.1%	+70.0%	*110*	54.9%	6.6%	+3.3%	+41.8%
	27.8%	2.3%	+3.1%	+69.0%		59.2%	3.4%	+1.5%	+39.4%
	28.6%	2.8%	+4.9%	+66.4%		62.9%	4.0%	+2.3%	+34.8%
134	58.9%	4.8%	+3.4%	+37.8%	*125*	31.6%	2.6%	+1.7%	+66.7%
	62.9%	4.1%	+3.0%	+34.0%		31.8%	1.8%	+1.3%	+66.8%
	65.1%	3.4%	+2.5%	+32.4%		31.1%	1.5%	+1.0%	+67.8%
147	63.5%	4.2%	+3.4%	+33.1%	*141*	53.6%	6.7%	+5.0%	+41.4%
	68.6%	2.7%	+2.6%	+28.8%		55.8%	5.7%	+3.7%	+40.5%
	70.9%	1.8%	+2.6%	+26.5%		56.9%	5.6%	+3.4%	+39.7%
Mean INT	54.7%	6.8%	+5.3%	+39.9%	*145*	40.5%	0.4%	+2.5%	+57.0%
	61.1%	5.5%	+4.4%	+34.5%		45.3%	0.4%	+2.8%	+51.9%
	64.0%	4.4%	+3.9%	+32.0%		46.2%	0.4%	+3.0%	+50.7%
Mean All	46.8%	4.8%	+3.8%	+49.3%	*146*	30.3%	2.9%	+3.9%	+65.7%
	52.9%	4.0%	+3.1%	+44.0%		36.0%	2.5%	+3.0%	+61.0%
	56.9%	3.2%	+2.7%	+40.4%		40.9%	1.9%	+2.1%	+51.0%
					Mean FP	40.4%	3.2%	+2.6%	+56.9%
						46.3%	2.8%	+2.1%	+51.6%
						51.1%	2.3%	+1.7%	+47.4%

Table 5.2: *Comparing an 128 word TVC with a L0 data cache of the same size for various associativities (direct mapped, 2-way and 4-way, top to bottom).*

the direct-mapped data cache component. However, this fraction is still small compared to the fraction of loads that would miss in an L0 (TVC-LI% + L0-LI%).

The results of this section suggest that while TVC effectiveness drops with decreased associativity, the TVC still captures most of the loads that would get hidden in an L0, while avoiding penalizing the majority of the loads that would see their latency increased had we used an L0 organization.

5.4.1.2 Effects of Block Size on Prediction Accuracy

So far we have assumed that the data cache component of either the TVC or the L0 is word-wide. In this section, we study how memory dependence status prediction and the effectiveness of the TVC and the L0 organizations varies when we increase the block size to 2, 4 and 8 words (8, 16 and 32 bytes respectively). For this experiment we use fully-associative data cache components of 128 words (512 bytes). Table 5.3 reports the results of these experiments. The same metrics as in Table 5.1 are used.

We can observe that the TVC still offers most of the benefits of using a small data cache to hide loads from the L1, while avoiding penalizing a large fraction of loads. Prediction accuracy is typically lower compared to a word-wide data cache component. The fraction that the TVC incorrectly does not hide (L0-HIT% column) is now higher, and in most cases increases with the block size. Similarly, the fraction of loads that the TVC incorrectly penalizes increases with the block size. The cause of this decrease in accuracy lies in the effect the increased block size has on the quality of the memory dependence status history of loads. Recall that we use the data cache component to detect dependences and build the history used by our predictor. When we use a larger block size and on a miss, additional data, beyond what is required by the access that missed are brought into the data cache. As a result of the spatial locality found in typical programs, the additional data cause the detection of memory dependences that do not necessarily correspond to real dependences in the program. In this case the real memory dependence status history of loads is convoluted by the prefetching effect of larger block sizes. A similar decrease in dependence status history quality is introduced when due to the block size used, data that would otherwise reside in the data cache is evicted. In this case, some dependences that exist in the program are not detected.

5.4.2 Store Dependence Status Prediction Accuracy

In this section we report results on the accuracy of a WAW (output) dependence status predictor for stores as a function of the data cache size used. The data cache sizes and assumptions are the same as the ones we used in Section 5.4.1. The same applies to the memory dependence status predictor used. We restrict our attention only to fully-associative data cache components and do not study the effects of decreased associativity or of increased block sizes.

For WAW dependence detection purposes, each entry of the TVC data cache was augmented to record the store that wrote to that memory location, if any. A single store could be recorded per TVC word. Figure 5.1, part (a) reports the fraction of stores that are correctly predicted by the dependence status predictor. In all programs and all data cache sizes studied, prediction accuracy was higher than 80%. Part (b) of the figure reports the fraction of stores that are correctly predicted and are killed in the TVC data cache. Part (d) of the figure compares the fraction of stores that are killed in the TVC, with the fraction of the stores that would have been killed in data cache of the same size. We can observe that most of the stores that would have been killed in an L0 data cache, are also killed in the TVC. This observation is also supported by the results of part (d), where we report the stores that are killed in the TVC as a fraction of the stores that would have been killed in an L0 of the same size. Finally, part (c) reports the fraction of stores that are incorrectly predicted that they will not be killed in the TVC. These are the stores that the TVC will incorrectly not hide from the L1 data cache. We can observe, that while this percentage is in most cases low, for some programs and depending on the data cache component used it is relatively large. In the worst observed case, (145.fpppp

	TVCHIT%	TVCLI%	L0HIT%	L0LI%		TVCHIT%	TVCLI%	L0HIT%	L0LI%
099	53.6%	6.9%	+8.1%	+38.3%	*101*	77.5%	0.3%	+0.0%	+22.4%
	48.5%	7.9%	+8.4%	+43.1%		78.2%	1.1%	+6.3%	+15.5%
	44.1%	7.2%	+8.2%	+47.7%		82.2%	4.8%	+0.7%	+17.1%
124	43.6%	1.8%	+2.6%	+53.8%	*102*	64.1%	1.7%	+13.0%	+22.9%
	49.7%	4.7%	+2.1%	+48.2%		84.0%	6.6%	+0.1%	+15.9%
	53.1%	4.5%	+4.2%	+42.8%		87.5%	4.1%	+0.0%	+12.5%
126	62.3%	4.7%	+3.7%	+33.9%	*103*	71.5%	0.5%	+0.9%	+27.6%
	64.6%	6.3%	+5.6%	+29.9%		72.4%	0.6%	+6.6%	+21.0%
	66.7%	5.8%	+5.1%	+28.2%		79.7%	3.7%	+1.4%	+18.9%
129	73.7%	4.7%	+1.4%	+24.9%	*104*	70.0%	0.4%	+0.1%	+29.9%
	68.6%	5.1%	+1.7%	+29.6%		71.2%	0.5%	+11.2%	+17.6%
	58.8%	6.3%	+4.2%	+37.0%		87.5%	5.8%	+0.6%	+11.9%
130	66.4%	5.7%	+3.3%	+30.3%	*107*	82.6%	1.7%	+0.1%	+17.4%
	59.4%	7.7%	+6.1%	+34.5%		86.2%	3.6%	+3.8%	+10.0%
	61.1%	6.1%	+6.0%	+32.9%		72.8%	6.0%	+0.9%	+26.3%
132	56.4%	7.4%	+7.9%	+35.7%	*110*	77.7%	1.8%	+0.8%	+21.6%
	67.3%	7.9%	+5.1%	+27.6%		80.0%	2.7%	+4.3%	+15.7%
	70.0%	6.1%	+4.1%	+25.8%		82.9%	3.5%	+1.8%	+15.3%
134	66.3%	3.4%	+1.8%	+31.8%	*125*	64.0%	0.5%	+0.1%	+35.9%
	62.9%	3.7%	+2.6%	+34.6%		67.9%	0.4%	+11.6%	+20.6%
	64.9%	3.8%	+2.4%	+32.7%		85.0%	7.1%	+1.0%	+13.9%
147	64.6%	3.0%	+2.1%	+33.3%	*141*	68.7%	1.8%	+1.1%	+30.2%
	68.1%	4.6%	+4.1%	+27.8%		74.1%	1.6%	+3.5%	+22.4%
	66.8%	4.2%	+3.6%	+29.6%		78.1%	3.1%	+2.0%	+19.9%
Mean INT	60.8%	4.7%	+3.9%	+35.3%	*145*	65.4%	0.1%	+3.1%	+31.5%
	61.1%	5.9%	+4.4%	+34.4%		69.7%	0.1%	+3.2%	+27.1%
	60.7%	5.5%	+4.7%	+34.5%		72.7%	0.1%	+3.4%	+23.9%
Mean All	66.5%	2.6%	+2.8%	+30.7%	*146*	68.6%	0.5%	+0.8%	+30.6%
	69.1%	3.7%	+5.1%	+25.8%		71.1%	1.2%	+5.6%	+23.3%
	71.8%	4.8%	+2.8%	25.4%		77.9%	3.6%	+1.3%	+20.8%
					Mean FP	71.0%	0.9%	+2.0%	+27.0%
						75.5%	1.8%	+5.6%	+18.9%
						80.6%	4.2%	+1.3%	18.1%

Table 5.3: *Comparing an 128 word TVC with a L0 data cache of the same size for various block sizes (2, 4 and 8 words, top to bottom).*

and 2K-word data cache component), as much as 14% of stores are incorrectly identified. For the data cache component of 128-words, this fraction is below 10%, and for most programs below 7.5%.

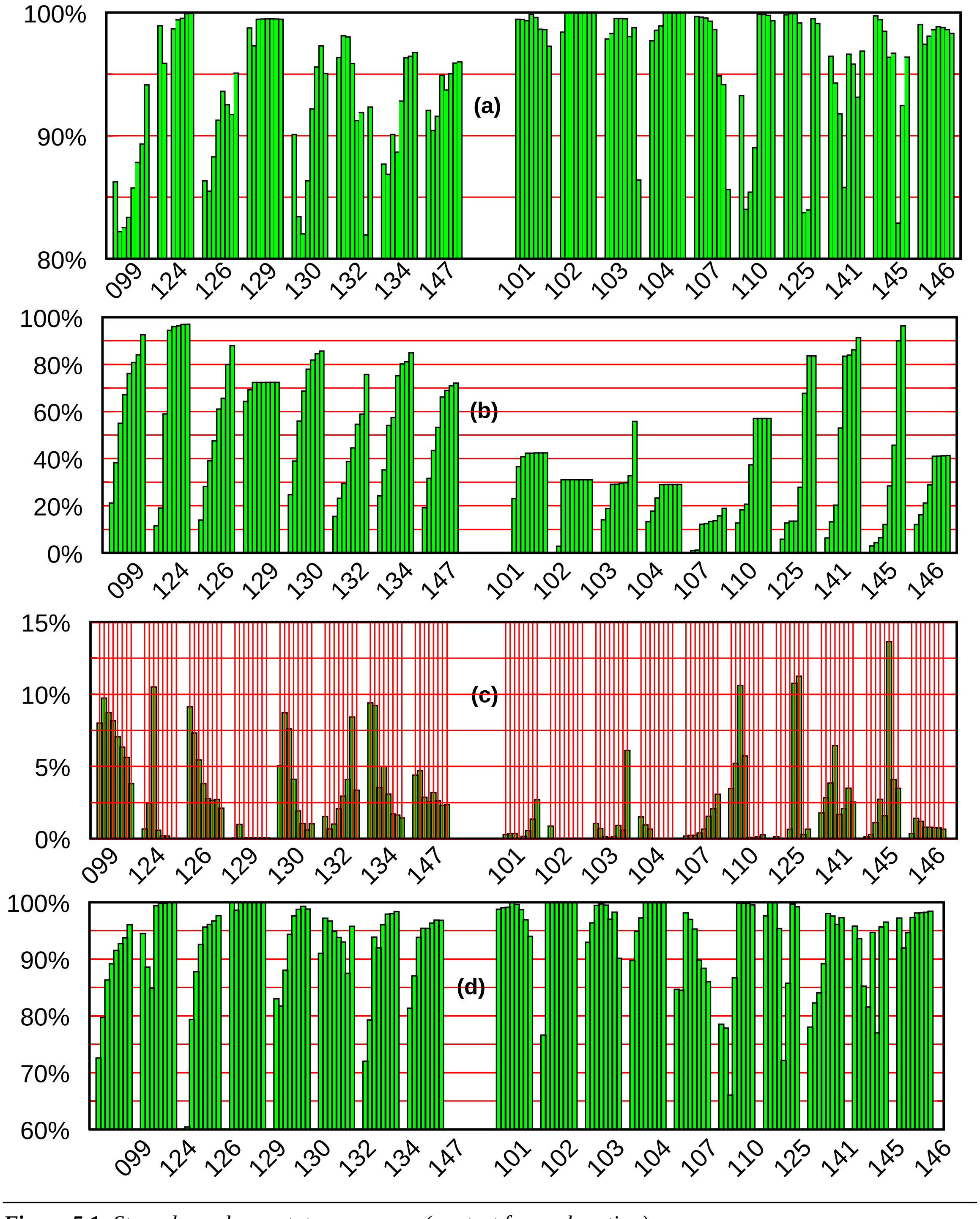


Figure 5.1: *Store dependence status accuracy (see text for explanation).*

The result of this section suggest that while WAW dependence status prediction is high, a noticeable fraction of stores may have their dependence status incorrectly identified.

5.5 Summary

In this chapter, we were motivated by the large fraction of loads that read a recently accessed memory location and of stores that are quickly killed by another store. We observed that these loads and stores could be serviced using a relatively small data cache reducing the bandwidth requirements imposed on the rest of the memory hierarchy. Unfortunately, we also noted that if a small cache was introduced in series with the L1 data cache, as it is done in traditional memory hierarchies, it would also result in a latency increase for all loads that do not hit in it.

To get the best of both worlds, that is to hide from the rest of the memory hierarchy those loads and stores that could be serviced with a relatively small data cache, while not increasing the latency of all other loads, we proposed using the Transient Value Cache, a memory hierarchy component placed at the L1-processor interface. The novelty of the TVC lies in its ability to adapt and appear either in-series with or in-parallel to the L1 data cache. The decision on whether to appear in-series or in-parallel is taken using a history-based, memory dependence status predictor. Specifically, before a load or a store is sent to the memory hierarchy, a prediction is made on whether it will find the data it needs in the TVC (if it is a load), or whether it will get killed in TVC (if it is store). If so, the load or store is send only to the TVC, in which case L1 data cache ports can be used to service other memory requests. If the prediction correctly indicates that a load will not find the data it needs in the TVC, it is send directly to the L1 data cache thus avoiding an increase in latency.

We have performed a trace driven study of the accuracy of TVC mechanism varying the size of the data cache component and found that prediction accuracy for loads is high, often above 90%. For a realistic data cache component of 512 bytes (128 words) we have found that the TVC approach could hide 66.9% and 55.2% of all loads on the average and for the integer and floating point codes respectively. More importantly, only 3.0% and 1.2% of all loads would observe a latency increase as the result of erroneous memory dependence status prediction. In contrast, a traditional memory organization that would place the 128-word data cache in series with the L1, would hide only an additional 3.3% (integer) and 1.1% (floating-point) of loads, while increasing the latency of an additional 29.8% (integer) and 43.7% (floating-point) of all loads. We also studied the effects of reduced associativity and of increased block size and found that while prediction accuracy drops, the TVC still hides the majority of loads that could be hidden in such data caches while avoiding penalizing a large fraction of other loads. While our evaluation is preliminary, it does provide an indication of the utility of the proposed technique. Further investigation is required, as ultimately the effectiveness of the TVC approach can be determined only when its effects on performance are studied.

Finally, we investigated using WAW dependence status prediction to predict whether a store will get killed if allocated in the data cache component of the TVC and found that prediction accuracy is high, although not as high as it is for RAW and RAR dependence status prediction for loads.

Chapter 6

Conclusion

In this last chapter, we first present a summary of our findings in Section 6.1. Finally, in Section 6.2 we conclude by pointing to a number of research directions that may extend this work or that stem from our experience gained while investigating memory dependence prediction and its applications.

6.1 Summary

As projected, advances in manufacturing technology will soon offer us the ability to construct single-chip devices containing hundreds of millions if not a billion of transistors [75]. This vast amount of on-chip resources provides us with both an opportunity and a challenge. The opportunity exists to try out techniques and build mechanisms that were not previously practical. Yet, making judicious use of this opportunity is bound to be challenging, more so if the underlying design tradeoffs change as many currently predict (e.g., wire transmission speed may limit how much area of the on-chip resources may be reachable within a clock cycle, or power may constrain the amount of circuitry we can operate [57].

In this context, it is not our contribution that we provide definite answers on how to best make use of this forthcoming opportunity. Rather, with this thesis we contribute a new tool, *memory dependence prediction*, that might be useful in developing techniques that may utilize these resources in a useful and hopefully better way. In particular, we have introduced two forms of memory dependence prediction: (1) *memory dependence status prediction* and (2) *memory dependence set prediction* or, simply *memory dependence prediction*. Memory dependence status prediction is technique that allows us to guess with high accuracy whether a load or a store will experience a dependence of a particular type (for this technique the exact dependence is not important, only whether a dependence exists is). Memory dependence prediction is a specialization of memory dependence status prediction where not only we guess whether a load or store has a dependence of a particular type, but also which this dependence (or dependences) is. Both techniques operate: (1) by observing the memory dependence behavior of instructions through the memory address name space, (2) by associating relevant memory dependence information with the corresponding static instructions (i.e., with the PC of a store or a load), and (3) by using the recorded information to predict the relevant memory dependence information the next time a load or a store is encountered without actual knowledge of the addresses being accessed.

In Chapter 2 we have provided experimental evidence in support of the efficacy of memory dependence prediction. In particular, we have identified that typical programs exhibit high regularity in both their memory dependence status and their memory dependence stream. We have shown that: (1) if at some point during execution, a load or a store experiences a dependence of a particular type, chances are that the next time the same instruction is encountered, it will experience a dependence of the same type as before, and (2) if at some point during execution, a load or a store experiences a particular memory dependence, then chances are the next time the same instruction is encountered it will experience the same memory dependence once again. These results suggest that past dependence behavior is a good indicator of forthcoming memory dependence behavior. Which, in turn suggests that memory dependences may be amenable to history-based prediction. Moreover, we have shown that the working set of stores and loads with dependences is relatively small (i.e., less then 4K for virtually all programs studied in this thesis). This observation suggests that structures of reasonable size will probably be sufficient to record memory dependence history information for the purposes of memory dependence prediction.

In itself memory dependence prediction is only useful in providing advance information of the memory dependence behavior of loads and stores. Techniques are required to make use of this information in some practical manner and for a practical purpose. To this extent, in this thesis we presented three micro-architectural techniques: (1) *dynamic speculation and synchronization of memory dependences,* (2) *speculative memory cloaking and bypassing,* and (3) *transient value cache.* The first two techniques aim at reducing the observed memory access latency, while the third technique aims primarily at providing support for multiple memory requests per cycle. In the next three sections we briefly describe each of these techniques along with a summary of the key results.

6.1.1 Dynamic Speculation and Synchronization of Memory Dependences

The first technique we proposed aims at supporting highly parallel execution of loads and stores in the presence of ambiguous memory dependence. In particular, this technique aims at mimicking the operation of an ideal load/store scheduler that has perfect in advance knowledge of all memory dependences. This technique operates by: (1) predicting those load-store pairs whose unrestricted execution will result in a memory dependence violation, and (2) delaying load execution only as long as it is necessary to avoid the memory dependence violation by enforcing synchronization with the appropriate store. Dynamic speculation and synchronization of memory dependences was motivated by the following observations: (1) naive memory dependence speculation—that is, always executing a load as soon as its address becomes available—offers significant performance improvements over not speculating memory dependences, and (2) the net penalty of erroneous dependence speculations when naive memory dependence speculation is used can become significant especially as the size of the instruction window increases. We have demonstrated that the two aforementioned observations hold under the following two processing models: (1) a centralized, continuous window processor model (typical modern superscalar) when loads cannot inspect preceding store addresses to determine whether memory dependences exist, and (2) a distributed, split-window processor model independently on whether store address information is available to loads prior to accessing memory.

Using timing simulations we have shown that memory dependence speculation and synchronization can reduce memory dependence mispeculations by at least an order of magnitude over naive memory dependence speculation. Moreover, we have studied three possible implementations of our proposed technique, one centralized and two distributed ones which we assumed to be increasingly easier and less costly to implement. We found that these three mechanisms can sustain average performance improvements of (1) 28.0% (integer) and 15.43% (floating point), (2) 22.4% and 10.9%, and (3) 20.2% and 9.0%, respectively and over naive memory dependence speculation. All three mechanisms performed close to what would be possible had we had perfect, in advance knowledge of all memory dependences, that is 31.21% (integer) and 17.35% (floating-point). This result suggests that our technique is quite successful in attaining both goals of memory dependence speculation.

We also studied memory dependence speculation under a centralized, continuous window processor model equipped with an 128-entry window and capable of issuing up to 8 instructions in parallel (4 loads or stores) and found that: (1) naive memory dependence speculation in almost all cases improved performance over no speculation. The actual improvements were heavily influenced by whether preceding store address information was available to loads prior to accessing memory. We found that when this information was not available, naive speculation improved performance over no speculation by 29.6% (integer) and 113% (floating-point). When store address-information was available, naive speculation improved performance over no speculation by 4.6% and 5.3% provided that inspecting store address did not increase load latency. (2) When store addresses were made available for load inspection, mispeculations were virtually non-existent. (3) When store addresses were not available for load inspection the net penalty of dependence mispeculation was significant. In this case, had we had perfect knowledge of all memory dependences, performance could improve by as much as 20.9% (integer) and 20.4% (floating-point) over naive speculation. An implementation of our speculation/synchronization mechanism offers performance improvements which are very close to those ideally possible: 19.7% (integer) and 20.4% (floating-point).

6.1.2 Speculative Memory Cloaking and Bypassing

Speculative memory cloaking and bypassing aims at reducing observed memory access latency. In this technique memory dependence prediction is used to build direct, albeit speculative communication links between loads and stores that access a common memory location, without actual knowledge of the memory location being accessed. This is done so loads may obtain a speculative value without having to wait until the traditional, address-based memory hierarchy is accessed. In particular, communication links are established between: (1) a load and store, when it is highly probable that the load is going to be reading the value written by the store, and (2) two loads, when it is highly probable that both loads will be accessing the same memory location.

Using trace driven simulations we have demonstrated that a straightforward model of cloaking/bypassing is capable of providing correct values for about 70% (integer) and 50% (floating-point) of all loads (coverage). Moreover, the same mechanism provided incorrect values for only 2.5% (integer) and 0.4% (floating-point) of all loads (mispeculation rate). We also compared cloaking/bypassing with a straightforward last-value load value prediction mechanism and found that cloaking/bypassing offers superior coverage and mispeculation rates. However, we have also shown that the two techniques are complementary as cloaking/bypassing correctly handles some loads that load value prediction does not and vice versa. Using timing simulations of an aggressive 8-way superscalar, with an 128-entry window that also uses naive memory dependence speculation we found that an implementation of cloaking/bypassing offered performance improvements of 6.03% (integer) and 4.9% (floating-point) when selective invalidation was possible (i.e., invalidating and re-executing only those instructions that used erroneous data). We also found that when combined with squash invalidation (i.e., invalidating and re-executing all instructions starting from the oldest one that used erroneous data) cloaking/bypassing resulted in performance degradation for virtually all programs. Finally, we studied two combinations of last-value prediction and cloaking/bypassing and found that while performance improved somewhat for most programs the performance improvements compared to a stand-alone cloaking/bypassing mechanism were barely noticeable.

6.1.3 Transient Value Cache

Finally, we proposed the transient value cache, a technique aiming at supporting multiple memory requests per cycle. For this technique we were motivated by two empirical observations about load and store behavior. We observed that: (1) a relatively large fraction of loads read a value that was either recently written by a store, or was recently read by another load, and (2) a large fraction of stores get killed (i.e., overwritten) by a subsequent, yet close in time store. In particular, we have shown that roughly 70% (integer) and 56% (floating-point) of all committed loads, access a memory location that is within the 128 most recent memory locations (word granularity) read by a preceding load or written to by a preceding store. Moreover, 55% (integer) and 23% (floating-point) of all committed stores, get overwritten by a subsequent store that accesses a memory location that is within the last 128 most recently accessed memory locations (word granularity). Based on the aforementioned observation we further noted that it would be possible to hide all those memory references using a relatively small data cache. However, had we placed this data cache in series with the L1 cache (as it is done in traditional memory hierarchies) the latency of all other loads that would not hit in the newly introduced data cache would now increase.

To get the best of both worlds, that is to hide all those loads and stores that would be hidden in a small data cache, while avoiding increasing the latency of other loads, we proposed the transient value cache. In our proposal, a small and preferably narrow cache is introduced in the processor-L1 data cache interface. Moreover, memory dependence status prediction is used to decide whether the newly introduced data cache should appear in-series with or in-parallel to the L1 data cache on a per access basis. Specifically, the TVC appears in-series with the L1 data cache for those loads that will likely find the data they need in the TVC and for those stores that will likely get overwritten in the TVC. For all other loads and stores the TVC appears in-parallel to the L1 data cache. Provided that memory dependence status prediction is correct, the potential benefits of the TVC approach are: (1) the loads that hit in the TVC and the stores that get killed in the TVC are hidden from the L1, freeing up L1 data port resources to be used by other loads and stores, (2) the latency of loads that would not hit in the TVC is not increased.

Using trace-driven simulation, we demonstrated that a TVC comprising a fully-associative data cache component of 128-words (512 bytes) and a memory dependence status predictor that utilizes 2-bit confidence counters, could hide roughly 67% (integer) and 55% of all committed loads from the rest of the memory hierarchy, while only 3% (integer) and 1.2% (floating-point) of all loads would observe increased latency by the introduction of the TVC. For a traditionally organized (i.e., in series with the L1 data cache) data cache component of the same size, only an additional 3.3% (integer) and 1.1% (floating-point) of all committed loads would be hidden from the L1 data cache, while an additional 29.8% and 46.6% of all loads would observe a latency increase. We also studied the effects of increased block size and decreased associativity and found that while prediction accuracy dropped, the TVC could still hide the vast majority of loads that would have been hidden in a traditionally organized cache of the same size, while avoiding to increase the latency for most of the loads that would not hit in such a data cache. Our results serve as an indication of the potential utility of the TVC approach. However, further investigation is required to determine the performance impact of the TVC approach.

Finally, we observed that the possibility exists to combine the TVC with the cloaking/bypassing mechanism resulting in a mechanism that can both reduce the latency of accessing some memory values and hide these accesses from the rest of the memory hierarchy.

6.2 Future Directions

Throughout this work our goals were: (1) to provide sufficient evidence in support of our observations about the regularity that exists in the memory dependence stream of ordinary programs, and (2) to present techniques that can exploit this regularity for some practical purpose, also providing sufficient evidence about their utility. In our opinion, this work represents only a first step toward a thorough investigation of the dynamic memory dependence behavior programs and of potential applications of memory dependence prediction. Certainly, our evaluation and treatment of all three techniques can be extended in many ways (for example, by considering alternative implementations, other prediction structures and by varying the configuration parameters). We will not attempt to enumerate these possibilities. Rather, in the few remaining sections we will briefly comment on some general research directions that are either directly related to memory dependence prediction or that stem from the experience we gained while working on this topic.

6.2.1 Correlating Memory Dependence Behavior with Program Elements and Data Structures

In our treatment of memory dependence behavior presented in Chapter 2 we restricted our analysis on what is the memory dependence behavior of the programs studied and, for the most part did not attempt to explain why behavior is such. Further investigation could focus on correlating memory dependence behavior with the program elements and the data structures that give rise to it. Such an investigation will improve our understanding of why memory dependences behave in certain ways. Hopefully, this information will be proven helpful in stimulating other applications of memory dependence prediction as it may for example, expose previously unidentified regularities in program behavior. It may also expose weaknesses in the way programs express a desired action hinting to better ways of expressing or of performing such actions (similar to what was the case for memory inter-operation communication and data-sharing).

Such an investigation may also focus on determining whether regularities exist in a coarser level. Even though we focused on the relationships formed when individual loads and stores access memory, it is likely that similar relationships may among exist parts of the code. For example, we may find that program operation is such that one function (or part of the program) generates data that some other function is consuming (consider a compiler where one function generates a parses a statement into an internal form which is then used by another function that scans through this form to generate statements). Such coarse level “dependence” information might be useful, for example, in a “smart” memory hierarchy in which a set of data is associated with the parts of the code that is generating or manipulating it. This information could be useful in managing data placement in the memory hierarchy. For example, once one data element of a set of data previously tagged is accessed the rest of the data could be moved in faster storage structures in hope that they will also be accessed.

6.2.2 Interaction with the Compiler

Throughout this work we have focused on dynamic, architecturally invisible techniques. Nevertheless, it might be possible to expose some of the mechanisms we propose to the compiler through ISA extensions or to rely on compiler provided information to improve the characteristics of our techniques. With such an approach we may either hope to improve the coverage or accuracy of our techniques and/or reduce the amount of physical resources required. For example, in memory dependence speculation and synchronization we may rely on compiler hints in order to avoid speculating some loads. Or, we may expose the memory dependence prediction information to the software so that a decision can be made on whether to execute a schedule that speculates a load or an alternative one that does not. Similarly, in cloaking/bypassing we may rely on compiler provided synonyms and finally, in the transient value cache we may rely on compiler provided memory dependence status information.

There are two important challenges in this context: (1) whether the compiler can provide accurate enough information, and (2) whether any of the improvements so obtained justify changing the architectural interface. We next comment on both issues. In all our techniques we relied on dynamically collected information. Whether this information can be accurately provided by the compiler is an open question. Certainly, profile information can be useful in this context. However, some of the underlying phenomena may be dynamic in nature and more importantly they may heavily depend on the particular hardware configuration. For example, the dependences that are mispeculated are not necessarily the same for processors with different instruction windows. One of the potential advantages of the techniques we proposed is that they are architecturally invisible. As such, they do not require modification of existing applications. Moreover, another potential advantage of our techniques is that they can be designed to fit the particular processor implementation. These two potential advantages may be lost if we choose to rely on a compiler directed approach.

Another important consideration with using compiler information to improve upon our mechanisms is whether we could obtain similar benefits by pre-existing software only approaches. For example, if we rely on the compiler for hints on whether to execute a load with ambiguous dependences, we may get performance that is similar to that possible with software-only memory dependence speculation. For these reasons a more fruitful approach could be to use the compiler to improve upon the hardware-only implementations, whenever this is desired. The challenge then is once again determining whether the additional benefits justify the approach.

A study of a compiler directed approach to improving the accuracy of memory dependence speculation and to providing synonyms for the purposes of speculative memory cloaking was reported in [69]. The focus of this work is on demonstrating that a compiler directed approach can provide similar performance benefits to those possible with a hardware-only approach.

6.2.3 Memory Communication and Sharing in Distributed Environments

Underlying cloaking and bypassing is the general idea that while we have chosen to express some program actions via the traditional address based interface, in an actual implementation we do not necessarily have to perform these actions in that way. In particular, we may devise techniques to identify what the intended action is and mechanisms to perform it faster or in a better way.

While we have investigated speculative memory cloaking and bypassing in the context of a single centralized and continuous window processor, in principle the same techniques and concepts might be applicable in other environments and processing models. In fact, in the case of cloaking and bypassing the potential benefits could be higher when the latency of accessing a value is increased. Such environments are, for example, multi-processors executing either sequential (as in Multiscalar) or explicitly parallel programs. In those environments programs also read and write memory values, and inter-operation communication and data-sharing occurs. More importantly, in such environments other inefficiencies of the address-based memory interface may be present which are not there in the sequential, centralized execution models we considered in Chapter 4 (e.g., data speculation or coherence related overheads [31, 32]). For example, in a multi-processor environment we may use memory dependence prediction of RAR dependences to speculatively propagate values read by one processor to another. Similarly, we may use RAW dependence prediction to speculatively propagate the value written by one processor to others converting the traditional “pull” approach (load asks for a value and memory reacts) to accessing memory values into an active “push” one (value is sent to the load). In fact, Kaxiras has recently studied such techniques in the context of explicitly-parallel programs and presented evidence in support of their utility [43]. Other techniques may be possible. However, much work remains in determining whether sufficient regularity exists in the memory dependence or more appropriately in the load/store relationship stream under these assumptions (For example, there is nothing to say that the memory dependence behavior observed on a lock will be sufficiently regular —the exact dependence pattern may be heavily data dependent).

Other opportunities for applications that use memory dependence information and prediction may also exist if the current predictions about the relative speed of wires and logic of semiconductor devices are proven correct [57]. In such an environment, memory dependence prediction may also be proven useful in combating the increased latencies that will be experienced in forthcoming, large-integration chips. For example, memory dependence prediction may be used to localize communication as much as possible by assigning those loads and stores that communicate to the same processing unit.

6.2.4 Support for Selective Invalidation and Data Speculation Resolution

In Chapter 4 we have seen that cloaking/bypassing is advantageous when selective invalidation and fast data speculation resolution is possible. In fact, most value speculative techniques proposed today assume that such mechanisms are possible. Yet, we noted that whether these two mechanisms can be implemented is still an open question. For this reason further investigation is required. Such an investigation may proceed in two directions: (1) We may focus on determining how tolerant are our techniques (or other value speculative techniques) to the speed and aggressiveness of the invalidation and data speculation resolution mechanisms. (For example, we assumed that a data speculative register dependence chain of any length can be resolved in a single cycle. It might be the case that limiting the length of the dependence chains that can be resolved in a single cycle does not severely impact overall performance.) Such an investigation will help in determining whether we really need ideally accurate and fast invalidation and data speculation resolution mechanisms. (2) In parallel we may also seek to develop actual mechanisms that can be implemented in a practical.

Moreover, our focus on this work was on demonstrating that our techniques can be used to improve the performance of highly optimized, wide-issue, long instruction window processors. It would be interesting to study whether our mechanisms can help a less aggressive processor to attain similar or better performance. If so, it is then interesting to study whether a combination of a less aggressive processor and of our techniques or other speculative techniques results in less complex, faster implementations.

6.2.5 Operation Prediction

Finally we comment on a research direction that stems from the experience we gained while working on memory dependence prediction. While the specific target of each of the three applications we presented is different their operation is similar: they all try to somehow predict in-advance what the program will do, and then use this information to optimize operation. Many other techniques that utilize some form of prediction exist (for example value, address and branch prediction). Most of these existing proposals to exploit regularities in program behavior focus in regularity that exists in the products of instruction execution and not so in the methods used to produce these results. For example, branch prediction observes the direction branch instructions follow with no regard to how this decision is being made. Similarly, value prediction observes the value stream produced or read by an instruction. Moreover, much of current proposals that aim at improving the accuracy of various predictors focus on methods to refine the history-information used trying to detect patterns in the stream of events being predicted (e.g., pattern based predictors [15, 12]). The underlying operation of such predictors however remains pretty much the same: we associate a prediction with a sequence of preceding events that led to it (for example, after branch A was taken 10 times, branch B was also taken).

However, the fact that memory dependences are predictable provides an indication that what actions (i.e., instructions) a program follows to produce its desired effects are not also random. Other evidence in support of this observation also exists [60, 51, 73, 24]. This observation hints to another direction of improving prediction accuracy and ultimately of building computing systems that are able to on-the-fly tailor their operation to better fit the currently executing program and the underlying semiconductor technology: that of studying what actions programs follow to produce their results and if possible using these very same actions to in-advance predict what the program will do. Our intuition is that often the sequence of actions used to produce a result is more regular than what this function produces (e.g., adding two arrays to produce a third, or traversing a linked list).

An example of a potential application is *operation prediction* where the predictor does not directly predict a desired information (e.g., which way a branch will go, or which value an instruction is going to produce) but rather, it predicts what sequence of operations will produce the desired information (for example, this load reads a value which is then compared with a constant to determine which direction a subsequent branch will follow). For operation prediction we may still rely on history-based prediction techniques. In this case, instead of building history that relates to the products of instruction execution, we build history that relates to the actions followed. Combined with other prediction techniques, operation predictors could, for example, be used to improve upon the accuracy of existing control flow predictors, value predictors or for prefetching memory data. In the case of branch prediction, an operation predictor could be used to pre-calculate the direction a branch will follow. Similarly, in the case of value prediction, an operation predictor can be used to pre-calculate the value an instruction is going to produce potentially long before the processor has had a chance to even fetch the corresponding instruction (note that the processor is limited the following factors: it has to fetch *all* instructions, it has maintain program semantics, and it has no notion of which part of the computation (slice) is more critical than others). In a sense, an operator predictor can be viewed as a generalized predictor which can tailor its prediction algorithm according to program behavior.

Bibliography

Conference/Publication Abbreviations

ASPLOS	International Symposium on Architectural Support for Programming Languages and Computer Architecture.
COMPCON	IEEE International Computer Conference
HPCA	International Symposium on High-Performance Computer Architecture
ICDD	International Conference on Computer Design
ICS	International Conference on Supercomputing
ISCA	Annual International Symposium on Computer Architecture.
ISSCC	International Solid-State Circuits Conference
MICRO	International Symposium on Microarchitecture
PACT	International Conference on Parallel Architectures and Compilation Techniques
PLDI	Conference on Programming Language Design and Implementation

References

[1] *PowerPC 620 RISC Microprocessor Technical Summary*. IBM Order number MPR620TSU-01, Motorola Order Number MPC620/D, October 1994.

[2] S. G. Abraham, R. A. Sugumar, D. Windheirser, B. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proc. on MICRO-26*, November 1993.

[3] D. Adams, A. Allen, R. Flaker J. Bergkvist, J. Hesson, and J. LeBlanc. A 5ns store barrier cache with dynamic prediction of load/store conflicts in superscalar processors. In *Proc. ISSCC*, February 1997.

[4] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[5] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Fast address calculation. In *Proc. ISCA-22*, June 1995.

[6] T. M. Austin and G. S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proc. ISCA-19*, May 1992.

[7] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proc. MICRO-28*, November 1995.

[8] T. M. Austin, T. N. Vijaykumar, and G. S. Sohi. Knapsack: A zero-cycle memory hierarchy component. Technical Report 1189, Computer Sciences Dept., University of Wisconsin-Madison, November 1993.

[9] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. Supercomputing '91*, 1991.

[10] U. Banerjee. *Dependence Analysis for Supercomputing*. Boston, MA: Kluwer Academic Publishers, 1988.

[11] P. Bannon and J. Keller. Internal architecture of alpha 21164 microprocessor. In *COMPCON'95*, March 1995.

[12] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[13] S. E. Breach. *Design and Evaluation of a Multiscalar Processor, in preparation*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, December 1998.

[14] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proc. MICRO-27*, pages 181–190, December 1994.

[15] I-Cheng K. Chen, J. T. Coffey, and Trevor N. Mudge. Analysis of Branch Prediction via Data Compression. In *Proc. ASPLOS-VII*, October 1996.

[16] W. Y. Chen. *Data Preload for Superscalar and VLIW Processors*. Ph.D. thesis, University of Illinois, Urbana, IL, 1993.

[17] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. ISCA-25*, June 1998.

[18] J. Dennis. Data Flow Supercomputers. *IEEE Computer*, November 1980.

[19] D. R. Ditzel and H. R. McLellan. Register Allocation for Free: The C Machine Stack Cache. In *Proc. ASPLOS-I}*, April 1982.

[20] K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proc. ISCA-24*, June 1997.

[21] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. In *IBM journal on research and development, 37(4)*, July 1993.

[22] J. R. Ellis. *Bulldog: A Compiler for a VLIW Architecture*. Ph.D. thesis, Yale University, February 1985.

[23] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presense of function pointers. In *Proc. SIGPLAN PLDI*, June 1994.

[24] A. Farcy, O. Temam, and R. Espasa. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. MICRO-31*, December 1998.

[25] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc. MICRO-30*, December 1997.

[26] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, November 1993.

[27] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[28] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proc. ASPLOS VI*, pages 183–193, October 1994.

[29] M. Gell-Mann. *The Quark and the Jaguar*. W. H. Freeman and Comparny, New York, 1994.

[30] M. Golden and T. Mudge. Hardware support for hiding cache latency. In *CSE-TR-152-93, University of Michigan, Dept. Of Electrical Engineering and Computer Science*, February 1991.

[31] Sridhar Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proc. HPCA-4*, February 1998.

[32] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. ASPLOS-VIII*, October 1998.

[33] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions in a processor capable of dispatching, issuing and executing multiple instructions in a single processor cycle, US Patent 5,615,350, filed on Dec. 1995, March 1997.

[34] G. J. Hinton, R. W. Martell, M. A. Fetterman, D. B. Papworth, and J. L. Schwartz. Circuit and method for scheduling instructions by predicting future availability of resources required for execution, US Patent 5,555,432, filed on Aug. 19, 1994, September 1996.

[35] A. S. Huang and J. P. Shen. A Limit Study of Local Memory Requirements Using Value Reuse Profiles. In *Proc. MICRO-28*, December 1995.

[36] A. S. Huang and J. P. Shen. The intrisinic bandwidth requirements of ordinary programs. In *Proc. ASPLOS-VII*, October 1996.

[37] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proc. ISCA-21*, May 1994.

[38] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *COMPCON'95*, 1995.

[39] W. W. Hwu and Y. N. Patt. Checkpoint Repair for High-Performance Out-of-Order Execution Machines. *IEEE Transactions on Computers*, C-36(12):1496–1514, December 1987.

[40] Q. Jacobson, S. Bennett, N. Sharma, and J. Smith. Control Flow Speculation in Multiscalar Processors. In *Proc. HPCA-3*, February 1997.

[41] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proc. MICRO-31*, December 1998.

[42] G. Kane. *MIPS R2000/R3000 RISC Architecture*. Prentice Hall, 1987.

[43] S. Kaxiras. *Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, August 1998.

[44] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. In *Digital Semiconductor, Digital Equipment Corp., Hudson, MA*, October 1996.

[45] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 architecture. In *Proc. of ICCD*, December 1998.

[46] J. Kin, M. Gupta, and W. H. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *Proc. MICRO-30*, December 1997.

[47] T. Knight. An architecture for mostly functional languages. In *Proc. ACM Conference on Lisp and Functional Programming*, August 1986.

[48] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. ISCA-8*, May 1981.

[49] S. M. Kurlander and C. N. Fischer. Minimum cost interprocedural register allocation. In *The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[50] D. Levitan, T. Thomas, and P. Tu. The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Processor. In *COMPCON'95*, March 1995.

[51] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Proc. ICS-11*, July 1997.

[52] M. H. Lipasti. *Value Locality and Speculative Execution*. Ph.D. thesis, Carnegie Mellon University, Pitsburgh, PA 15213, April 1997.

[53] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. on MICRO-29*, December 1996.

[54] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. ASPLOS-VII*, October 1996.

[55] W. L. Lynch, G. Lauterbach, and J. I. Chamdani. Low Load Latency through Sum-Addressed Memory (SAM). In *Proc. ISCA-25*, June 1998.

[56] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proc. ASPLOS V*, 1992.

[57] D. Matzke. Will Physical Scalability Sabotaze Performance Gains? In *IEEE Computer, 30(9)*, September 1997.

[58] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corp., WRL, June 1993.

[59] G. D. McNiven and E. S. Davidson. Analysis of Memory Referencing Behavior for Design of Local Memories. In *Proc. ISCA-15*, May 1988.

[60] S. Mehrotra and L. Harrison. Examination of a memory access clasification scheme for pointer-intensive and numeric programs. In *Proc. ICS-10*, September 1997.

[61] A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. ISCA-24*, June 1997.

[62] A. Moshovos and G.S. Sohi. Streamlining inter-operation communication via data dependence prediction. In *Proc. MICRO-30*, December 1997.

[63] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. A dynamic approach to improve the accuracy of data speculation. Technical Report 1316, Computer Sciences Dept., University of Wisconsin-Madison, March 1996.

[64] M. Moudgill and J. H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. In *IBM research report RC 20857 (91318)*, May 1997.

[65] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[66] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. ISCA-24*, June 1997.

[67] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. In *IEEE Computer, 31(5)*, May 1998.

[68] G. Reinman and B. Calder. Predictive Techniques for Aggresive Load Speculation. In *Proc. MICRO-31*, December 1998.

[69] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided load marking for memory renaming. Technical Report CS98-593, University of California, San Diego, July 1998.

[70] J. A Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proc. MICRO-30,* December 1997.

[71] A. Rogers and K. Li. Software support for speculative loads. In *Proc. ASPLOS-V,* October 1992.

[72] E. Rotenberg, Q. Jacobson, Y. Sazeides, and Jim Smith. Trace processors. In *Proc. on MICRO-30,* December 1997.

[73] A. Roth, A. Moshovos, and G. S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. ASPLOS-VIII*, October 1998.

[74] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proc. MICRO-30*, December 1997.

[75] Semiconductor Industry Association. *The National Roadmap for Seminconductors: Technology Needs*, 1997 edition. (Chapter on Overal Roadmap on Technology Characteristics).

[76] D. J. Shippy and T. W. Griffith. POWER2 fixed-point, data cache, and storage control units. In *IBM journal on research and development, 38(5),* October 1994. An on-line revised version can be found at: http://www.rs6000.ibm.com/resource/technology/fxu.html.

[77] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proc. ASPLOS-V,* October 1992.

[78] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. ISCA-17*, Seattle, WA, May 1990.

[79] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proc. ISCA-24*, June 1997.

[80] A. Sodani and G. S. Sohi. Understanding the Differences Between Value Prediction and Instruction Reuse. In *Proc. MICRO-31*, December 1998.

[81] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, March 1990.

[82] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. ISCA-22*, June 1995.

[83] G. S. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proc. ASPLOS-IV,* April 1991.

[84] S. Steely, D. Sager, and D. Fite. Memory reference tagging, US Patent 5,619,662, filed on Aug. 1994, April 1997.

[85] J. G. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. HPCA-4*, January 1998.

[86] The Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. *SPEC Newsletter, available on-line from http://www.specbench.org/osg/cpu95/news/cpu95descr.html,* September 1995.

[87] J.Y. Tsai and P.-C. Yew. The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. In *Proc. PACT'96,* October 1996.

[88] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proc. MICRO-28,* December 1995.

[89] G. S. Tyson and T. M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. MICRO-30*, December 1997.

[90] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proc. ISCA-24*, June 1997.

[91] A. H. Veen. Dataflow Machine Architectures. *ACM Computing Surveys, vol. 18,* December 1986.

[92] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, May 1998.

[93] D. W. Wall. Global register allocation at link-time. In *SIGPLAN'86 Symposium on Compiler Construction*, January 1986.

[94] L. Widigen, E. Sowadksy, and K. McGrath. Eliminating operand read latency. In *Computer Architecture News, 24(5)*, December 1996.

[95] K. M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proc. ISCA-24*, June 1997.

[96] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Processors. In *Proc. ISCA-23*, May 1996.

[97] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. PLDI*, June 1995.

[98] S. J. E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical report, WRL Research Report 93/5, Western Research Laboratory, 1993.

[99] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive training branch prediction. In *Proc. ISCA-19*, May 1992.