Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors

Andreas Moshovos
Electrical and Computer Engineering
Northwestern University
moshovos@ece.nwu.edu

Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
sohi@cs.wisc.edu

Abstract

We consider a variety of dynamic, hardware-based methods for exploiting load/store parallelism, including mechanisms that use memory dependence speculation. While previous work has also investigated such methods [19,4], this has been done primarily for split, distributed window processor models. We focus on centralized, continuous-window processor models (the common configuration today). We confirm that exploiting load/store parallelism can greatly improve performance. Moreover, we show that much of this performance potential can be captured if addresses of the memory locations accessed by both loads and stores can be used to schedule loads. However, using addresses to schedule load execution may not always be an option due to complexity, latency, and cost considerations. For this reason, we also consider configurations that use just memory dependence speculation to guide load execution. We consider a variety of methods and show that speculation/synchronization can be used to effectively exploit virtually all load/store parallelism. We demonstrate that this technique is competitive to or better than the one that uses addresses for scheduling loads. We conclude by discussing why our findings differ, in part, from those reported for split, distributed window processor models.

1 Introduction

Building memory systems capable of sustaining the demands of modern high-performance processors remains a continuous challenge. While ideally fast and large memory systems are either too expensive or impossible to build, several techniques have been developed to closely approximate them. One of the most successful methods today is to build a memory hierarchy. While quite successful, memory hierarchies provide only a partial solution to the memory problem. A complementary technique is to tolerate memory latency by exploiting parallelism in the load/store request stream. While memory hierarchies reduce the time it takes to respond to loads, the goal here is to send loads to memory earlier, as far in advance as possible of the instructions that need the data. As a result, memory latency is overlapped with other useful computation. Unfortunately, determining the earliest point when a load can safely access memory requires knowledge of its memory dependences.

Exploiting load/store parallelism can be done either statically or dynamically, using hardware- or software-based methods. In this paper, we study a variety of dynamic, hardware-based methods. In particular, we consider methods that are derived from combinations of the following parameters: (1) whether an address-based scheduler is used, and (2) whether memory dependence speculation [7,11,1,15,19,4,28,9,2,33] is used. In an *address-based scheduler*, load and store addresses are used to *determine* memory dependences for guiding load execution. In principle, this is similar to the register-based schedulers modern instruction-level-parallel capable processors use for extracting parallelism in the instruction stream (instead of using register dependences we use memory dependences). With *memory dependence speculation*, a load may execute even when some of its dependences are presently unknown. Memory dependence speculation can be useful, since, in contrast to register dependences, memory dependences cannot be determined as instructions are inserted into the instruction window (addresses are calculated at run-time and not necessarily in program order). Accordingly, waiting to determine all the dependences a load has may unnecessarily delay its execution. However, in using memory dependence speculation care must be taken to balance the benefits of correct speculation against the penalty paid on miss-speculations (a miss-speculation occurs when a true memory dependence is violated during execution). For this purpose, several memory dependence speculation methods have been developed. These range from naively speculating all loads immediately, to using memory dependence prediction to guess memory dependences and enforce synchronization among loads and stores [19,4,9]. We study all these alternatives (see Section 2.1 for descriptions).

Previous studies have also investigated similar methods for exploiting load/store parallelism [19,4]. However, these studies assumed distributed, split-window, dynamically scheduled superscalar processor models

(*split window*). In this work we study memory dependence speculation under a continuous, centralized window processor model (*continuous window)*. (In Section 2.2, we clarify the differences between these two models.) As we demonstrate, the choice of a split versus a continuous window greatly impacts the effectiveness of each method creating different tradeoffs and options. There are several reasons why a study assuming a continuous window is warranted: While future, high-performance processors may be forced to use split, distributed windows [7,27,21,13,30,31,24,6,29,8], virtually all modern processors use centralized, continuous windows. Moreover, traditionally, techniques developed initially for high-performance processors sooner or later find their way into other classes of processors (e.g., many of the techniques initially used for mainframes in the 60's and 70's found their way into micro-processors during the 80's and 90's). We are currently experiencing a proliferation of mobile and embedded processor applications. As technology progresses, such processors may use more aggressive techniques such as memory dependence speculation. Finally, distribution may come at the price of lower IPC (instructions per cycle) [21]. Depending on the specific application, it may be preferable to use a centralized window with a slower clock rate to achieve higher performance. Accordingly, it is important to determine what the tradeoffs are under continuous window processor model.

The rest of this paper is organized as follows: In Section 2, we provide the background information necessary for justifying our experimental results and choices. We discuss methods for exploiting load/store parallelism, including memory dependence speculation. In Section 2.1, we review a number of previously proposed memory dependence speculation policies, while in Section 2.2 we clarify the differences between split and continuous window processor models. In Section 3, we present our experimental analysis. In Sections 3.2 through 3.6, we focus on a centralized, continuous-window processor model. In Section 3.7, we demonstrate that in contrast to the continuous window processor, address-based scheduling coupled with naive memory dependence speculation is an insufficient solution for a split-window processor. We also explain why memory dependence speculation behaves differently under these two processor models. We summarize our findings in Section 4.

2 Methods for Exploiting Load/Store Parallelism

To tolerate slower memory devices we may exploit load/store parallelism executing loads in an order that is ultimately restricted only by true memory dependences[1]. One way of exploiting load/store parallelism is to *first* *determine* the true dependences of a load and *then* use this information to schedule its execution. Determining memory dependences requires inspection of the addresses loads and stores access. For this purpose we may use an address-based load/store scheduler. With this mechanism loads go through *two* scheduling phases. In the first, the base register dependence(s) is used to schedule the load's address calculation. In the second phase, the address is used to determine the load's memory dependences. This information is used to schedule the load's memory access. When an address-based scheduler is used, it makes sense to also have stores proceed into two phases. In the first, they calculate their addresses as soon as their base register becomes available. This address is inserted into the address-based scheduler for blocking the execution of dependent loads. The second step in store execution entails waiting for the actual data value. As soon as it becomes available, the address-based scheduler is notified releasing any dependent loads.

Even when an address-based scheduler is in place, waiting to determine the dependences of loads may offer suboptimal performance. With this method, a load that has calculated its address is forced to wait until it is established that no preceding, yet un-executed store writes to the same address (i.e., the load has no *unresolved* memory dependences). Accordingly, it is possible to unnecessarily delay a load from accessing memory. This is the case if a load that has no true dependences is forced to wait until all preceding stores have calculated their addresses.

To expose the parallelism that is hindered by the use of run-time calculated addresses we may use memory dependence speculation. Under *memory dependence speculation*, we do not delay executing a load until we determine that it has no true dependences. Instead, we guess whether a load has any. As a result, a load may access memory before a preceding store with which a dependence *may* exist. Eventually, when preceding store addresses are calculated, we can determine whether the resulting execution order was valid. If no true dependence was violated then speculation was successful and no further action is necessary. In this case, performance may have improved as the load executed earlier making memory *appear* faster. However, if a true dependence was violated then speculation was erroneous (i.e., a miss-speculation). In this case corrective action is necessary. *Squash invalidation,* the hardware-based miss-

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 for clarity.

speculation recovery method used today, works by invalidating and re-executing all instructions following the miss-speculated load.

Memory dependence speculation can be used with and without an address-based scheduler. Using an address-based scheduler is desirable as it allows us to use as much dependence information as possible in deciding when to execute a load. However, building an address-based scheduler for wide-issue and long window processors may be a daunting proposition. Considerations include cost, complexity, and impact on load latency. Consequently, it is desirable to devise mechanisms that offer similar benefits without the need for such a scheduler. For this reason, we also consider configurations that do not use address-based scheduling. In this case, a mechanism is required to detect memory dependence violations. This can be done by recording all speculative loads and by having stores check for violations when they write to memory. The latency through this mechanism can only impact miss-speculation penalty and not load or store latency. This is because this process does not produce any data for further computation.

When using memory dependence speculation, care must be taken to balance the performance benefits gained on correct speculation against the penalty paid on miss-speculations [19]. The miss-speculation penalty includes the following three components: (1) The work thrown away to recover from the miss-speculation (which in the case of squash invalidation, i.e., invalidating all instructions after the miss-speculated load, may include unrelated computations). (2) The time, if any, required to perform the invalidation. Finally, (3) the opportunity cost associated with not executing some other instructions instead of the mispeculated load and the instructions that used erroneous data [19]. Ideally, loads would execute as early as possible while miss-speculations would be completely avoided.

In the next section, we review previously proposed memory dependence speculation policies. However, before we do so, we note that two other possibilities exist for reducing the penalty of miss-speculation [19]: (1) minimizing the amount of work lost on miss-speculation or (2) reducing the time required to redo this lost work [19]. Instruction reuse falls into the same category [25]. A technique to reduce the amount of work lost on miss-speculation is *selective invalidation*. With this technique, only the instructions that used erroneous data are invalidated and re-executed on misspeculation [16]. As we demonstrate in Section 3.4, under these assumptions memory dependence miss-speculations are virtually non-existent, hence there is no problem with miss-speculations and no need for a selective invalidation mechanism.

2.1 Memory Dependence Speculation Policies

We consider the following five memory dependence speculation policies: (1) naive memory dependence speculation, (2) speculation/synchronization, (3) selective speculation, (4) store barrier, and (5) no speculation. To better understand the potential pros and cons of each policy it is best if we first consider the ideal memory dependence speculation policy [19]. In this policy loads are delayed only as long as it is necessary. This requires perfect, *a priori* knowledge of all relevant memory dependence. 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.

Naive Memory Dependence Speculation: This an overly optimistic form of memory dependence speculation, yet it is very simple. In this scheme, loads access memory immediately after address calculation even if unresolved dependences exist. While this scheme schedules loads aggressively, it may suffer from a high number of miss-speculations. As we demonstrate in Section 3.3, this is so when not using an address-based scheduler.

Memory Dependence Speculation/Synchronization: This scheme mimics ideal memory dependence speculation [19,4]. This is done by: (1) predicting whether the immediate execution of a load is likely to violate a true data dependence, and if so, (2) predicting the store (or stores) the load depends upon, and, (3) enforcing synchronization between the dependent instructions. Initially, in the absence of any dependence information, naive memory dependence speculation is used. When a miss-speculation occurs, information about the corresponding store and load is recorded. The next time any of the two executes the recorded information is used to predict the dependence and to enforce synchronization.

Selective Memory Dependence Speculation: This is a simplification of memory dependence speculation. Here we carry out only the first part of the ideal 3-part operation described previously. In this scheme, the loads that are likely to cause miss-speculation are not speculated. Instead, they wait until all their ambiguous dependences are resolved [19,4,14,33]. Due to the lack of explicit synchronization, this prediction policy may unnecessarily delay loads and, for this reason, negatively impact performance. In practice, and as we demonstrate in Section 3.5, selective data dependence speculation does not perform close to ideal memory dependence speculation.

Store Barrier: This scheme predicts whether a store has true dependences that would normally get mispeculated [9,2]. If it does, *all* loads following the store in

question are made to wait for the store's address. This policy can be successful in both eliminating miss-speculations and in delaying loads with dependences only as long as it is necessary. However, it may unnecessarily delay other unrelated loads. Compared to the two previous methods, the store barrier policy may require smaller predictors (only entries for stores are required).

No Speculation: Finally, we can choose to avoid speculating memory dependences altogether. This is the simplest and most pessimistic scheme, as loads wait until all their memory dependences are resolved.

2.2 Split vs. Continuous Window Processors

In this section, we clarify the differences between split-window and continuous-window processor models.

Centralized, continuous-window: All instructions are inserted in the window in program order. Moreover, in deciding which instructions to execute, program order priority is used. That is, older instructions are given preference. Finally, the instructions currently in the window form a continuous part of the dynamic execution trace.

Distributed, split-window: In this configuration the window is split into a number of smaller sub-windows. Instructions are not necessarily inserted in the window in program order. Moreover, the instructions currently in the window do not necessarily form a continuous part of the execution trace.

The previous studies on memory dependence speculation have primarily assumed split window models. In [19], a model of the Multiscalar architecture [7,27] was used, while, in [4], a model of the Alpha 21264 processor [13,14] was used. While quite dissimilar, both models use distributed instruction schedulers that may not use program order priority in scheduling store and load address calculations. Moreover, in the Alpha processor model, stores wait for both data and the base register before issuing. As a result, a load will not be prevented from accessing memory prematurely even if the store could calculate its address early enough. Under the Multiscalar execution model, the window is split over several units. Each sub-window is assigned a continuous portion of the dynamic execution trace; however, fetching proceeds independently across units. As a result, not only is program order priority not used in scheduling decisions (across units), but also a load may be fetched before a preceding in program order store. This work differs in that: (1) we consider a continuous instruction window, and (2) we also study address-based load/store scheduling — assuming that stores can post their addresses as soon as possible.

3 Experimental Analysis

In this section, we study various methods of extracting load/store parallelism. For clarity we use an **A/B** naming scheme for the various configurations, where **A** denotes whether an address-based scheduler is used and **B** denotes what memory dependence speculation policy is used. We use **AS** for configurations that use an address based scheduler and **NAS** for those that don't. We use **NO**, **NAV**, **SEL**, **STORE**, **SYNC** for no speculation, naive, selective, store barrier and speculation/synchronization respectively. Finally, we use **ORACLE** for a speculation method that has perfect, in advance knowledge of all memory dependences (see Section 3.2 for a detailed description). For example, we will use **NAS/SEL** for configurations that do not use address-based scheduling but utilize selective memory dependence speculation.

The rest of this section is organized as follows: We start in Section 3.2, by determining how much there is to be gained from exploiting load/store parallelism. In Section 3.3, we demonstrate that naive memory dependence speculation (NAS/NAV) can be used to attain some of these performance benefits without using an address-based scheduler. In Section 3.4, we consider using an address-based scheduler and its interaction with memory dependence speculation (AS/--- configurations). In Section 3.5, we consider using selective (NAS/SEL) and store barrier (NAS/STORE) speculation to improve performance when no address-scheduler is present. Finally, in Section 3.6, we do the same for speculation/synchronization (NAS/SYNC). We conclude by explaining why our findings are different from those reported for split-window environments.

3.1 Methodology

In our experiments we used the SPEC'95 programs which we compiled for the MIPS-I architecture [12], using the 2.7.2 version of the GNU gcc compiler (flags: -O2 -funroll-loops -finline-functions). We translated FORTRAN codes to C using AT&T's f2c compiler. To attain reasonable simulation times we: (1) modified the standard *train* or *test* inputs, and (2) used *sampling* [32,23,3]. Table 1 reports the dynamic instruction count, the fraction of loads and stores and the sampling ratios per program. A description of the modified inputs can be found in [18]. The observation size used is 50,000 instructions. The sampling ratios are reported under the "SR" columns as "timing:functional" ratios. These ratios resulted in roughly 100M instructions being simulated in timing mode. During the functional portion of the simulation, the following structures were simulated: I-cache, D-cache, and branch prediction. In early experiments we found that sampling affects absolute performance only slightly (less than 1.5% for all programs except 102.swim where the change was about 3%). In the rest of the evaluation, we will refer to the benchmarks by using

the first numbers of their name shown in Table 1.

Program	*IC*	*Loads*	*Stores*	*SR*
SPECint'95				
099.go	133.8	20.9%	7.3%	N/A
124.m88ksim	196.3	18.8%	9.6%	1:1
126.gcc	316.9	24.3%	17.5%	1:2
129.compress	153.8	21.7%	13.5%	1:2
130.li	206.5	29.6%	17.6%	1:1
132.ijpeg	129.6	17.7%	8.7%	N/A
134.perl	176.8	25.6%	16.6%	1:1
147.vortex	376.9	26.3%	27.3%	1:2
SPECfp'95				
101.tomcatv	329.1	31.9%	8.8%	1:2
102.swim	188.8	27.0%	6.6%	1:2
103.su2cor	279.9	33.8%	10.1%	1:3
104.hydro2d	1,128.9	29.7%	8.2%	1:10
107.mgrid	95.0	46.6%	3.0%	N/A
110.applu	168.9	31.4%	7.9%	1:1
125.turb3d	1,666.6	21.3%	14.6%	1:10
141.apsi	125.9	31.4%	13.4%	N/A
145.fpppp	214.2	48.8%	17.5%	1:2
146.wave5	290.8	30.2%	13.0%	1:2

Table 1. *Benchmark Execution Characteristics. Instruction counts ("IC" columns) are in millions.*

We employ execution-driven timing simulation using a modified version of the Multiscalar simulator [3]. All but system code references are included. System calls are handled by trapping to the OS of the simulation host. Moreover, event-driven simulation is used for both the out-of-order (OOO) core and the memory system. The default configuration used in our continuous window processor experiments is detailed in Table 2.

3.2 Performance Potential of Load/Store Parallelism

An initial consideration is whether exploiting load/store parallelism can yield significant enough performance improvements to justify the additional costs. These costs include mechanisms to detect or predict memory dependences, mechanisms that use dependence information to schedule load execution, and mechanisms to detect memory dependence violations if memory dependence speculation is used.

In this section, we show the importance of exploiting load-store parallelism by comparing two processor models: **NAS/NO** and **NAS/ORACLE**. The first does not exploit load/store parallelism. The second implements an approximation of ideal memory dependence speculation. It has an oracle disambiguation mechanism and executes loads as soon as possible without ever violating a true memory dependence. (We will explain in more detail why this is an approximation of ideal memory dependence speculation in Section 3.4.1.) Execution under the first model proceeds as follows: After an instruction is fetched, it is decoded and placed into the instruction window where its register dependences and register data 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. Consequently, loads may execute out-of-order only with respect to other loads and non-store instruc-

Fetch Unit	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 [17]. 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. It takes a combined 4 cycles for an instruction to be fetched and placed into the reorder buffer. 32 integer, 32 floating point, HI, LO and FSR. 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 L1 data cache. Combines store requests for load forwarding.
Date Caches	32K, 2-way set associative, 4 banks, 256 sets per bank, 32 bytes per block, 2 cycle hit, 10 cycle miss to unified L2, 50 cycle miss to main memory. Lockup-free, 8 primary miss per bank, 8 secondary misses per primary. LRU replacement.
Unified L2 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 Mem	Infinite, 34 cycle + #4 word transfer * 2 cycles access.

Table 2. *Default configuration for continuous window experiments.*

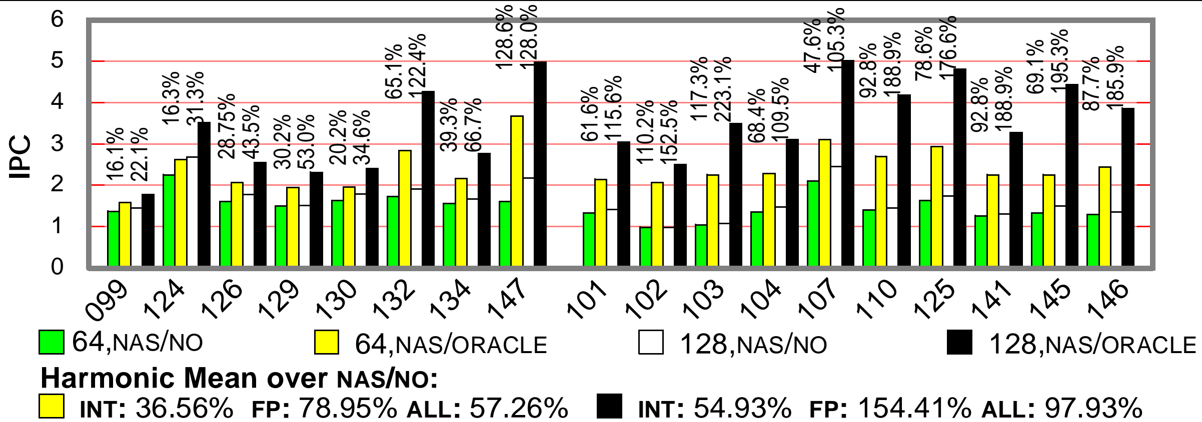

Figure 1: *Performance (as IPC) with and without exploiting load/store parallelism. Notation used is "instruction window size","load/store execution model". Speedups of NAS/ORACLE speculation over NAS/NO speculation are given on top of each bar.*

tions. 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 to write their data.

Figure 1 reports IPCs for the two aforementioned configurations. We consider configurations with 64-entry and 128-entry instruction windows. (The 64-window configuration is derived from Table 2, by reducing issue width to 4, load/store ports to 2, and all functional units to 2.) 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 ("NAS/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 relative to performance as the instruction window increases.

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. We report false dependences as a fraction over all committed loads. We account for false dependences once per executed load and at 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.

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

Table 3. *Fraction of loads with false dependences (FD columns) and average false dependence resolution latency in cycles (RL columns) for the 128-entry instruction window processor.*

3.3 Performance with Naive Memory Dependence Speculation

As we have seen, extracting load/store parallelism can result in significant performance improvements. In this section, we consider using naive memory dependence speculation for extracting these performance improvements. For this purpose, we assume the same processor

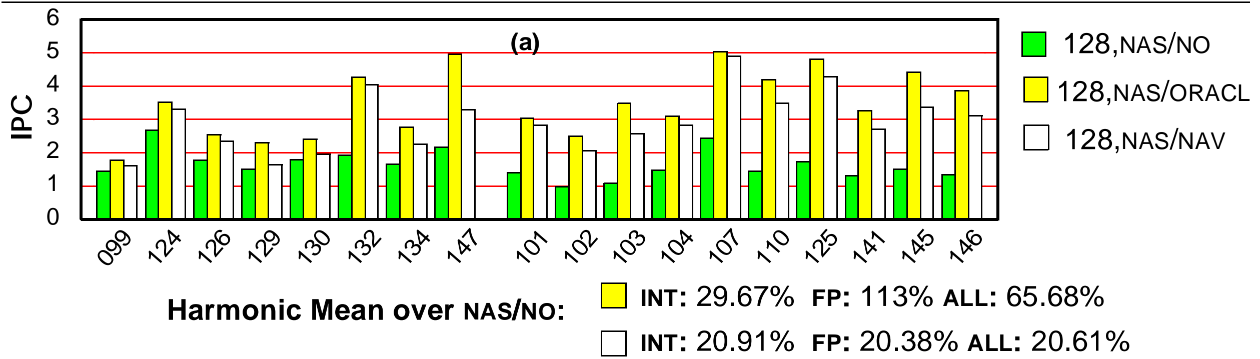

Figure 2: *Performance with naive memory dependence speculation and no address-scheduler.*

model assumed in the previous section, but we allow loads to speculatively access memory as soon as their address operands become available (NAS/NAV). All speculative load accesses are recorded in a separate structure, so that preceding stores can detect whether a true memory dependence was violated by a speculatively issued load. Note that the latency through this detection mechanism impacts only how quickly miss-speculations are detected. For this reason, slower mechanisms may be tolerable. Also, note that a similar load issue model was used by Chrysos and Emer in their study of memory dependence speculation [4].

Figure 2, part (a) reports performance (as IPC) for the 128-entry processor model when, from left to right, no speculation is used (NAS/NO), when oracle dependence information is available (NAS/ORACLE), and when naive speculation is used (NAS/NAV). We can observe that for all programs, NAS/NAV results in higher performance compared to NAS/NO. However, the performance difference between NAS/NAV and NAS/ORACLE is significant, supporting our claim that the net penalty of miss-speculation can become significant. As supported also by the measurements of column NAV of Table 4, memory dependence miss-speculations are at fault. There we report memory dependence miss-speculation frequency. We measure miss-speculation frequency as a percentage over all committed loads.

In this context, the various memory dependence speculation methods we reviewed in Section 2.1 could be used to reduce the net performance penalty of miss-speculation. However, before we consider this possibility (which we do in Sections 3.5 and 3.6), we first investigate using an address-based scheduler to extract load/store parallelism and its interaction with memory dependence speculation.

3.4 Using Address-Based Scheduling

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. In their pioneering work, Patt, Melvin, Hwu and Shebanow have also considered some of the tradeoffs involved in using address-based scheduling in dynamically-scheduled high-performance micro-architectures [22].

We start by considering two models: AS/NO and AS/NAV. In both models, 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 (they only wait for base register data). Furthermore, loads are allowed to inspect preceding store addresses before accessing memory. If a true dependence is found, a load always waits. When no speculation is used (**AS/NO**), loads wait until all their ambiguous dependences are resolved. When naive memory dependence speculation is used (**AS/NAV**), a miss-speculation 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 miss-speculation[2]. As we noted earlier, under AS/NAV miss-speculations are virtually non-existent. There are three reasons why this is so. (1) Loads that would otherwise access memory get delayed because they can detect a true dependence. (2) Loads with unresolved, yet true dependences are still allowed to access memory. However, before they have a chance of propagating the memory supplied value the correct value is provided by the corresponding store. (3) Loads are delayed because preceding stores consume resources to have their addresses calculated and posted for disambiguation purposes.

Figure 3 compares the performance of AS/NAV and AS/NO. Part (a) reports the relative performance of AS/NAV over AS/NO, while part (b) reports absolute performance

2. These conditions do not violate the semantics of sequential programs. However, they may violate the semantics of explicitly parallel programs. A discussion of the latter issue is beyond the scope of this paper.

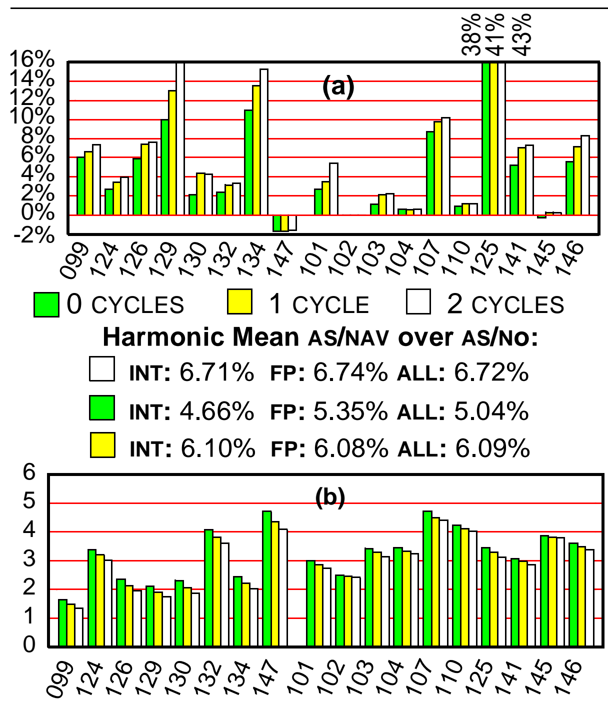

Figure 3: *(a) Relative performance of naive memory dependence speculation (AS/NAV) as a function of address-based scheduler latency (0, 1, and 2 cycles). Performance variation is reported with respect to the same processor model that does not use memory dependence speculation (AS/NO). Base performance (IPC for AS/NO) is shown in part (b).*

(IPC) for the base configuration (AS/NO). We 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 in part (a), we should note that the base configuration is different for each bar.

For most programs, naive memory dependence speculation is still a win. Performance differences are not as drastic as they were when the address scheduler was unavailable, yet they are still significant. More importantly, the performance difference between AS/NO and AS/NAV increases as the latency through the load/store scheduler also increases. For 147.vortex and 145.fpppp, AS/NAV performs worse than AS/NO. Miss-speculations are not the cause for this degradation. Rather, it is increased resource contention caused by loads with ambiguous dependences that get to access memory speculatively only to receive a new value from a preceding store. 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 (Section 2).

The results of this section are that using address-based scheduling exposes some of the load/store parallelism. Moreover, using naive memory dependence speculation can further improve performance. Interestingly, in this case, memory dependence miss-speculations are virtually non-existent. For this reason, we do not consider any of the other memory dependence speculation schemes. Diep, Nelson and Shen have also shown that an AS/NAV policy can improve performance over no speculation [5]. This was done for a model of the PowerPC 620 architecture. Here we take into account the impact of address-based scheduling on load latency and assume larger instruction windows. Moreover, we compare it with the oracle memory dependence speculation method (next Section).

3.4.1 Comparing with Oracle Memory Dependence Speculation

Having shown that address-based scheduling can expose some of the load/store parallelism, we now compare the performance so obtained to that of the oracle scheme of Section 3.2 (NAS/ORACLE). 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 (AS/NAV). The reason is that preceding stores consume resources to calculate and post their addresses for scheduling purposes. Such resources are 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. Moreover, the complexity of an address-based scheduler may be daunting for wider issue machines. For these reasons, 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.

Figure 4 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, part (b), AS/NO 0-CYCLE configuration). From left to right, the four bars report performance with: (1) oracle disambiguation and no address-based scheduler (NAS/ORACLE, Section 3.2), (2) through (4) naive memory dependence speculation and address-based scheduler (AS/NAV) with a latency of 0, 1 and 2 cycles respectively (which we evaluated earlier in this section). With few exceptions, the 0-cycle AS/NAV and the NAS/ORACLE perform equally well. Once address-based scheduling increases load latency by

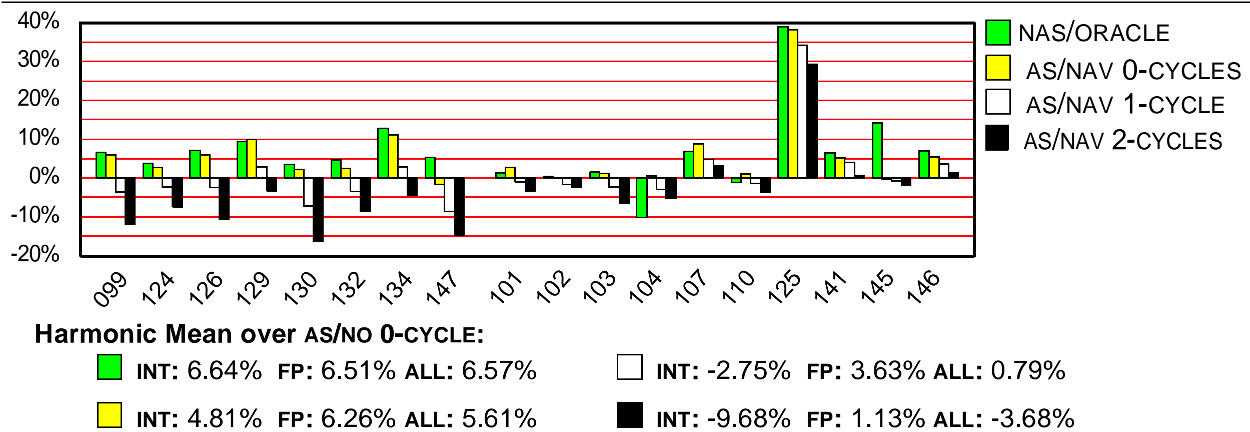

Figure 4:*Comparing oracle disambiguation and address-based scheduling plus naive memory dependence speculation. Base configuration uses a 0-cycle address-based scheduler and no speculation (AS/NO).*

1 or more cycles, performance degrades making this an under-performing solution.

In some cases, the 0-cycle AS/NAV configuration performs slightly better than NAS/ORACLE. This result is an artifact of our euphemistic use of the term "oracle". In our oracle model, a store is allowed to issue only after both its data and address operands become available. Consequently, 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 get their value at least 3 cycles after the moment the store was issued. In contrast, when the address-based scheduler is in place, a store may calculate its address long before its data is available. As a result, dependent loads can access the store's value as soon as it becomes available. In an actual implementation, it may be possible to overlap store address calculation and store data reception without using an address-based scheduler by allowing loads to issue when the store reaches the memory stage (e.g., [10, 28]).

In this section, we demonstrated that naive speculation coupled with address-based scheduling (AS/NAV) offers performance similar to that possible with the oracle method (NAS/ORACLE) that utilizes perfect, *a priori* knowledge of all memory dependences. However, we have also demonstrated that if address-based scheduling increases load latency even by as little as 1 cycle, performance gradually degrades. For these reasons, we next consider configurations that do not use address-based scheduling, but utilize more accurate memory dependence speculation techniques to approximate ideal memory dependence speculation.

3.5 Selective Speculation and Store Barrier Speculation

In this section, we consider using selective (NAS/SEL) or store barrier (NAS/STORE) memory dependence speculation as alternatives to address-based scheduling. We start from a configuration that has no address-based scheduler and introduce predictors that provide hints for load scheduling to the existing register-based scheduler. In both memory dependence speculation schemes we used a 4K, 2-way set associative memory dependence predictor. For selective speculation the predictor guesses whether a load has a dependence. If so, this load is not speculated. That is we wait until all preceding stores have calculated their address and received their data. For store barrier, the predictor guesses whether a store has a dependence. If so, all subsequent loads are delayed until this store calculates its address and receives its data. Both predictors use 2-bit saturating counter-based confidence automatons. It takes 3 miss-speculations on a specific load or store before the existence of a dependence is predicted. All counters are reset every 1 million cycles to allow adapting back [4].

Figure 5 shows how performance varied over NAS/NAV (Figure 2). We use NAS/NAV as our base configuration since both NAS/SEL and NAS/STORE attempt to improve accuracy over it. While successful in some cases, both techniques fall short of providing performance close to oracle (NAS/ORACLE). Even worse, in some cases performance drops below that possible with naive memory dependence speculation. As we explained in Section 2.1, either technique may both improve or hurt performance.

3.6 Speculation/Synchronization

In this section, we consider using speculation/synchronization (NAS/SYNC) for approximating NAS/ORACLE.

Speculation/synchronization works by initially using naive memory dependence speculation for all loads. Once a miss-speculation is detected, information about the dependent loads and stores is stored in a memory dependence prediction table (MDPT) [19]. The next time

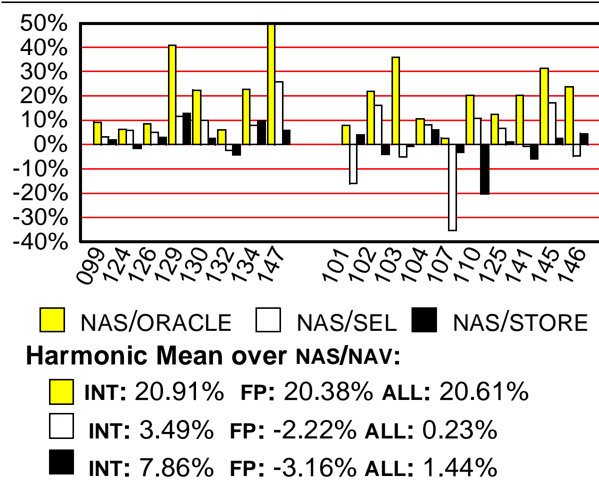

Figure 5: *Using Selective Memory Dependence Speculation (NAS/SEL) or the Store Barrier method (NAS/STORE) to approximate ideal speculation. Base configuration uses naive memory dependence speculation and has no address-based scheduler (NAS/NAV, Section 3.3).*

the same instructions are encountered the MDPT is used to predict the dependence and to enforce synchronization. The speculation/synchronization mechanism we used 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 [20,4]. No confidence mechanism is associated with each MDPT entry; once an entry is allocated, synchronization is always enforced. However, we flush the MDPT every one million cycles to reduce the frequency of false dependences [4]. Synchronization is incorporated into the register-scheduler, which we assume to follow the RUU model [26]. This is done as follows: an additional[3] register identifier is introduced per RUU entry. This identifier is used to introduce 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, use that register identifier to mark themselves as consumers of the MDPT supplied synonym. Synchronization is achieved by: (1) making loads wait for the closest 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.

3. It's not really necessary to use an additional register specifier. Stores may use their unused destination register field, while loads may use their unused second source register field.

Figure 6, part (a) reports performance results relative to naive memory dependence speculation (NAS/NAV, Section 3.3). As it can be seen NAS/SYNC offers most of the performance improvements that are possible with NAS/ORACLE. We also provide the miss-speculation rates with NAS/SYNC in Table 4 (reported is the number of miss-speculations over all committed loads). Miss-speculations are virtually non-existent. This observation suggests that for the most part the performance differences compared to NAS/ORACLE are the result of either (1) false dependences, or (2) of failing to identify the appropriate store instance with which a load had to synchronize with.

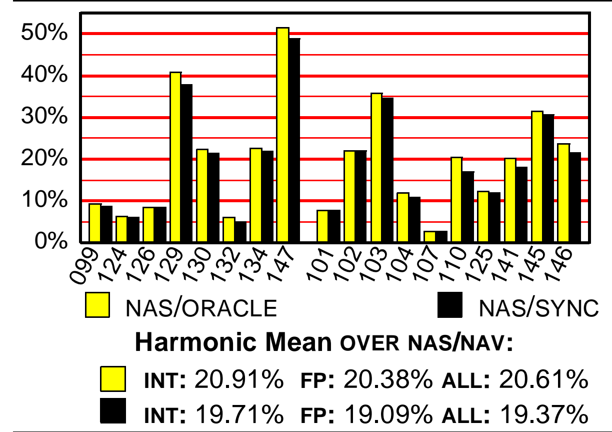

Figure 6: *Performance of an implementation of speculation/synchronization. Base is NAS/NAV.*

	NAV	SYNC		NAV	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 4. *Memory dependence miss-speculation rate with our speculation/synchronization mechanism ("SYNC" columns) and with naive speculation ("NAV" columns).*

The results of this section suggest that the speculation/synchronization method can offer performance that is very close to that possible with a method that utilizes ideal dependence information. Moreover, this configuration is an attractive alternative to address-based scheduling as it completely alleviates the need for an address-based scheduler.

3.7 Interaction of Window Type and Memory Dependence Speculation - Discussion

We have shown that memory dependence miss-speculations can be virtually eliminated in a continuous window with an address-based scheduler. Moreover, we have seen that performance with a 0-cycle latency address-based scheduler coupled with naive memory dependence speculation (AS/NAV) closely approximates the oracle configuration (NAS/ORACLE). This result suggests that there is no need for more advanced memory dependence speculation techniques if building a 0-cycle address-based scheduler is possible. However, previous work has shown that this is not true for a split window processor model [18]. In particular, it has been shown that even if a 0-cycle address-based scheduler can be built, it cannot eliminate a large number of miss-speculations when naive memory dependence speculation is used. In this section, we explain why an address-based scheduler avoids virtually all miss-speculations for the continuous window processor, but fails to do so for the split window processor.

For our purposes, we will use the example of Figure 7. Part (a) 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 memory dependence speculation.) Part (b) shows how two iterations of this loop may be executed under the continuous window execution model. Under this model, instructions are fetched in order and the window is filled up gradually. Consequently, by the time the dependent load (load a[i]) has a chance to calculate its address, all preceding relevant addresses, including that of store a[i], have also been calculated. 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 demonstrated in Section 3.4, memory dependence miss-speculations are virtually non-existent in this environment.

Let us now consider a split window model. Under this different set of assumptions, instructions are not necessarily fetched in program order. Moreover, enforcing program order priority in the scheduler may not be possible. Under this model, the two iterations of the loop may be assigned to different units (sub-windows), as shown in part (b) of Figure 7. Accordingly, 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 instantaneously, it would not be possible to avoid the miss-speculation.

4 Summary

We have studied various methods for exploiting load/store parallelism under a continuous window processor model. Our findings were:

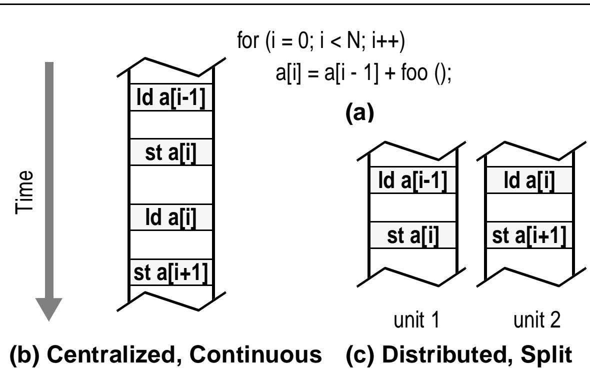

Figure 7: *Executing a loop (a) under: (b) a continuous-window execution model, and (c) a split-window execution model.*

1. Exploiting load/store parallelism can greatly improve performance over not doing so. For an 128-entry window processor performance improved by about 55% (integer) and 154% (floating-point) on the average.
2. Using address-based scheduling captures most of this performance potential even when no memory dependence speculation is used. Naive memory dependence speculation (AS/NAV) can offer speedups of 4.6% (integer) and 5.3% (floating point) over no speculation (AS/NO) for a 0-cycle address-based scheduler configuration. However, as the latency through the address-based scheduler increases, performance degrades making this an under performing option. In this case, we may consider using memory dependence speculation without an address-based scheduler.
3. When no address-based scheduler is used, naive memory dependence speculation (NAS/NAV) can offer some of potential performance of exploiting load/store parallelism. In particular, compared to no speculation (NAS/NO) speedups of 29% (integer) and 113% (floating-point) were possible. However, net miss-speculation penalty is high, justifying using more advanced memory dependence speculation policies.
4. Selective (NAS/SEL) and store barrier (NAS/STORE) speculation are not robust techniques. While they can at times improve performance over naive memory dependence speculation (NAS/NAV), they often hurt performance. Overall, no significant performance improvements were observed over naive memory dependence speculation.
5. Memory dependence speculation/synchronization (NAS/SYNC) can significantly improve performance over naive memory dependence speculation (NAS/

NAV). Moreover, it offers performance very close to that obtained when perfect, in-advance knowledge of all memory dependences was available. In particular, with speculation/synchronization performance improved by 19.7% (integer) and 19.1% (floating-point) on average. With a method that utilized perfect memory dependence information the corresponding speedups would have been 20.9% (integer) and 20.4% (floating-point) over naive speculation. The potential advantage of this design is that it leverages the existing register dependence scheduler to also implement load/store scheduling.

This study deepens our understanding of memory dependence speculation and of a variety of methods for exploiting load/store parallelism. We have identified cases when it is advantageous to use memory dependence speculation under a continuous window processor environment. Moreover, we have identified conditions under which advanced memory dependence speculation policies may be utilized. We have shown that speculation/synchronization may be used not only as a performance enhancing technique but also to simplify the design of aggressive, dynamically scheduled processors with continuous windows.

References

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

[2] D. Adams, A. Allen, R. F. 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*, Feb. 1997.

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

[4] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. International Symposium on Computer Architecture-25*, June 1998.

[5] T. A. Diep, C. Nelson, and J. P. Shen. Performance evaluation of the powerpc 620 microarchitecture. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 163–175, June 1995.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.

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

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

[9] 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, Mar. 1997.

[10] 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, Sept. 1996.

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

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

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

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

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

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

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

[18] A. Moshovos. *Memory Dependence Prediction*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Dec. 1998.

[19] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

[20] A. Moshovos and G. Sohi. Streamlining inter-operation communication via data dependence prediction. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[21] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

[22] Y. N. Patt, S. W. Melvin, W.-M. Hwu, and M. C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. *In Proc. 18th International Symposium on Microarchitecture*, 1985.

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

[24] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proc. on Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[25] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

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

[27] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. International Symposium on Computer Architecture-22*, June 1995.

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

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

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

[31] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proc. International Symposium on Computer Architecture-24*, June 1997.

[32] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Processors. In *Proc. International Symposium on Computer Architecture-23*, May 1996.

[33] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculative techniques for improving load related scheduling. In *International Symposium on Computer Architecture-26*, May 1999.