Read-After-Read Memory Dependence Prediction

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

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

Abstract: *We identify that typical programs exhibit highly regular read-after-read (RAR) memory dependence streams. We exploit this regularity by introducing read-after-read (RAR) memory dependence prediction. We also present two RAR memory dependence prediction-based memory latency reduction techniques. In the first technique, a load can obtain a value by simply naming a preceding load with which a RAR dependence is predicted. The second technique speculatively converts a series of* $LOAD_1$-USE_1,...,$LOAD_N$-USE_N *chains into a single* $LOAD_1$-USE_1...USE_N *producer/consumer graph. Our techniques can be implemented as surgical extensions to the recently proposed read-after-write (RAW) dependence prediction based speculative memory cloaking and speculative memory bypassing. On average, our techniques provide correct values for an additional 20% (integer codes) and 30% (floating-point codes) of all loads. Moreover, a combined RAW- and RAR-based cloaking/bypassing mechanism improves performance by 6.44% (integer) and 4.66% (floating-point) even when naive memory dependence speculation is used. The original RAW-based cloaking/bypassing mechanism yields improvements of 4.28% (integer) and 3.20% (floating-point).*

1 Introduction

Modern high-performance processors exploit regularities in "typical" program behavior. Extensively studied examples include caching, branch prediction and value prediction. This experience points to a possible direction for further performance improvements: identifying currently unknown regularities in program behavior and exploiting these regularities to our advantage. Following this rationale, we identify that typical programs exhibit highly regular "read-after-read" (RAR) memory dependence streams. A RAR dependence exists between two loads if both access the same address and no intervening store writes to the same address. We have found that if at some point two loads are RAR dependent, then with high probability these loads will be RAR dependent again soon even though they may be accessing a *different* address.

To exploit this regularity we present: (1) history-based RAR memory dependence prediction, and (2) two techniques that use this prediction to reduce memory latency. In RAR memory dependence prediction an earlier detection of a RAR dependence is used to predict the dependence the next time the same loads are encountered. We use this prediction to create a new name space through which loads can get speculative values. In our technique, a load can get a value by identifying a preceding load that also reads it (i.e., a RAR dependence exists with that load). Using PC-based prediction this identification takes place early in the pipeline without actual knowledge of memory addresses. We further reduce load latency by transforming a number of $LOAD_1$-USE_1,...,$LOAD_N$-USE_N chains into a single, yet speculative $LOAD_1$-USE_1...USE_N producer/consumer graph. Consequently, the first load can propagate its value to the consumers of all its RAR dependent loads.

An advantage of our techniques is that they can be implemented as surgical extensions to the recently proposed *speculative memory cloaking (cloaking)* and *speculative memory bypassing (bypassing)* respectively [15]. Figure 1 provides an overview of these techniques. Part (a) shows the original RAW-prediction-based cloaking and bypassing, while part (b) shows our RAR-prediction-based techniques. Our RAR-prediction-based cloaking and bypassing complement their RAW counterparts by predicting loads that the original cloaking and bypassing cannot. These are loads that do not experience RAW dependences. However, the utility of our techniques extends also to loads that have RAW dependences with *distant* stores. While RAW-based cloaking and bypassing is theoretically possible for such loads, practical considerations may prevent us from detecting the corresponding RAW dependences. As we explain in detail at the end of Section 3.1, this is the result of the limited scope of the underlying dependence detection mechanisms.

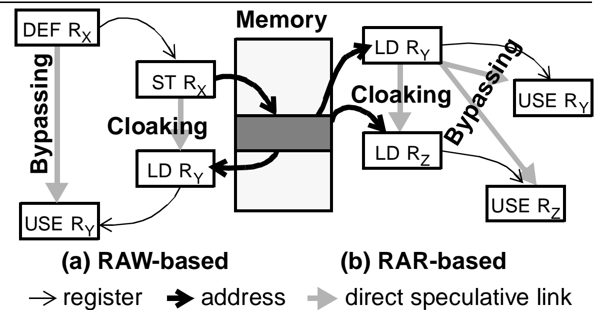

Figure 1: *Speculative Memory Cloaking and Bypassing. (a) Original proposal: Exploiting RAW dependences. (b) Our techniques: Exploiting RAR dependences.*

Our contributions are: (1) we demonstrate that regularity exists in the RAR memory dependence stream of typical programs, (2) we introduce history-based RAR memory dependence prediction, (3) we propose applications of this prediction, and (4) we compare the accuracy of our techniques and of load value prediction [12] and show that the two approaches are complementary.

The rest of this paper is organized as follows: in Section 2 we demonstrate that programs exhibit regular RAR memory dependence streams. In Section 3 we discuss the rationale for our RAR memory dependence prediction based methods and how they can be implemented as extensions to RAW-based cloaking (Section 3.1) and RAW-based bypassing (Section 3.2). In Section 4 we review related work. In Section 5 we

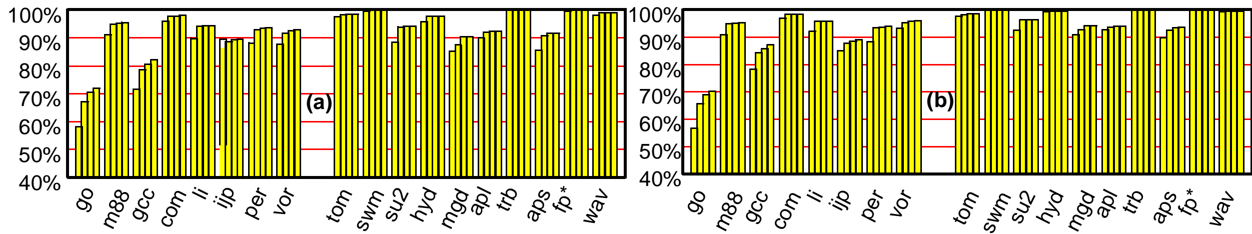

Figure 2: *Memory Dependence Locality of RAR dependences (range: 1 to 4). Address window size: (a) Infinite, (b) 4K entry.*

evaluate the accuracy and performance of our techniques. Finally, in Section 6 we summarize our findings. For clarity we use the terms *dependence* and *memory dependence* interchangeably.

2 Quantifying RAR Memory Dependence Stream Regularity

In this section, we demonstrate that the RAR dependence stream of the SPEC95 programs is regular (our methodology and benchmarks are described in Section 5.1). We show that most loads exhibit temporal locality in their RAR-dependence stream. That is, once a load experiences a RAR dependence, chances are that it will experience the same RAR dependence again soon. Moreover, we demonstrate that the working set of RAR-dependences per load is relatively small. These properties enable history-based prediction of RAR dependences.

We represent RAR dependences as (PC_1, PC_2) pairs where PC_1 and PC_2 are instruction addresses of RAR-dependent loads. Generally, given a set of loads that access the same memory address, RAR dependences exist between *any pair* of loads in program order (provided of course that no intervening store writes to the same address). We restrict our attention to RAR dependences between the *earliest* in program order load (source) and any of the subsequent loads (sinks). For example, given the sequence LD_1 A, LD_2 A, LD_3 A, we will account for the (LD_1 A, LD_2 A) and (LD_1 A, LD_3 A) dependences only and not for the (LD_2 A, LD_3 A) dependence. This definition is convenient for RAR dependence prediction and for its applications we present in Section 3 as it allows us to keep track of a single RAR dependence per executed load (ignoring data granularity).

To show that RAR-dependence streams are regular we measure the *memory dependence locality* of loads with RAR dependences. We define *memory-dependence-locality(n)* as the probability that the same RAR dependence has been encountered within the last *n unique* RAR dependences experienced by preceding executions of the same static load. *Memory-dependence-locality(1)* is the probability that the same RAR dependence is experienced in two consecutive executions of this load. A high value of *memory-dependence-locality(1)* suggests that a simple, "last RAR dependence encountered"-based predictor will be highly accurate. For values of *n* greater than 1, *memory-dependence-locality(n)* is a metric of the working set of RAR memory dependences per static load. Of course, a small working set does not imply regularity.

Figure 2, part (a) shows locality results for sink loads. Given a *(source, sink)* RAR dependence we define the *source* to be the earliest in program order load. From our definition of RAR dependences it follows that sink loads will typically have a single source load. The locality range (value of *n*) shown is 1 to 4 (left to right). The Y axis reports fractions over all sink loads executed. Locality is high for all programs. More than 70% of all loads experience a dependence among the four most recently encountered RAR dependences.

We also measured how locality would change had we placed a restriction on how far back we could search to find the earliest source load. Such a restriction is interesting from the perspective of history-based prediction as we need a mechanism to detect RAR dependences. To be of practical use this mechanism will have to be of finite size. Accordingly, we include locality measurements for an address window of 4K. We define an *address window* of size *s* to be the maximum number of unique addresses that can be accessed between a source and a sink load. The intuition behind this metric is a table tracking the *s* most recent addresses accessed can be used to detect memory dependences. As seen by the results of Figure 2, part (b), locality is high, in some cases higher than it was when all accesses were considered (shorter dependences seem to be more regular that distant ones).

3 Reducing Memory Latency via RAR Memory Dependence Prediction

We start by reviewing the RAW-based cloaking and bypassing techniques. We then explain how our RAR-based techniques fit under the same framework.

Memory can be viewed as an interface that programs use to express desired actions. Viewing memory as an interface allows us to separate specification from implementation: just because we have chosen to express an action via memory we do not have to implement it the exact same way. The recently proposed *cloaking* and *bypassing* methods approached memory as way of specifying inter-operation communication, that is of passing values from stores to dependent loads [15]. This specification is *implicit* and it introduces overheads which are not inherent to communication: address calculation and disambiguation. Unfortunately, caching, the current method of choice to speeding-up memory communication, cannot reduce these overheads. Moreover, these overheads may increase as pipelines grow deeper and as windows get wider. Fortunately, we can eliminate these overheards if we express memory communication *explicitly*. In an explicit specification the load and the store are given knowledge of the communication that has to occur so that they can locate each other directly. Cloaking uses RAW memory dependence prediction to create this representation on-the-fly in a program transparent way. Moreover, dependent stores and loads do not change the communicated value (ignoring sign-extension and data-type issues). They are simply used simply to pass a value that some other instruction (producer) creates to some other instruction(s) that consumes it. Bypassing extends cloaking by linking actual producers and their consuming instructions directly.

Following a similar line of thinking, we observe that another common use of memory is *data-sharing*, that is to hold data that is read repeatedly. Data-sharing is also expressed implicitly and similar overheads are introduced. This can be seen using the example of Figure 3. In part (a) two load instructions, LOAD and LOAD', are shown which at run-time access the same memory location. Part (b) shows a possible sequence of events. Initially LOAD is fetched, its address is calculated and a value is read from memory. Later on, LD' is encountered. At this point both loads have been encountered and the value is available. Yet, LD' has to calculate its address and go to memory to read the same value. Moreover, depending on whether memory dependence speculation is used, accessing the memory value may be further delayed to establish that no intervening store accesses the same memory location. It is important to note that while LD and LD' are accessing a common address every time they are encountered, this address may be *different* every time. For example, this is the case in the example of part (c) of Figure 3 where each of the elements of list "l" is accessed twice from within different functions.

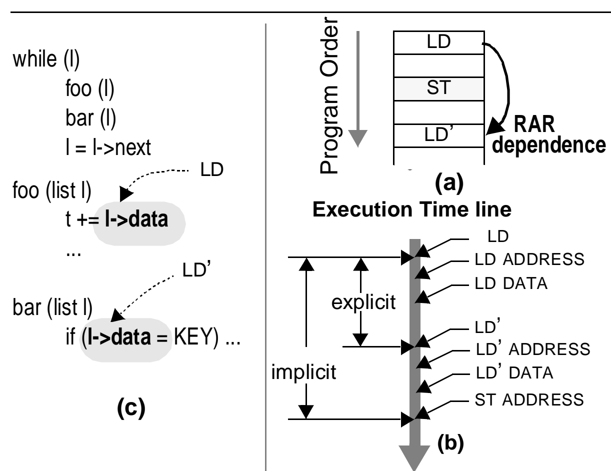

Figure 3: *An example of data-sharing. (a) Trace with two loads accessing the same memory location. (b) Time-line of execution. (c) Code with RAR dependences.*

As with memory communication, an explicit representation of data-sharing can eliminate the aforementioned overheads. In the preceding example, LD' could obtain a value by just naming LD. Creating an explicit representation of data-sharing is the goal of our RAR dependence prediction based methods. Observing that data-sharing gives rise to RAR dependences we propose PC, history-based RAR memory dependence prediction and use it to explicitly represent data-sharing. We also observe that similarly to inter-operation communication, loads that access a common memory location do not change the value they read. Accordingly we propose a RAR extension to bypassing in which consumers of loads with RAR dependences are linked directly to the earliest possible load that is predicted to access the common memory location. The effect of our RAR extensions is illustrated in Figure 1, part (b).

3.1 RAR-Based Speculative Memory Cloaking

In this section we explain how we use RAR memory dependence prediction to streamline data-sharing. Our method works as follows: the first time a RAR dependence is encountered, the identities of the dependent loads are recorded and a new name is assigned to them (i.e., with their PCs). The next time these instructions are encountered, the previously assigned name can be used to propagate a value from the first in program order load to the second. We illustrate the exact process with the example of Figure 4 where we show how an earlier detection of a RAR dependence between LOAD and LOAD' is used the second time these instructions are encountered to provide a speculative value for LOAD'. The first step is detecting the RAR dependence. This is done via the use of a *Dependence Detection Table (DDT)* [15]. The DDT is an address indexed cache the records the PC of a load or a store that accessed the corresponding address. When the first instance of LOAD calculates its address it also creates a new entry in the DDT (action (a)). Later, LOAD' may access the DDT using the same address (action b) where it will locate the entry for LOAD. At this point we have detected a RAR dependence between the two instructions. As a result, an association of the two loads with a preferably unique name, a *synonym,* is created in the *Dependence Prediction and Naming Table (DPNT)* (action 1). This is a PC-indexed table and two entries are created one for LOAD and one for LD'. When a later instance of LD is encountered (part (b)), its PC is used to access the DPNT predicting whether a RAR dependence will be observed (action 2). Provided that the dependence is predicted, storage for the synonym is allocated in the *Synonym File (SF)* (action 3). The SF is a synonym-indexed structure. Initially, the SF entry is marked as empty as no value is yet available. When LD's memory access completes, the value read is also written into the SF marking the entry as full (action 4). When LD' is encountered, its PC is used to access the DPNT and predict the RAR dependence (action 5). Using the DPNT provided synonym LD' can access the SF and obtain a speculative value (action 6). This value can be propagated to dependent instructions (action 7). Eventually, when LD' calculates its address and completes its memory access, the value read from memory can be used to verify whether speculative value was correct (action 8). If it was, speculation was successful. If not, value misspeculation occurs. While we assumed that LD's memory access completes before LD' is encountered, this technique is useful even when this is not so.

We have deliberately used the same support structures as in the original RAW-based cloaking. In fact, the two techniques are virtually identical provided that we treat the first load in a RAR dependence as the producer of the memory value. However, while in RAW-based cloaking the value becomes available as soon as the store receives it from the instruction that produces it, in RAR-based cloaking the value has to be fetched from memory by the first load. These observations suggest that our RAR-based cloaking technique can be implemented as a surgical extension to RAW-based cloaking. For this purpose we need to record loads in the DDT. Moreover, we need to mark loads as producers in the DPNT. For this we use two predictors per entry, one for consumer prediction and one for producer prediction. In the DDT we chose to record loads only when no preceding store has been recorded for the same address. Moreover, we record a load in the DDT only when no other load has been recorded for the same address. This is done to annotate

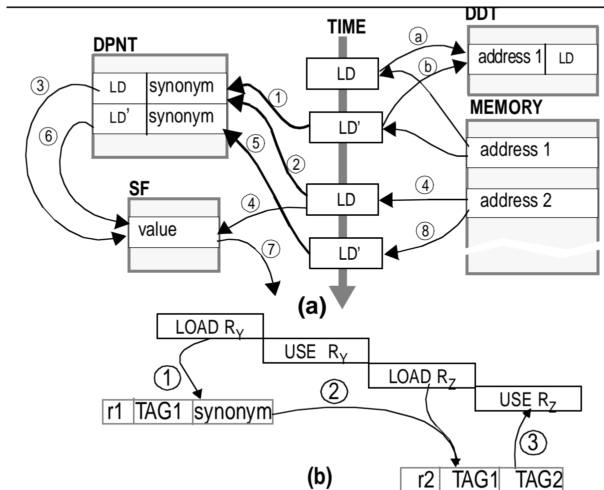

Figure 4: *RAR-based speculative memory cloaking (part (a)) and bypassing (part (b)).*

the earliest in program order load as the producer of a value for cloaking purposes.

At this point we can explain why our RAR-based method can be used to predict some of the loads that have RAW dependences with distant stores. The size of the DDT limits how far we can search to locate the source instruction for both RAW and RAR dependences. When a load has a dependence with a distant store it is likely that the latter will be evicted from the DDT long before the load is encountered. Consequently, the RAW dependence will go undetected and RAW-based cloaking will not be performed. However, if the load has RAR dependences with not so distant loads, these dependences may be detected and subsequently used to predict the load's value using RAR-based cloaking.

3.2 RAR-Based Speculative Memory Bypassing

The process of RAW-based bypassing is shown in part (a) of Figure 1. As shown, bypassing speculatively converts a DEF-STORE-LOAD-USE dependence chain into a DEF-USE one, in effect bypassing the store and load instructions. Consequently, the value can flow directly from the producer (DEF R_X) to the consumer (USE R_Y). The goal of our RAR-based extension to bypassing is shown in part (b) of Figure 1. We assume that a RAR dependence exists between "LOAD R_Y" and "LOAD R_Z". While RAR cloaking will allow "LOAD R_Z" to obtain a speculative value by naming "LOAD R_Y", its consumer, "USE R_Z", will have to wait until "LOAD R_Y" propagates this value. With our method, "USE R_Z" is speculatively linked directly to "LOAD R_Y". As with cloaking, the proposed method can be implemented as an extension to the RAW-based bypassing. This can be done by treating the oldest in program order load of a RAR dependence similarly to a store of a RAW dependence. The only difference is that this "producing" load cannot be eliminated. Figure 4, part (b) illustrates how the cloaking provided synonym is used to propagate the target register tag (TAG1) of "LD R_Y" to "USE R_Z".

4 Related Work

An obvious alternative to cloaking is register allocation which eliminates load and store instructions altogether. However, register allocation is not always possible for numerous reasons ranging from fundamental limitations (e.g., addressability) to practical considerations (e.g., register file size, programming conventions and legacy codes). Cloaking and bypassing are architecturally invisible. As such, we may deploy them only when justified by the underlying technological tradeoffs. Moreover, they may capture dynamic dependence behavior.

Numerous software and hardware address-prediction techniques have been used to reduce load access latency, e.g., [1,2,6,9,18,4,3]. Cloaking is orthogonal to address-prediction-based techniques as it does not require a predictable access pattern. A technique closely related to cloaking is *load value prediction* [12], a special case of value prediction [11,7]. Cloaking does not directly predict the loaded value, rather it predicts its producer or another load that also accessed the same location. This property may be invaluable for programs with large data sets.

Moshovos, Breach, Vijaykumar and Sohi introduced RAW memory dependence prediction for scheduling loads [14]. Tyson and Austin [20] and Moshovos and Sohi [15,16] introduced RAW-based cloaking. The *memory renaming* proposal of Tyson and Austin combines cloaking with value prediction. Lipasti's *Alias prediction* [10] is also similar to cloaking. Moshovos and Sohi proposed RAW-based speculative memory bypassing [15]. Jourdan, Ronen, Bekerman, Shomar and Yoaz proposed a similar method [8] where address information and prediction is used to eliminate loads and to increase coverage. Reinman, Calder, Tullsen, Tyson and Austin investigated a software-guided cloaking approach [17].

5 Evaluation

This section is organized as follows: In Section 5.1 we describe our methodology. The first step in using cloaking is building dependence history. Accordingly, in Section 5.2 we measure the fraction of memory dependences observed as a function of DDT size. In Section 5.3 we investigate an aggressive cloaking mechanism and study its accuracy. In Sections 5.4 through 5.5 we present a characterization of the speculated loads by considering their address and value locality characteristics. In Section 5.6, we measure the performance impact of a combined cloaking and bypassing mechanism.

5.1 Methodology

We have used the SPEC'95 programs which we compiled for the MIPS-I architecture using GNU's *gcc* compiler version 2.7.2 (flags: -O2 -funroll-loops -finline-functions). We converted FORTRAN codes first to C using AT&T's *f2c* compiler. To attain reasonable simulation times we modified the standard *train* or *test* inputs, and we used sampling for some programs. Table 5.1 reports the dynamic instruction count, the fraction of loads and stores and the sampling ratio per program. We note that when we simulated our cloaking/bypassing mechanisms using unmodified input data sets from the SPEC95 suite the resulting accuracy was close, often better than that observed with the modified input data sets. *We used sampling only for the timing experiments of section 5.6. We did not use sampling for 099.go, 126.gcc, 130.li, 132.ijpeg, 147.vortex, 107.mgrid and 141.apsi.* For the rest of the benchmarks we chose sampling ratios that resulted in roughly 100M instructions being simulated in timing mode. The observation size is 50,000

instructions. We report sampling ratios under the "SR" columns as "timing:functional" ratios. For example, an 1:2 sampling ratio amounts to simulating 50,000 instructions in timing mode and then switching to functional simulation for the next 100,000 instructions. During functional simulation the I-cache, D-cache, and branch predictors are simulated. Even when sampling was used, the accuracy of all evaluated techniques was very close, often identical to that measured when the whole program was simulated using functional simulation. In our evaluation we will use the abbreviations shown under the "Ab." column of Table 5.1.

Program	*Ab.*	*IC*	*Loads*	*Stores*	*SR*
SPECint'95					
099.go	***go***	133.8	20.9%	7.3%	N/A
124.m88ksim	***m88***	196.3	18.8%	9.6%	1:1
126.gcc	***gcc***	316.9	24.3%	17.5%	N/A
129.compress	***com***	153.8	21.7%	13.5%	1:2
130.li	***li***	206.5	29.6%	17.6%	N/A
132.ijpeg	***ijp***	129.6	17.7%	8.7%	N/A
134.perl	***per***	176.8	25.6%	16.6%	1:1
147.vortex	***vor***	376.9	26.3%	27.3%	N/A
SPECfp'95					
101.tomcatv	***tom***	329.1	31.9%	8.8%	1:2
102.swim	***swm***	188.8	27.0%	6.6%	1:2
103.su2cor	***su2***	279.9	33.8%	10.1%	1:3
104.hydro2d	***hyd***	1,128.9	29.7%	8.2%	1:10
107.mgrid	***mgd***	95.0	46.6%	3.0%	N/A
110.applu	***apl***	168.9	31.4%	7.9%	1:1
125.turb3d	***trb***	1,666.6	21.3%	14.6%	1:10
141.apsi	***aps***	125.9	31.4%	13.4%	N/A
145.fpppp	***fp****	214.2	48.8%	17.5%	1:2
146.wave5	***wav***	290.8	30.2%	13.0%	1:2

Table 5.1: *Benchmark Execution Characteristics. Instruction counts ("IC" columns) are in millions.*

The simulators we used are modified versions of the Multiscalar timing simulator. Our base processor is capable of executing up to 8 instructions per cycle and is equipped with a 128-entry instruction window. It takes 5 cycles for an instruction to be fetched, decoded and placed into the re-order buffer for scheduling. It takes one cycle for an instruction to read its input operands from the register file once issued. Integer functional unit latencies are 1 cycle except for multiplication (4 cycles) and division (12 cycles). Floating-point functional unit latencies are as follows: 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. An 128-entry load/store scheduler (load/store queue) capable of scheduling up to 4 loads and stores per cycle is used to schedule load/store execution. It takes at least one cycle after a load has calculated its address to go through the load/store scheduler which implements *naive memory dependence speculation* [14]. That is: (1) a load may access memory even preceding store addresses are unknown, (2) a load will wait for preceding stores that are known to write to the same address, (3) stores post their address even when their data is not yet available, and (4) stores may post their data or address out-of-order. We have found that for our centralized window processor model this speculation policy offers performance very close to that possible with ideal speculation [13]. The base memory system comprises: (1) a 128-entry write buffer, (3) a non-blocking 32Kbyte/16 byte block/4-way interleaved/2-way set associative L1 data cache with 2 cycle hit latency, (4) a 64K/16 byte block/8-way interleaved/2-way set associative L1 instruction cache with 2 cycle hit latency, (5) a unified 4Mbyte/8-way set associative/128 byte block L2 cache with a 10 cycle hit latency, and (6) an infinite main memory with 50 cycles miss latency. Miss latencies are for the first word accessed. Write buffers of 32 blocks each are included between L1 and L2, and between L2 and main memory. Additional words incur a latency of 1 cycle (L2) or 2 cycles (main memory). All write buffers perform write combining and hits on miss are simulated for loads and stores. For branch prediction we use a 64-entry call stack and a 64k-entry combined predictor that uses a 2-bit counter selector to choose among a 2-bit counter-based and a GSHARE predictors.

In all experiments we used a level of indirection (i.e., synonym) to track multiple RAW and RAR dependences per load and store. This is necessary, as some loads experience different dependences through different control paths. Instead of using the full merge algorithm assumed by Moshovos and Sohi [15], we used the incremental algorithm Chrysos and Emer proposed in the context of memory dependence speculation/synchronization [5]. These methods attack scenarios where a dependence is detected between loads or stores that have different synonyms already assigned. For example, consider the following sequence: ST_1 A, LD_1 A, ST_2 B, LD_2 B, ST_1 C, LD_2 C. Initially ST_1 and LD_1 will be assigned a synonym, say X, because they both access address A. Then ST_2 and LD_2 will be assigned a different synonym, say Y, because they access address B. When the (ST_1,LD_2) dependence is encountered the two instructions have different synonyms already assigned to them. We have found that in such cases and if we are to use a common policy for all loads, it is best to *always* merge all dependences into the same communication group rather than *never* doing so. In the original cloaking proposal, one of the two synonyms is selected (e.g., X) and all instances of the second one (e.g., Y) are replaced in the DPNT. This action requires an associative lookup/update in the DPNT. Chrysos and Emer proposed just replacing the synonym of largest value and only for the corresponding instruction (e.g., if $X > Y$, then ST_1's synonym will be replaced with Y). Because of the bias in the synonym selection, eventually all relevant instructions will be given the same synonym. No noticeable difference in accuracy was observed between the two methods. Finally, we did not provide explicit support for dependences between instructions that access different data types as such dependences are rare in the SPEC95 benchmarks. This might not be the case for other programs. The original RAW-based cloaking and bypassing proposal discusses potential support for such dependences [15].

5.2 Memory Dependence Detection

In this section, we measure the fraction of memory dependences that is visible with various DDT sizes. These measurements provide a first indication of the fraction of loads that can obtain a speculative value via cloaking. Figure 5 reports the fraction of dynamic (committed) loads with detectable RAW or RAR dependences as a function of DDT size (range is 32 to 2K

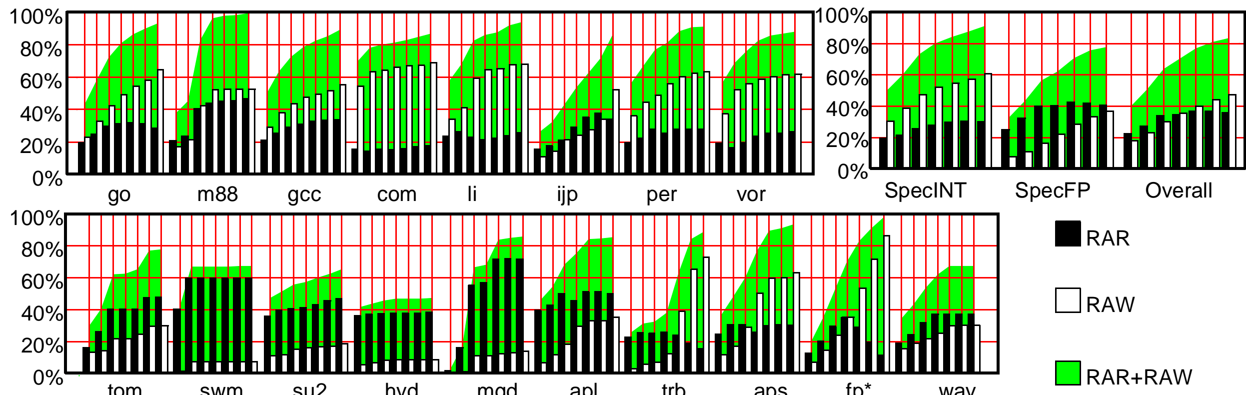

Figure 5: *Fraction of loads with RAW or RAR dependences as a function of DDT size. Range is 32 to 2K in power of 2 steps.*

entries and we use LRU replacement policy). Shown is the total number of loads with dependences (grey shaded area) and a breakdown in terms of the dependence type (RAW or RAR).

The averaged results (upper right) show that a large fraction of loads get their value via a dependence that is visible even with the smaller DDTs. Overall, dependences are more frequent for the integer codes. The relative fractions of RAR and RAW dependences are dissimilar for the two classes of programs. In integer codes and for the smaller DDT sizes, RAW dependences are almost twice as frequent as RAR dependences are. In the floating point codes the roles are almost reversed. It seems that Fortran codes are dominated by a large number of variables with long lifetimes that are not register allocated. As we move toward larger DDT sizes, more RAW dependence are detected. While RAR dependence frequency also increases for up to DDTs of 512 entries, virtually no increase is observed for larger DDTs. We even observe a decrease in RAR dependence frequency between 1K and 2K DDTs for some floating-point codes. The increased frequency of RAW dependences is the cause: some of the RAR dependences are among loads that also have a RAW dependence with a distant store. When smaller DDTs are used the store is evicted from the DDT due to limited space.

The results of this section suggest that a DDT of moderate size (e.g., 128 entries) can capture dependences for a large fraction of loads (roughly 70% and 60% for the integer and floating-point programs respectively). Moreover, a significant fraction of loads have a visible RAR dependence but no visible RAW dependence (e.g., 25% (integer) and 40% (floating-point) of all loads for the 128-entry DDT). For the rest of the evaluation we focus on configurations that use a 128-entry DDT. We have found that this table yields accuracy close of often better to that achieved with larger DDTs.

5.3 Cloaking Coverage And Misspeculation Rates

In this section, we measure the accuracy of two cloaking predictors. We use two metrics: *coverage* and *misspeculation rate* both measured as a fraction over all executed loads. *Coverage* is the fraction of loads that get a correct value via cloaking. The complement of coverage, the fraction of loads that get an incorrect value, is the *misspeculation rate*. For the purposes of this study we assume infinite DPNTs and evaluate predictors with the following two confidence mechanisms: (1) non-adaptive 1-bit, and (2) a 2-bit automaton. The second confidence mechanism enables cloaking as soon as a dependence is detected. However, once a misprediction is encountered it requires two correct predictions before allowing a predicted value to be used again. We include results for the non-adaptive predictor as it provides a rough upper bound on coverage.

Figure 6 reports cloaking coverage (part (a)) and misprediction rates (part (b)). Two bars are shown, one for each confidence mechanism: the left one is for the 1-bit non-adaptive, while the right one is for the aforementioned 2-bit automaton. A breakdown in RAW (grey) and RAR (white) dependences is also shown. We observe that on the average RAR dependences offer roughly an additional 20% (integer) and 30% (floating-point) of correctly speculated loads. We also observe that only a minor loss in coverage is incurred when the adaptive predictor is in place. As the results on misspeculation rates (part (b)) show, this loss comes at the benefit of a drastic reduction in misspeculations. We can observe that for the integer codes RAR misspeculations are frequent and in some cases even more frequent than RAW dependences. For the floating point programs, RAR dependences are either the sole source of misspeculations 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 predictor. On average the adaptive predictor reduces misspeculations by almost an order of magnitude compared to the non-adaptive predictor. The misspeculation rates are 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) comes from RAW dependences. In the rest of the evaluation we restrict our attention to the adaptive predictor.

5.4 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 in two consecutive executions. We present these measurements to offer additional insight on the type of loads that are correctly handled by cloaking. The results are shown in Figure 7, part (a). 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. We breakdown the left bar into three categories depending on whether a RAW, a RAR or no dependence is detected by our 128-entry DDT. We can observe

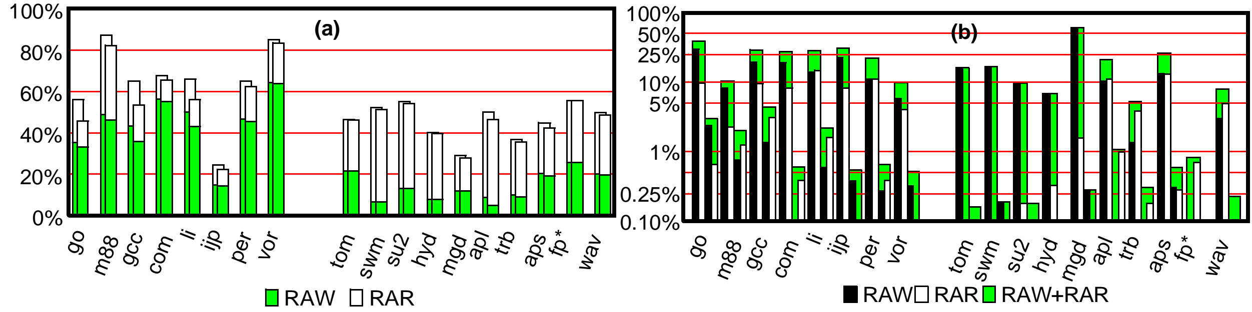

Figure 6: *Breakdown of cloaking accuracy per dependence type: (a) coverage, and (b) misprediction rates (logarithmic Y axis). Two predictors are shown per program (see text). Percentages are over all loads.*

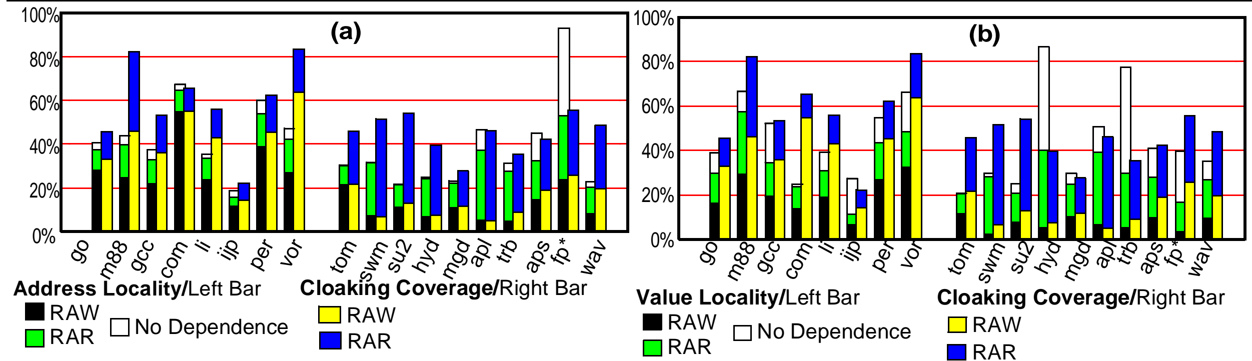

Figure 7: *(a) Address Locality breakdown. (b) Value Locality breakdown.*

that many loads covered by cloaking do not exhibit address locality. We can also observe that with the exception of 145.fpppp, there are very few loads that exhibit address locality but have no dependence (145.fpppp exhibits similar behavior if a larger DDT is used).

5.5 Value Locality/Prediction Measurements

In this section, we measure the value locality of loads and its correlation to cloaking coverage. We do so as value prediction can also be used to predict a load value, possibly earlier than cloaking would allow. Figure 7, part (b) reports the fraction of loads that exhibit value locality alongside with a breakdown of loads that get a correct value via cloaking. As in the previous section we provide a breakdown of the loads that exhibit value locality based on whether they have a dependence detected. For most programs, cloaking coverage is higher than value locality. Value locality is higher only for 132.ijpeg, 104.hydro2d, 110.applu and 125.turb3d. Moreover, cloaking predicts more of the loads with dependences.

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. For this experiment we simulated a fully-associative last-value predictor with 16K entries. The cloaking mechanism we use has a 16K DPNT, a 128-entry DDT and a 2K set associative synonym file. All structures are assumed to be fully-associative. The results are shown in Table 5.1. We also present a breakdown of the values obtained via cloaking/bypassing in terms of the dependence type. We can observe that for most programs, value prediction captures some loads that cloaking/bypassing does not and vice versa. Moreover, 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. While context-based value predictors could be used to increase load value prediction coverage, cloaking offers a concise way of representing information for prediction purposes for a large fraction of loads. These observations suggest a potential synergy of the two techniques.

	Cloaking/Bypassing			VP
	RAW	**RAR**	**Total**	
go	23.43%	5.75%	29.18%	5.29%
m88	14.23%	10.62%	24.85%	1.88%
gcc	18.15%	5.89%	24.04%	8.01%
com	41.18%	0.99%	42.18%	0.22%
li	31.08%	1.08%	32.17%	6.14%
ijp	8.67%	5.25%	13.93%	11.24%
per	21.72%	1.57%	23.29%	7.82%
vor	29.52%	3.33%	32.85%	5.03%
tom	10.22%	15.35%	25.58%	0.24%
swm	6.43%	19.98%	26.41%	0.37%
su2	7.18%	25.89%	33.08%	2.67%
hyd	3.02%	1.29%	4.31%	49.94%
mgd	2.34%	0.43%	2.77%	2.60%
apl	3.18%	8.29%	11.46%	12.60%
trb	2.27%	0.55%	2.82%	41.94%
aps	8.85%	4.47%	13.34%	9.67%
fp*	22.46%	17.87%	40.34%	18.17%
wav	10.08%	12.84%	22.92%	5.94%

Table 5.1: *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).*

5.6 Performance Impact

In this section, we evaluate the performance impact of a combined cloaking and bypassing mechanism. The rest of this section is organized as follows: In Section 5.6.1 we describe the cloaking/bypassing mechanism we simulated. In Section 5.6.2 we measure how performance varies when cloaking/bypassing is used for two misspeculation handling models. We also measure the improvements obtained by augmenting cloaking/bypassing with our RAR dependence based techniques. In Section 5.6.2, we measure the performance impact of our techniques.

5.6.1 Configuration

The cloaking/bypassing mechanism we used comprises: (1) a 128-entry fully-associative DDT with word granularity, (2) an 8K, 2-way set-associative DPNT, and finally, (3) an 1K, 2-way set associative synonym file. Figure 8 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 via the DDT. Synonym file updates and DPNT updates also occur at commit time. Dependence predictions are initiated as soon as instructions enter the decode stage. For bypassing, loads and stores that are predicted as producers associate the actual producer of the desired value with their synonym via *synonym rename table* (SRT) entry. That is, SRT entries associate synonyms with physical registers. Loads that are predicted as consumers inspect the SRT and the SF in parallel to determine the current location of the their synonym. If an SRT entry is found, the synonym resides in the physical register file as the corresponding load or store has yet to commit. Otherwise, the synonym is in the SF. At most 8 predictions can be made per cycle and at most 8 instructions can be scheduled for cloaking or bypassing per cycle.

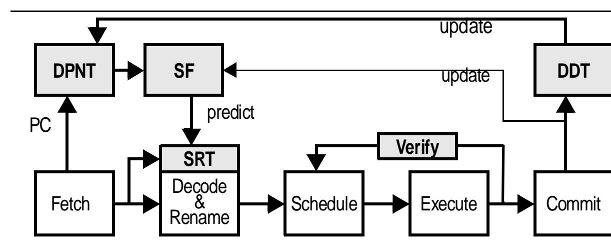

Figure 8: *Integrating cloaking/bypassing into a pipeline.*

Misspeculations are signalled only when an instruction has actually read an incorrectly speculated value. We have experimented with two misspeculation recovery mechanisms. The first is *selective invalidation*. This mechanism re-executes only those instructions that used incorrect data. The second is *squash invalidation* and works by invalidating all instructions starting from the one that was mispeculated. These instructions have to be re-fetched from scratch. We also experimented with an oracle mechanism that does not speculate when this would result in misspeculation. We found that selective invalidation offers performance similar to such a mechanism.

A challenge shared by most value speculative techniques is *data speculation resolution*, that is how quickly we can establish that speculative values are correct. Moreover, care must be taken to avoid destructive interference with branch prediction [19]. We assumed the ability to resolve all speculation in a register dependence chain as soon as its input values are resolved. Whether such a mechanism is practical is still an open question. Finally, we disallow control resolution on branches with value speculative inputs.

5.6.2 Performance with a Cloaking/Bypassing Mechanism

Figure 9 we measure how performance varies when cloaking/bypassing is used. Reported is the speedup or slowdown with respect to the base processor that uses no cloaking/bypassing. Four bars are shown. The two on the left are with selective invalidation. The dark bar is for the original RAW-based techniques while the grey bar is for our extended mechanism. The other two bars report performance with squash invalidation (grey for RAW-based and white for RAW+RAR-based cloaking). Using squash invalidation rarely results in performance improvements. In contrast, speedups are observed for all programs when selective invalidation is used. Comparing RAW-based cloaking/bypassing with our proposed RAW+RAR-based mechanism we observe that for most programs further improvements are attained. In some cases the improvements are significant in absolute terms. In relative terms the additional improvements are considerable especially when we take into account that they come at a small cost over the original RAW-based cloaking/bypassing. On the average performance improvements are up to 6.44% (integer) and 4.66% (floating-point) from 4.28% and 3.20% respectively. An anomaly is observed for two cases where our extended mechanism results in somewhat lower performance. This is because we use a common DDT for both RAW and RAR dependences. As a result some RAW dependences are not detected because stores get evicted by loads to different addresses. Using separate DDTs one for stores and one for loads eliminates this anomaly.

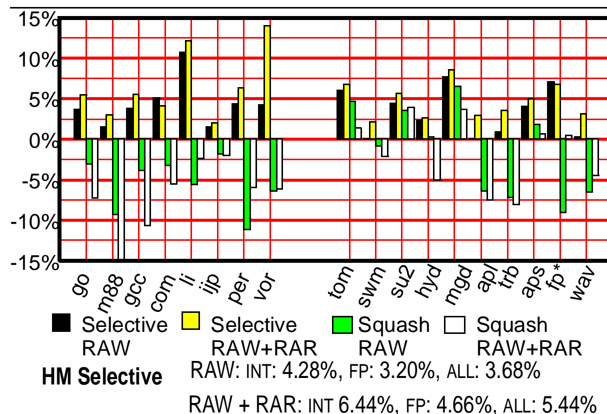

Figure 9: *Performance of RAW and RAW+RAR cloaking/bypassing with two misspeculation handling mechanisms.*

In Figure 10 we report speedups for a processor that does not speculate on memory dependences (i.e., loads wait for all preceding stores to calculate their address). We do so for completeness and as most studies in value speculative techniques assume such a configuration. It can be seen that in most cases speedups are significantly higher (often double) compared to Figure 9 were the base processor uses memory dependence speculation (see Section 5.1). There are cases where the speedups are smaller. This is the result of having loads wait for the

addresses of all preceding stores. This results in a longer critical path comprised mostly of loads which cloaking/bypassing cannot attack.

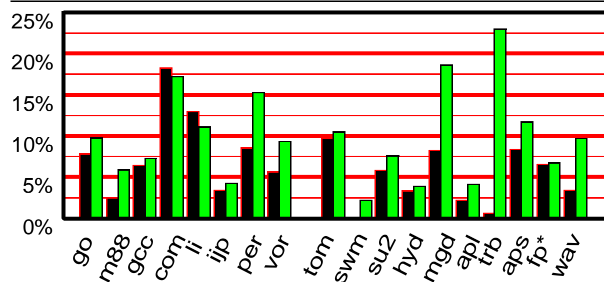

Figure 10: *Speedup when no memory dependence speculation is used. Left bar: RAW-based. Right bar: RAW+RAR-based.*

6 Conclusion

We have identified that typical programs exhibit highly regular RAR memory dependence streams and exploited this property by introducing history-based RAR memory dependence prediction. For most programs more than 80% of all loads with a RAR dependence experienced the same RAR dependence as the last time they were executed. We used this prediction to develop two memory latency reduction techniques: RAR-based cloaking and bypassing. We showed how these techniques can be implemented as surgical extensions to the recently proposed RAW-based cloaking and bypassing.

On the average, our RAR extensions provide correct speculative values for an additional 20% (integer codes) and 30% (floating-point codes) of all loads. This increase is significant compared to the 45% (integer codes) and 15% (floating-point codes) of loads that get a correct speculative value via the original, RAW-dependence-based cloaking and bypassing. We studied the performance of the resulting mechanism and its interaction with two misspeculation handling techniques and found that selective invalidation is necessary for the given predictor. We observed average speedups of 6.44% (integer) and 4.66% (floating point). For the same configuration the speedups of the original RAW-based cloaking/bypassing are 4.28% and 3.20% respectively. These improvements come at virtually no cost. When we used a base configuration that does not use memory dependence speculation our techniques yield speedups of 9.8% (integer) and 6.1% (floating-point). We also found that the combined RAW- and RAR-dependence-based cloaking/bypassing mechanism offers in most cases superior accuracy compared to last-value load value prediction.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and by an equipment donation from Intel Corporation.

References

[1] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Fast address calculation. In *Proc. International Symposium on Computer Architecture-22*, June 1995.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proc. Annual International Symposium on Microarchitecture-28*, Nov. 1995.

[3] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proc. International Symposium on Computer Architecture-26*, May 1999.

[4] B.-C. Cheng, D. A. Connors, and W.-M. Hwu. Compiler-directed early load-address generation. In *Proc. Annual International Symposium on Microarchitecture-31*, Dec. 1998.

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

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

[7] F. Gabbay and A. Medelson. Speculative Execution Based on Value Prediction. Technical report, TR-1080, EE Dept., Technion-Israel Institute of Technology, Nov. 1996.

[8] 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. Annual International Symposium on Microarchitecture-31*, Dec. 1998.

[9] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *Proc. International Conference on Supercomputing-11*, July 1997.

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

[11] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. on Annual International Symposium on Microarchitecture-29*, Dec. 1996.

[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems-VII*, Oct. 1996.

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

[14] 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.

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

[16] A. Moshovos and G. S. Sohi. Speculative memory cloaking and bypassing. *International Journal of Parallel Programming*, Oct. 1999.

[17] 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.

[18] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.

[19] A. Sodani and G. S. Sohi. Understanding the Differences Between Value Prediction and Instruction Reuse. In *Proc. Annual International Symposium on Microarchitecture-31*, Dec. 1998.

[20] G. S. Tyson and T. M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. Annual International Symposium on Microarchitecture-30*, Dec. 1997.