Reducing Memory Latency via Read-after-Read Memory Dependence Prediction

Andreas Moshovos, *Member*, *IEEE*, and Gurindar S. Sohi, *Senior Member*, *IEEE*

Abstract—We observe that typical programs exhibit highly regular read-after-read (RAR) memory dependence streams. To exploit this regularity, we introduce read-after-read (RAR) memory dependence prediction. This technique predicts whether: 1) A load will access a memory location that a preceding load accesses and 2) exactly which this preceding load is. This prediction is done without actual knowledge of the corresponding memory addresses. We also present two techniques that utilize RAR memory dependence prediction to reduce memory latency. In the first technique, a load may obtain a value by naming a preceding load with which an RAR dependence is predicted. The second technique speculatively converts a series of $\mathrm{LOAD_1}$-$\mathrm{USE_1}, \ldots, \mathrm{LOAD_N}$-$\mathrm{USE_N}$ chains into a single $\mathrm{LOAD_1}$-$\mathrm{USE_1} \ldots \mathrm{USE_N}$ producer/consumer graph. This is done whenever RAR dependences are predicted among the $\mathrm{LOAD_i}$ instructions. Our techniques can be implemented as small extensions to the previously proposed read-after-write (RAW) dependence prediction-based speculative memory cloaking and speculative memory bypassing. On average, our RAR-based techniques provide correct values for an additional 20 percent (integer codes) and 30 percent (floating-point codes) of all loads. Moreover, a combined RAW- and RAR-based cloaking/bypassing mechanism improves performance by 6.44 percent (integer) and 4.66 percent (floating-point) over a highly aggressive dynamically scheduled superscalar processor that uses naive memory dependence speculation. By comparison, the original RAW-based cloaking/bypassing mechanism yields improvements of 4.28 percent (integer) and 3.20 percent (floating-point). When no memory dependence speculation is used, our techniques yield speedups of 9.85 percent (integer) and 6.14 percent (floating-point).

Index Terms—Memory dependence prediction, load, cache, dynamic optimization.

1 Introduction

Modern high-performance processors use techniques that rely on regularities in typical program behavior to enhance performance. A well-understood example is caching. Caching exploits the tendency of programs to access the same or in-close-proximity memory data. It does so to approximate a fast and large memory (that is either impossible to build or prohibitively expensive), using smaller yet faster memories. Other examples are branch and value prediction. The accumulated experience with regularity-based techniques suggests that, in our search for higher performance, it may be useful to further study typical program behavior, trying to identify other previously unknown regularities. Such regularities could then be exploited to our advantage. Following this rationale, in this paper, we identify that typical programs exhibit highly regular "read-after-read" (RAR) memory dependence streams. An RAR dependence exists between two loads if they access the same address and no store to the same address appears between them in the sequential execution order. We have found that if, at some point during execution, two loads are RAR dependent, then, with high probability, these loads will again be RAR dependent soon, even though they may be accessing a *different* address.

- *A. Moshovos is with the Electrical and Computer Engineering Department, University of Toronto, 10 King's College Rd., Toronto, Ontario, M5S 3G4, Canada. E-mail: moshovos@eecg.toronto.edu.*
- *G.S. Sohi is with the Computer Sciences Department, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706.*

Manuscript received 4 Jan. 2000; revised 14 Feb. 2001; accepted 16 May 2001. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110624.

We also present techniques that exploit this regularity to our advantage. In particular, 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 an RAR dependence is used to predict the dependence the next time the same loads are encountered. We demonstrate how to use this prediction to create a new name space free of aliases through which loads can get speculative values. In our technique, a load can get a value by identifying a preceding load that reads the desired value (i.e., an RAR dependence exists with that load). Using instruction-address-based (PC-Based) prediction, this identification takes place early in the pipeline without actual knowledge of memory addresses. This is a value speculative technique and requires verification through memory (either by accessing the memory value or by establishing that no memory dependences were violated). To further reduce memory latency, we propose using RAR memory dependence prediction to transform a number of $\mathrm{LOAD_1}$-$\mathrm{USE_1}, \ldots, \mathrm{LOAD_N}$-$\mathrm{USE_N}$ chains into a single $\mathrm{LOAD_1}$-$\mathrm{USE_1} \ldots \mathrm{USE_N}$ producer/consumer graph. As a result, the first load that accesses a memory location propagates its value to the consumers of all its RAR-dependent loads. This is also a speculative technique and is applicable only when all dependent loads appear within the current instruction window.

An advantage of our techniques is that they can be implemented as small extensions to the previously proposed *speculative memory cloaking (cloaking)* and *speculative memory bypassing (bypassing)*, respectively [22]. While we discuss the details in Sections 3.1 and 3.2, we present an

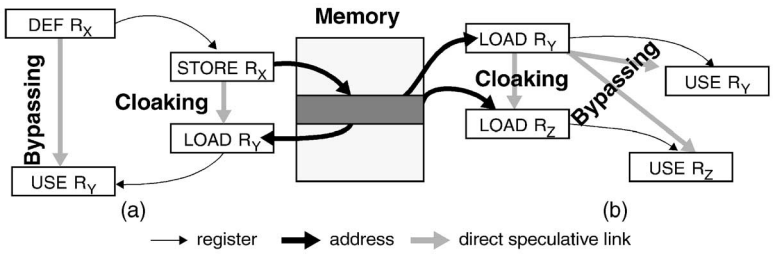

Fig. 1. Speculative memory cloaking and bypassing: (a) Original proposal: exploiting read-after-write dependences. (b) Newly proposed techniques: exploiting read-after-read dependences.

overview of the four techniques in Fig. 1. Fig. 1a shows the original cloaking and bypassing, while Fig. 1b shows our RAR memory dependence-based techniques. As originally proposed, cloaking uses history-based read-after-write (RAW) memory dependence prediction to speculatively pass values from stores to loads. Bypassing, an extension of cloaking, further reduces memory latency by converting DEF-STORE-LOAD-USE dependence chains into direct, albeit speculative, DEF-USE ones. As shown in Fig. 1b, our RAR memory dependence prediction-based cloaking and bypassing complement their RAW counterparts, increasing overall coverage. Our techniques predict loads that the original cloaking and bypassing cannot. These are loads that do not experience RAW dependences. However, the utility of our techniques also extends to loads that have RAW dependences with *distant* stores. Such RAW dependences often escape detection due to the limited scope of the underlying detection mechanisms. A more detailed discussion of this issue is given in Section 3.1.

The contributions of this paper 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 [18] and show that the two approaches are complementary.

The rest of this paper is organized as follows: In Section 2, we demonstrate that regularity exists in the RAR memory dependence stream of typical programs. In Section 3, we discuss the rationale for our RAR memory dependence prediction-based methods for reducing memory latency. In Section 3.1, we present our RAR version of cloaking and discuss how it can be implemented as an extension of RAW cloaking. In Section 3.2, we do the same for RAR memory dependence prediction-based bypassing. In Section 4, we review previous work. In Section 5, we evaluate the accuracy and performance of our techniques. Finally, in Section 6, we summarize our findings.

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 an 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 $(\mathrm{PC}_1, \mathrm{PC}_2)$ 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 $\mathrm{LD}_1\ \mathrm{A}, \mathrm{LD}_2\ \mathrm{A}, \mathrm{LD}_3\ \mathrm{A}$, we will account for the $(\mathrm{LD}_1\ \mathrm{A}, \mathrm{LD}_2)$ and $(\mathrm{LD}_1\ \mathrm{A}, \mathrm{LD}_3\ \mathrm{A})$ dependences only and not for the $(\mathrm{LD}_2\ \mathrm{A}, \mathrm{LD}_3\ \mathrm{A})$ dependence. This definition is convenient for RAR dependence prediction and for its applications, which we present in Section 3, as it allows us to keep track of a single RAR dependence per executed load (ignoring data granularity issues, i.e., a load that reads two bytes, each of which was read separately by different preceding loads).

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 that the dependences are predictable.

Fig. 2a shows locality results for sink loads for the SPEC95 programs (see Table 1 for additional information on the programs used in this study). 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

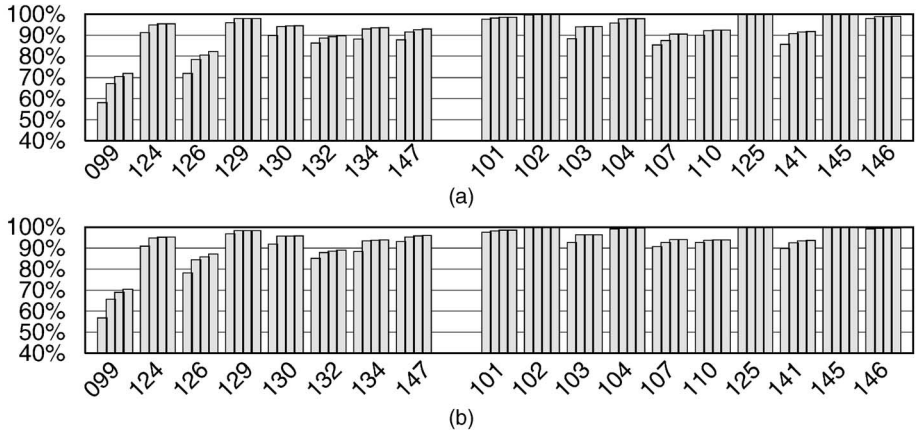

Fig. 2. Memory dependence locality of read-after-read dependences for the SPEC95 programs (X axis). Details about the programs are given in Table 1. Locality range shown is 1 to 4. (a) Infinite address window. (b) 4K entry address window.

executed. Locality is high for all programs. More than 70 percent 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 a 4K address window. We define s, the size of an *address window* to be the maximum number of *unique* addresses that can be accessed between a source and a sink load. The metric is inspired by a practical memory dependence detection mechanism, where a table tracking the s most recent addresses accessed is used to detect memory dependences. As seen by the results of Fig. 2b, locality is high, in some cases higher than it was when all accesses were considered. This implies that shorter dependences seem to be more regular than distant ones.

3 Reducing Memory Latency via RAR Memory Dependence Prediction

In this section, we review the RAW-based cloaking and bypassing techniques and then explain how our RAR-based techniques fit under the same framework.

Memory can be viewed as a storage mechanism or 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 in exactly the same way. The previously proposed *cloaking* and *bypassing* methods approached memory as a way of specifying interoperation communication, that is, of passing values from stores to dependent loads [22]. 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

TABLE 1
Benchmark Execution Characteristics

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

Instruction counts ("IC" columns) are in millions.

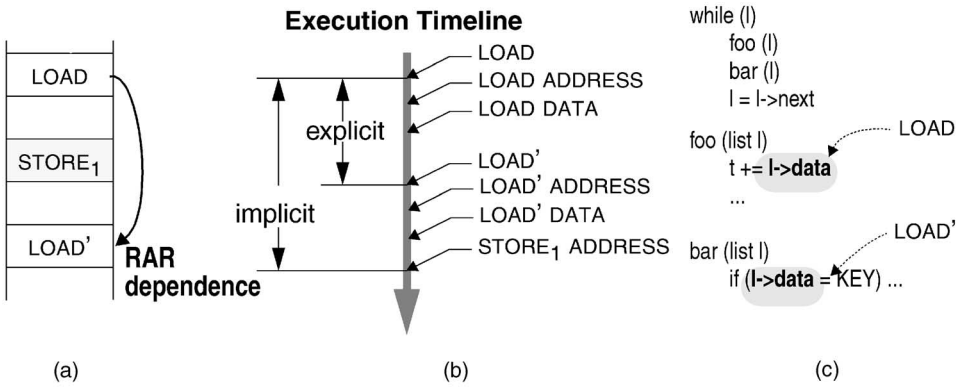

Fig. 3. An example of data-sharing. (a) Program segment with two loads that will access a common memory location communicate. (b) Time-line of execution. (c) Code with RAR dependences.

eliminate these overheards if we express memory communication *explicitly*. In an explicit specification, the load and the store can locate each other directly. Cloaking uses RAW memory dependence prediction to create this representation on-the-fly in a program transparent way. As a result, store-to-load communication may happen before address calculation and disambiguation. It is possible to further reduce by communication latency by exploiting the fact that dependent stores and loads do not change the communicated value (ignoring sign-extension and data-type issues). They are simply used 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 Fig. 3. In Fig. 3a, two load instructions, LOAD and LOAD', are shown which, at runtime, access the same memory location. Fig. 3b shows a possible sequence of events. Initially, LOAD is fetched, its address is calculated, and a value is read from memory. Later on, LOAD' is encountered. At this point, both loads have been seen and the value is available. Yet, LOAD' 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 LOAD and LOAD' 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 Fig. 3c, where each of the elements of list "l" is accessed twice from within different functions.

As with memory communication, an explicit representation of data-sharing can eliminate the aforementioned overheads. In the preceding example, LOAD' could obtain a value by just naming LOAD. 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 interoperation communication, loads that access a common memory location do not change the value they read. Accordingly, we propose an 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 Fig. 1b.

3.1 RAR Memory Dependence Prediction-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 an 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. This we illustrate with the example of Fig. 4, where we show how an earlier detection of an 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 by the *Dependence Detection Table (DDT)* [22]. The DDT is an address indexed cache which records the PC of a load or a store that accessed the corresponding address. The PC of a load or a store is their instruction address. After the first instance of LOAD calculates its address, it also creates a new entry in the DDT at commit time (Fig. 4a). Later, LOAD' may access the DDT using the same address (Fig. 4b) where it will locate the entry for LOAD. At this point, we have detected an 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 LOAD'. When a later instance of LOAD is encountered (Fig. 4b), 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

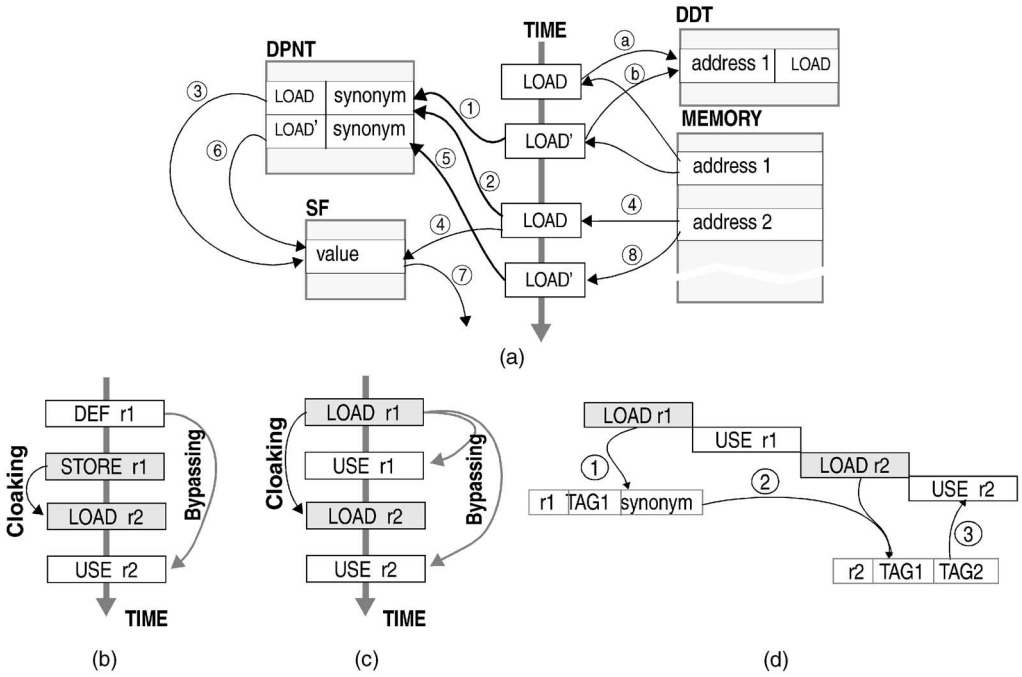

Fig. 4. (a) RAR-based cloaking. (b)-(d) RAR-based bypassing.

value is yet available. When LOAD'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 to 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 LOAD' 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 miss-speculation occurs. While we assumed that LOAD's memory access completes before LOAD' 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 an 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 small 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 had 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 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 Memory Dependence Prediction-Based Speculative Memory Bypassing

The process of RAW-based bypassing is shown in Fig. 1a. 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 ($\mathrm{DEF}\,R_X$) to the consumer ($\mathrm{USE}\,R_Y$). The goal of our RAR-based extension to bypassing is shown in Fig. 1b. We assume that a RAR dependence exists between "$\mathrm{LOAD}\,R_Y$" and "$\mathrm{LOAD}\,R_Z$." While RAR cloaking will allow "$\mathrm{LOAD}\,R_Z$" to obtain a speculative value by naming "$\mathrm{LOAD}\,R_Y$," its consumer, "$\mathrm{USE}\,R_Z$," will have to wait for "$\mathrm{LOAD}\,R_Y$" to propagate this value. With our method, "$\mathrm{USE}\,R_Z$" is speculatively linked directly to "$\mathrm{LOAD}\,R_Y$." 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 an RAR dependence, similarly to a store of a RAW dependence. The only difference is that this "producing" load cannot be eliminated. Fig. 4b 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) [14], [30]. Cloaking and bypassing are architecturally invisible. As such, we may deploy them only when justified by the underlying technological trade-offs. 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], [3], [4], [9], [15], [27], [7]. 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* [18], a special case of value prediction [17], [10]. 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 et al. introduced RAW memory dependence prediction for scheduling loads [20], [21]. Tyson and Austin [29] and Moshovos and Sohi [22] introduced RAW-based cloaking. The *memory renaming* proposal of Tyson and Austin combines cloaking with value prediction. Lipasti's *Alias prediction* [16] is also similar to cloaking. Moshovos and Sohi proposed RAW-based speculative memory bypassing [22]. Jourdan et al. proposed a method [12] where address information and prediction is used to eliminate loads and to increase coverage. Reinman et al. investigated a software-guided cloaking approach [25] and compared various load latency reduction techniques [24].

5 Evaluation

In this section, we present experimental evidence in support of the utility of the technique we propose. We use a two step approach. Initially, we investigate cloaking and bypassing, ignoring timing considerations. This allows us to study cloaking and bypassing without having to be concerned with side effects introduced by their interaction with other execution techniques. Once we have studied the potential of cloaking and bypassing, we then consider how a particular implementation performs.

The rest of this section is organized as follows: In Section 5.1, we describe our methodology. The first step in using cloaking is that of building dependence history. For this reason, 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 5.6, we present a characterization of the speculated loads. We consider their address and value locality characteristics and also measure the instruction distance between dependent loads and stores. In Section 5.7, we measure the performance impact of a combined cloaking and bypassing mechanism. We also compare it to load value prediction and study a combined mechanism.

5.1 Methodology

For our experiments, we have used the SPEC'95 programs. We compiled these programs for the MIPS-I architecture [13] using the GNU gcc compiler version 2.7.2 (flags: -O2 -funroll-loops -finline-functions). For FORTRAN codes, we used AT&T's f2c compiler to convert them to C. In order to attain reasonable simulation times we: 1) modified the standard *train* or *test* inputs and 2) used sampling for some programs [5], [23], [32]. Table 1 reports the dynamic instruction count, the fraction of loads and stores, and the sampling ratio per program. We used sampling only for the timing experiments of Section 5.7. 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 (i.e., sample size). In all sampling simulations, the observation size is 50,000 instructions. We report sampling ratios under the "SR" columns as "timing:functional" ratios. For example, a 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 the functional portion of the simulation, the following structures were simulated: I-cache, D-cache, and branch prediction. In our evaluation, we will refer to the benchmarks by using the first numbers of their name shown in Table 1.

The simulators we used are modified versions of the Multiscalar timing simulator [5]. This simulator uses event-driven simulation for both the processor core and the memory system. Our base processor is capable of executing up to eight instructions per cycle and is equipped with a 128-entry instruction window. The processor is pipelined and it takes five cycles for an instruction to be fetched, decoded, and placed into the instruction scheduler. It takes one cycle for an instruction to read its input operands from the register file once issued. Integer functional unit latencies are one cycle except for multiplication (four cycles) and division (12 cycles). Floating-point functional unit latencies are as follows: two cycles for addition/subtraction and comparison (single and double precision or SP/DP), four cycles SP multiplication, five cycles DP multiplication, 12 cycles SP division, 15 cycles DP division. A 128-entry load/store scheduler (load/store queue) capable of scheduling up to four 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* [21]. That is: 1) A load may access memory even when there are preceding stores that have yet to calculate their address, 2) a load will wait for preceding stores that are known to write to the same memory location, and 3) stores post their address even when their data is not yet available. Previous work has shown that memory dependence speculation can have a significant impact on base performance [8], [21], thus impacting the relative importance of any load value speculative technique (the results of Fig. 11 in Section 5.7.2 support this observation). We have found that, for our centralized window processor

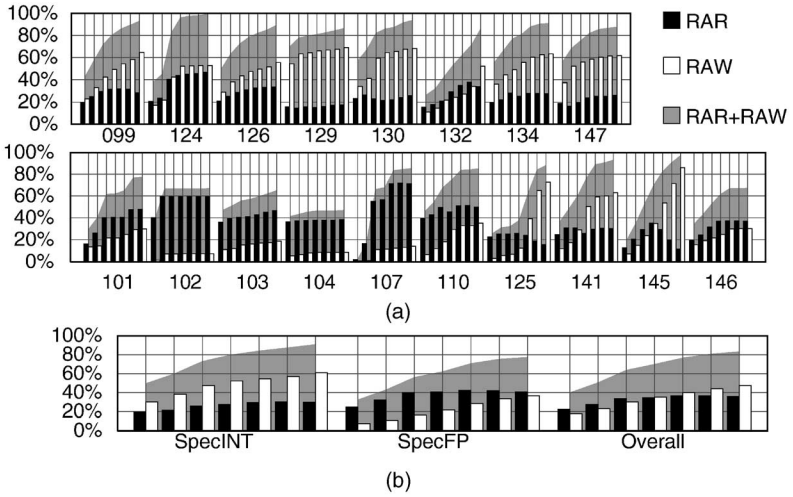

Fig. 5. Fraction of loads that have RAW or RAR dependences as a function of dependent detection table size. Range is 32 to 2K entries in steps that are powers of two. (a) Per program results. (b) Average over integer, floating-point, and all programs.

model, naive memory dependence speculation offers performance very close to that possible with ideal speculation. The base memory system is comprised of:

1. a 128-entry write buffer,
2. a nonblocking 32Kbyte/16 byte block/4-way interleaved/2-way set associative L1 data cache with two cycle hit latency,
3. a 64K/16 byte block/8-way interleaved/2-way set associative L1 instruction cache with two cycle hit latency,
4. a unified 4Mbyte/8-way set associative/128 byte block L2 cache with a 10 cycle hit latency, and
5. 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 one cycle (L2) or two 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 base and a GSHARE predictor [19]. In our experiments of Sections 5.2 through 5.7, we use various cloaking/bypassing configurations. The exact configurations are detailed as needed in each section.

As per the original cloaking and bypassing proposal, in all experiments, we used a level of indirection to represent and predict multiple RAW and RAR dependences (i.e., synonyms) per load and store. Instead of using the full merge algorithm assumed by Moshovos and Sohi [22], we used the incremental algorithm Chrysos and Emer proposed in the context of memory dependence speculation/synchronization [8]. 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:

$$\mathrm{ST}_1\ \mathrm{A}, \mathrm{LD}_1\ \mathrm{A}, \mathrm{ST}_2\ \mathrm{B}, \mathrm{LD}_2\ \mathrm{B}, \mathrm{ST}_1\ \mathrm{C}, \mathrm{LD}_2\ \mathrm{C}.$$

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 $(\mathrm{ST}_1, \mathrm{LD}_2)$ dependence is encountered, the two instructions have different synonyms already assigned to them. 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. In this case, the synonyms for both LD_1 and ST_2 will be updated. 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. In our example, if X > Y, then the synonym for ST_1 and only for ST_1 will be set to Y. This delays updating the entries for ST_2 and LD_1 until the corresponding dependences are encountered again. Because of the bias in the synonym selection, eventually all relevant instructions will be given the same synonym. We note that this algorithm resulted in accuracy that was virtually identical to that possible with full merging. Finally, we did not provide explicit support for dependences between instructions that access different data types. We did so as such dependences are rare in the SPEC95 benchmarks. This might not be the case for other programs. Potential support for such dependences is discussed in the original RAW-based cloaking and bypassing proposal [22].

5.2 Memory Dependence Detection

In this section, we measure the fraction of memory communication and data-sharing activity 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. Fig. 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 entries). Shown is the total number of loads with dependences (gray shaded area) and a breakdown in terms of the dependence type (RAW or RAR).

As shown by the averaged results (Fig. 5b), a large fraction of loads get their value via a dependence that is

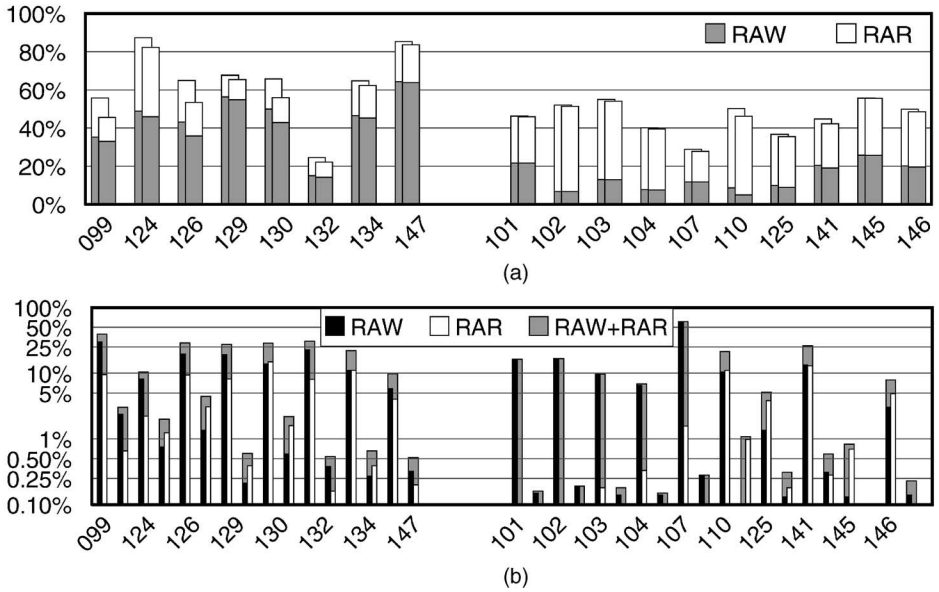

Fig. 6. Cloaking accuracy: (a) coverage and (b) misprediction rates (logarithmic Y axis). Two predictors are shown per program: 1-bit nonadaptive (left) and the 2-bit automaton described in the main text (right). Percentages are over all loads. A breakdown in terms of the dependence type is also shown.

visible even with the smaller DDTs. Overall, dependences are more frequent for the integer codes. The relative fractions of RAR and RAW dependences, and, for this reason, their potential importance, 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. 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 dependences are detected. While RAR dependence frequency also increases for DDTs of up to 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 percent and 60 percent for the integer and floating-point programs, respectively). Moreover, we have seen that the fraction of loads that have a visible RAR dependence but no visible RAW dependence is significant. For example, with the 128-entry DDT, these fractions are 25 percent (integer) and 40 percent (floating-point). 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 to and often better than that achieved with larger DDTs.

5.3 Cloaking Coverage and Miss-Speculation Rates

In this section, we measure the accuracy of two cloaking predictors. We use two metrics: *coverage* and *miss-speculation rate*, both measured as a fraction over all executed loads. We define *coverage* as the fraction of loads that get a correct value via cloaking. The complement of coverage is the *miss-speculation rate*, which is the fraction of loads that get an incorrect value. For the purposes of this study, we assume infinite DPNTs and evaluate predictors with the following two confidence mechanisms: 1) nonadaptive 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 nonadaptive predictor as it provides a rough upper bound on coverage (once a dependence is detected, it will always try to use cloaking).

Fig. 6 reports cloaking coverage (Fig. 6a) and misprediction rates (Fig. 6b). A breakdown in RAW (gray) and RAR (white) dependences is also shown. Focusing on coverage, we observe that, on the average, RAR dependences offer roughly an additional 20 percent (integer) and 30 percent (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 miss-speculation rates (Fig. 6b) show, this loss comes at the benefit of a drastic reduction in miss-speculations. Also shown is a breakdown in terms of the dependence that causes the misprediction (i.e., the source of the value). RAW dependences are shown with the black bars while RAR dependences are shown with the white bars. Their sum, being the overall miss-speculation rate, is shown in gray (note the Y scale is logarithmic in this graph). We can observe that, for the integer codes, RAR miss-speculations are frequent and, in some cases, even more frequent than RAW dependences. This suggests that RAW dependences are more regular than RAR dependences in the integer codes. For the floating-point programs, RAR dependences are either the sole source of miss-speculations or they cause as many miss-speculations as RAW dependences do.

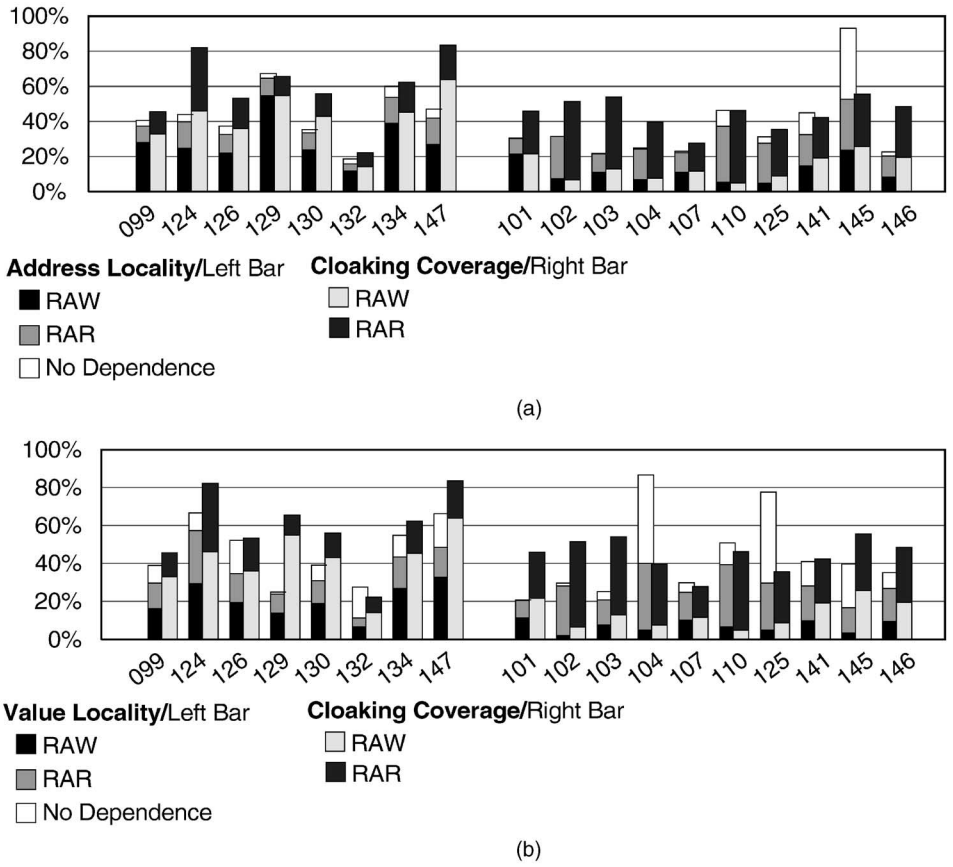

Fig. 7. (a) Address locality breakdown. (b) Value locality breakdown.

However, it should be noted that, for most floating-point programs, RAR dependences are also responsible for most of the loads that are correctly communicated. On average, the adaptive predictor reduces miss-speculations by almost an order of magnitude compared to the nonadaptive predictor. The miss-speculation rates are 2 percent, 0.35 percent, and 1.01 percent for the integer, floating, and all programs, respectively. From that, 1.1 percent, 0.17 percent, and 0.54 percent (percentage of loads) comes from RAW dependences. In the rest of the evaluation, we will focus on cloaking mechanisms that use 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 Fig. 7a. 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 break down the loads that exhibit address locality into three categories, depending on whether they have a RAW, RAR, or no dependence detected by our 128-entry DDT. We can observe 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 do not also have a dependence (145.fpppp exhibits similar behavior if a larger DDT is used).

5.5 Value Locality and Value Prediction Measurements

In this section, we measure the value locality of loads and its relation to cloaking coverage. We also compare the accuracy of a stride-based value predictor with that of our cloaking/bypassing mechanism. We do so as value prediction can also be used to allow loads to obtain their value early, possibly earlier than cloaking would allow. Fig. 7b reports the fraction of loads that exhibit value locality alongside of a breakdown of loads that get a correct value via cloaking. Following the widely used definition of value locality, this includes those dynamic loads that read the same value as they did last time they were encountered. 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 for 132.ijpeg, 104.hydro2d, 110.applu, and 125.turb3d. Moreover, cloaking covers more of the loads with dependences. This phenomenon is more pronounced for those loads that experience RAW dependences where cloaking coverage is sometimes twice the fraction of loads that exhibit value locality. However, in some cases, quite a few loads exhibit value locality, but are not predicted by cloaking. These observations suggest a potential synergy of the two techniques.

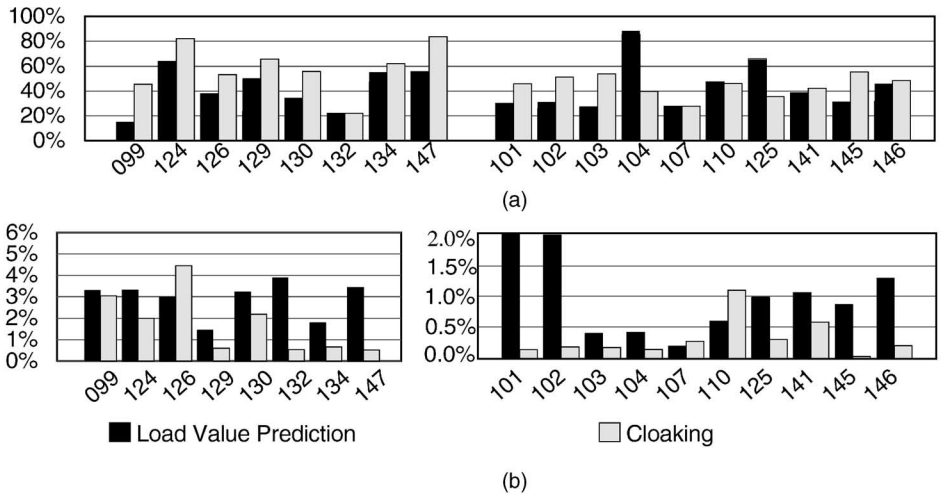

Fig. 8. Comparing a stride-based value predictor with cloaking. (a) Coverage: loads that get a correct value. (b) Miss-speculation: loads that get an incorrect value.

We next compare a value predictor and a cloaking mechanism, both utilizing finite resources. For this experiment, we simulate a fully associative stride-based predictor with 16K entries. The cloaking mechanism we use has a 16K DPNT, a 128-entry DDT, and a 2K synonym file. All structures are assumed to be fully associative. Fig. 8 reports the resulting coverage (Fig. 8a) and miss-speculation rates (Fig. 8b). The value prediction results are shown by the black left bar, while the cloaking prediction results are shown by the right gray bar. Fig. 8a shows that cloaking offers better coverage than value prediction for most programs. Similar trends are seen by miss-speculations. Cloaking experiences less miss-speculations than the value prediction for all programs except 129.compress and 146.wave5.

Of course, these results cannot be generalized; different or larger predictors may improve value prediction (and cloaking) and the performance impact of each technique can only be judged when timing is considered. Even so, these results suggest that cloaking provides a way to obtain load values early for a significant fraction of loads that do not exhibit value locality. This observation hints at a potential synergy between the two techniques. To better understand how value prediction and cloaking/bypassing relate, we measured the fraction of loads that get a correct value from cloaking/bypassing but not from value prediction and vice versa. The results are shown in Table 2. We also present a breakdown of the values obtained via cloaking/bypassing in terms of the dependence type. We can observe that, indeed, for most programs, value prediction captures some loads that cloaking/bypassing does not and vice versa. 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. A potential advantage of memory dependence prediction is that it relies on regularities in the instruction stream. This may be important when the working set of values is relatively large.

TABLE 2
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)

	Cloaking/ Bypassing	VP		Cloaking/ Bypassing	VP
099	29.18%	5.29%	***101***	25.58%	0.24%
124	24.85%	1.88%	***102***	26.41%	0.37%
126	24.04%	8.01%	***103***	33.08%	2.67%
129	42.18%	0.22%	***104***	4.31%	49.94%
130	32.17%	6.14%	***107***	2.77%	2.60%
132	13.93%	11.24%	***110***	11.46%	12.60%
134	23.29%	7.82%	***125***	2.82%	41.94%
147	32.85%	5.03%	***141***	13.34%	9.67%
			145	40.34%	18.17%
			146	22.92%	5.94%

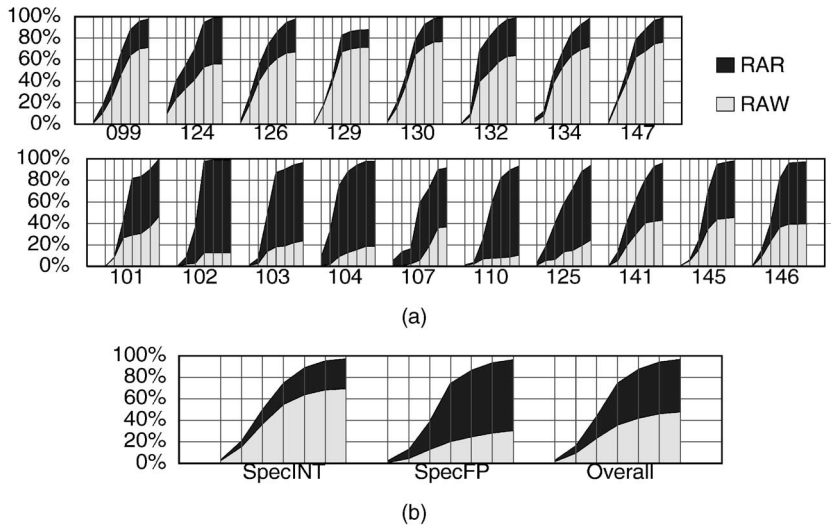

Fig. 9. Cumulative dynamic instruction distance distribution between source instruction and loads that get a correct value via cloaking. Range is four to 16K instructions. Samples are taken at the following distances: 4, 16, 64, 256, 1K, 4K, and 16K (powers of 4). (a) Per program measurements. (b) Averaged measurements.

5.6 Dynamic Instruction Distance Distribution

In this section, we measure the distance in dynamic instructions between the loads that get a correct value via cloaking and the source instruction that supplied that value. This measurement provides additional insight on the nature of load values that our mechanisms handle. Fig. 9 reports the fraction of loads that get a correct value via cloaking as a function of dynamic instruction distance. The range shown is four to 16K instructions and samples are taken at distances that are powers of four. Fig. 9a reports the per program measurements, while Fig. 9b reports averaged measurements for the integer, floating-point, and all programs.

As can be seen by the results of Fig. 9b, for 50 percent of all correctly communicated loads, the source of the value is within 64 instructions. This result provides an upper bound on the fraction of loads that could also benefit from bypassing in a 64-instruction window processor. This percentage rises to roughly 75 percent when we consider distances of up to 256 instructions. It can also be seen that about 15 percent of all correct values correspond to dependences that span more than 1K instructions. For some programs (e.g., 129.compress and 107.mgrid), about 10 percent of all correct values correspond to dependences that even exceed 16K instructions. This result suggests that even the relatively small detection table we used is capable of capturing memory communication that spans large regions of the dynamic instruction stream.

5.7 Performance Impact

In this section, we evaluate the performance impact of a combined cloaking and bypassing mechanism. We do so by simulating a dynamically scheduled ILP processor, measuring its performance with and without a cloaking/bypassing mechanism. Furthermore, we evaluate the performance impact of our RAR-based extension over the original RAW-based cloaking/bypassing. The rest of this section is organized as follows: In Section 5.7.1, we describe the cloaking/bypassing mechanism we simulated. In Section 5.7.2, we measure how performance varies when cloaking/bypassing is used for two miss-speculation handling models. We also measure the improvements obtained by augmenting cloaking/bypassing with our RAR memory dependence-based techniques.

5.7.1 Configuration

The cloaking/bypassing mechanism we used is comprised of: 1) a 128-entry fully-associative DDT with word granularity, 2) an 8K, 2-way set-associative DPNT, and, finally, 3) a 1K, 2-way set associative synonym file. Fig. 10a illustrates how the various components of the cloaking/bypassing mechanism are integrated in the processor's

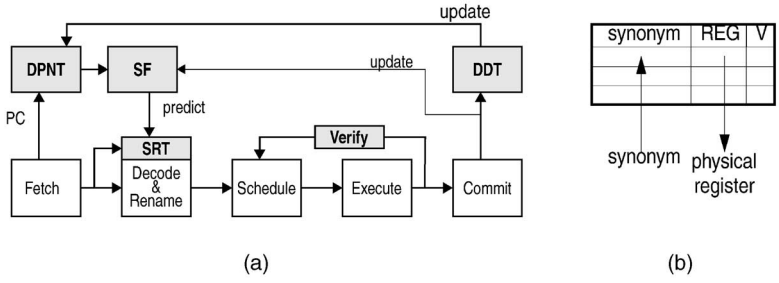

Fig. 10. (a) An out-of-order processor pipeline with a cloaking/bypassing mechanism. (b) The Synonym Rename Table (SRT).

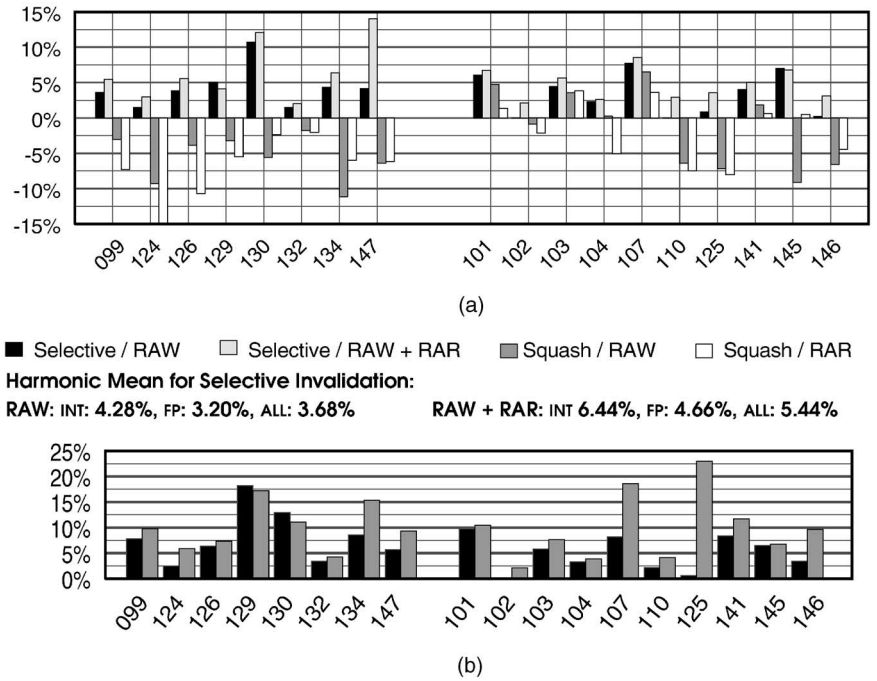

Fig. 11. (a) Performance of RAW and RAW+RAR cloaking/bypassing. Two miss-speculation mechanisms are simulated: selective invalidation and squash invalidation (see text). (b) Speedup over a processor that does not use memory dependence speculation. Left dark bar: RAW cloaking/bypassing. Right gray bar: RAW+RAR cloaking/bypassing.

pipeline. Dependence detection occurs when loads or stores commit by accessing the DDT. DPNT updates also occur at commit time. Dependence predictions are initiated as soon as instructions enter the decode stage. At most eight predictions can be made per cycle and at most eight instructions can be scheduled for cloaking or bypassing per cycle. Moreover, no data type information is used for cloaking/bypassing purposes.

For the purposes of bypassing, we introduce the *synonym rename table* (SRT). This is a synonym-indexed table that maps synonyms to physical registers. Recall that, in bypassing, we directly link consumers with the actual producer of values passed through memory, completely bypassing the intervening memory operations. As shown in Fig. 10b, an SRT entry is comprised of: 1) a valid bit, 2) a synonym identifier, and 3) a physical register identifier. The physical register identifier points to the target register of the actual producer of a synonym mapped value. In the case of RAW-bypassing, this is the source register of the store, while, in RAR-bypassing, it is the target register of the first among the RAR-dependent loads. SRT entries are allocated for loads and stores that are predicted as synonym producers. Loads that are predicted as synonym consumers inspect both the SRT and the synonym file in parallel to determine the current location of the appropriate synonym. If an entry is found in the SRT, the synonym resides in the physical register file (or in a reservation station) as the corresponding load or store has yet to commit. Otherwise, the synonym resides in the synonym file. In our simulations, we have modeled a fully associative SRT. Other organizations are possible.

The final piece of the cloaking/bypassing mechanism is responsible for: 1) verifying speculatively communicated values and 2) recovering from miss-speculations. Miss-speculations are signaled only when an instruction has actually read and used an incorrect value. For the purposes of this evaluation, we have experimented with two miss-speculation recovery mechanisms. The first is *selective invalidation* [16], [26]. This mechanism reexecutes 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 refetched from scratch. We also experimented with an oracle mechanism that does not speculate when this would result in miss-speculation. 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. Furthermore, as also reported in [28], care must be taken to avoid destructive interference with other prediction techniques, especially branch prediction. In this study, 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, in order to avoid interfering with branch prediction, we disallow control resolution on branches with value speculative inputs.

5.7.2 Performance with a Cloaking/Bypassing Mechanism

Fig. 11b shows 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 gray bar is for the extended mechanism we propose. The other two bars report performance with squash invalidation (gray for RAW-based and white for RAW+RAR-based cloaking). Using squash invalidation rarely results in performance improvements. Alternative confidence mechanisms [6], [29] could potentially improve performance under this configuration. However, such an investigation is beyond the scope of this paper. In contrast to squash invalidation, 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 the small cost associated with extending the original RAW-based cloaking/bypassing with RAR prediction. On the average, performance improvements are up to 6.44 percent (integer) and 4.66 percent (floating-point) from 4.28 percent and 3.20 percent, respectively. An anomaly is observed for two cases where our extended mechanism results in somewhat lower performance. The primary cause of this anomaly is that we use a common DDT for both RAW and RAR dependences. As a result, some RAW dependences are not detected by the combined DTT because stores get evicted by more recent loads.

In Fig. 11b, we report speedups for a configuration that does not speculate on memory dependences. In this case, loads wait for all preceding stores to calculate their address (see Section 5.1). We do so for completeness and as most studies in value speculative techniques assume such a configuration. Two bars are shown per benchmark. The left dark one is for the original RAW cloaking/bypassing, while the right gray one is for the combined RAW and RAR mechanism. It can be seen that, in most cases, speedups are significantly higher (often double) compared to Fig. 11, where the base processor uses memory dependence speculation (see Section 5.1). There are cases where the speedup is lower. In those cases, other instructions (e.g., loads not speculated) dominate performance. This experiment demonstrates that the use of memory dependence speculation greatly impacts the performance impact of our load value speculative techniques.

6 Conclusion

In this work, we have identified that typical programs exhibit highly regular RAR memory dependence streams. For most programs, more than 80 percent of all loads with an RAR dependence experienced the same RAR dependence as the last time they were executed. This property facilitates history-based RAR memory dependence prediction. We exploited this prediction to develop two memory latency reduction techniques: RAR-based cloaking and bypassing. An advantage of our techniques is that they can be implemented as small extensions to the original, RAW-dependence-based mechanism. Moreover, we studied the performance impact of the resulting mechanism.

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

Future research may focus on optimizing the naive, first-cut predictors we used in this study. Another direction stems from viewing memory as a primitive for synthesizing elaborate actions. In this work, we considered the two most primitive actions synthesized through memory: 1) inter-operation communication and 2) data sharing. Further investigation may identify other, more elaborate actions memory is used for. It may be possible to develop mechanisms to perform these actions faster. Finally, the techniques we propose might be useful in the context of explicitly parallel systems.

References

[1] T.M. Austin, D.N. Pnevmatikatos, and G.S. Sohi, "Fast Address Calculation," *Proc. 22nd Int'l Symp. Computer Architecture,* June 1995.

[2] T.M. Austin and G.S. Sohi, "Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency," *Proc. 28th Ann. Int'l Symp. Microarchitecture,* Nov. 1995.

[3] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Int'l Conf. Supercomputing,* 1991.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *Proc. 26th Int'l Symp. Computer Architecture,* May 1999.

[5] S.E. Breach, "Design and Evaluation of a Multiscalar Processor," PhD thesis, Univ. Wisconsin-Madison, Dec. 1998.

[6] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," *Proc. 26th Int'l Symp. Computer Architecture,* May 1999.

[7] B.-C. Cheng, D.A. Connors, and W.-M. Hwu, "Compiler-Directed Early Load-Address Generation," *Proc. 31st Ann. Int'l Symp. Microarchitecture,* Dec. 1998.

[8] G.Z. Chrysos and J.S. Emer, "Memory Dependence Prediction Using Store Sets," *Proc. 25th Int'l Symp. Computer Architecture,* June 1998.

[9] R.J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM J. Research and Development,* vol. 37, no. 4, July 1993.

[10] F. Gabbay and A. Medelson, "Speculative Execution Based on Value Prediction," Technical Report, TR-1080, Electrical Eng. Dept., Technion-Israel Inst. of Technology, Nov. 1996.

[11] M. Golden and T. Mudge, "Hardware Support for Hiding Cache Latency," Technical Report CSE-TR-152-93, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, Feb. 1991.
[12] 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," *Proc. 31st Ann. Int'l Symp. Microarchitecture,* Dec. 1998.
[13] G. Kane, *MIPS R2000/R3000 RISC Architecture.* Prentice Hall, 1987.
[14] S.M. Kurlander and C.N. Fischer, "Minimum Cost Interprocedural Register Allocation," *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* Jan. 1996.
[15] J. Gonzalez and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," *Proc. Int'l Conf. Supercomputing,* July 1997.
[16] M.H. Lipasti, "Value Locality and Speculative Execution," PhD thesis, Carnegie Mellon Univ., Apr. 1997.
[17] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th Ann. Int'l Symp. Microarchitecture,* Dec. 1996.
[18] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Ann. Symp. Architectural Support for Languages and Operating Systems,* Oct. 1996.
[19] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Equipment Corp., Western Research Laboratory, June 1993.
[20] A. Moshovos, "Memory Dependence Prediction," PhD thesis, Univ. of Wisconsin-Madison, Dec. 1998.
[21] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th Int'l Symp. Computer Architecture,* June 1997.
[22] A. Moshovos and G. Sohi, "Streamlining Inter-Operation Communication via Data Dependence Prediction," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* Dec. 1997.
[23] M. Reilly and J. Edmondson, "Performance Simulation of an Alpha Microprocessor," *Computer,* vol. 31, no. 5, May 1998.
[24] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," *Proc. 31st Ann. Int'l Symp. Microarchitecture,* Dec. 1998.
[25] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin, "Profile Guided Load Marking for Memory Renaming," Technical Report CS98-593, Univ. of California, San Diego, July 1998.
[26] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* Dec. 1997.
[27] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *Proc. Ann. Int'l Symp. Microarchitecture-30,* Dec. 1997.
[28] A. Sodani and G.S. Sohi, "Understanding the Differences Between Value Prediction and Instruction Reuse," *Proc. 31st Ann. Int'l Symp. Microarchitecture,* Dec. 1998.
[29] G.S. Tyson and T.M. Austin, "Improving the Accuracy and Performance of Memory Communication through Renaming," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* Dec. 1997.
[30] D.W. Wall, "Global Register Allocation at Link-Time," *Proc. Symp. Compiler Construction,* Jan. 1986.
[31] L. Widigen, E. Sowadksy, and K. McGrath, "Eliminating Operand Read Latency," *Computer Architecture News,* vol. 24, no. 5, Dec. 1996.
[32] K.M. Wilson, K. Olukotun, and M. Rosenblum, "Increasing Cache Port Efficiency for Dynamic Superscalar Processors," *Proc. 23rd Int'l Symp. Computer Architecture,* May 1996.

Andreas Moshovos received the PhD degree in computer sciences from the University of Wisconsin-Madison and the MSc and Ptyhio degrees in computer science from the University of Crete, Greece. He is an assistant professor in the Electrical and Computer Engineering Department of the University of Toronto. His research interests are in computer architecture and microarchitecture for high-performance, low-cost, and power-aware microprocessors. He is the recipient of the US National Science Foundation CAREER award. He is a member of the IEEE and the ACM.

Gurindar S. Sohi received the PhD degree in electrical and computer engineering from the University of Illinois, Urbana-Champaign. He is a professor in the Computer Sciences and the Electrical and Computer Engineering Departments at the University of Wisconsin-Madison. His research interests focus on architectural and microarchitectural techniques for high-performance microprocessors, including instruction-level parallelism, out-of-order execution with precise exceptions, nonblocking caches, speculative multithreading, and memory-dependence speculation. He is a senior member of the IEEE and a member of the ACM.