Memory Dependence Prediction in Multimedia Applications

Andreas Moshovos MOSHOVOS@ECE.NORTHWESTERN.EDU
Electrical and Computer Engineering, 2145 Sheridan Road,
Northwestern University, Evanston, IL 60208, U.S.A.

Gurindar S. Sohi SOHI@CS.WISC.EDU
Computer Sciences, 1210 Dayton Street,
University of Wisconsin-Madison, Madison, WI 53706, U.S.A.

Abstract

We identify that a set of multimedia applications exhibit highly regular read-after-read (RAR) and read-after-write (RAW) memory dependence streams. We exploit this regularity to predict both RAW and RAR memory dependences. We also study how two previously proposed memory dependence prediction-based memory latency reduction techniques perform for this multimedia workload. In the first technique, a load can obtain a value by simply naming a preceding load (or store) with which a RAR (or RAW) dependence is predicted. The second technique speculatively converts a series of LOAD_1-USE_1,...,LOAD_N-USE_N (or DEF-STORE-LOAD-USE) chains into a single LOAD_1-USE_1...USE_N (or DEF-USE) producer/consumer graph. We show that via memory dependence prediction it is possible to correctly predict 33.3% of all loads on the average. Moreover, the two memory dependence prediction based techniques result on average performance improvements of 2.6% over a highly-aggressive, out-of-order, superscalar processor. The actual range of performance improvements is 0% to 8.5%. When cache latency is increased from 2 to 3 cycles, performance improves by 3.75% on average, with the range being 0% to 16.35%.

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, in previous work we have identified that typical programs exhibit highly regular "read-after-read" (RAR) and "read-after-write" (RAW) memory dependence streams [15,16].

To exploit this regularity we have presented: (1) history-based *memory dependence prediction* for both RAR and RAW dependences, and (2) two techniques that use this prediction to reduce memory latency. In memory dependence prediction an earlier detection of a RAR or RAW dependence is used to predict the dependence the next time the same loads or stores are encountered. In *speculative memory cloaking* (cloaking for short - Figure 1, part (a)), we used this prediction to create a new name space through which loads can get speculative values. This is done by predicting a preceding load or store with which a dependence may exist. Using PC-based prediction this identification takes place early in the pipeline without actual knowledge of memory addresses. Further reduction in load latency is possible with *speculative memory bypassing* (bypassing for short - Figure 1, part (b)). In this technique, we use RAW memory dependence prediction to transform DEF-STORE-LOAD-USE chains into a direct, albeit speculative, DEF-USE ones. Bypassing can also exploit RAR memory dependence prediction. In this case, we transform 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

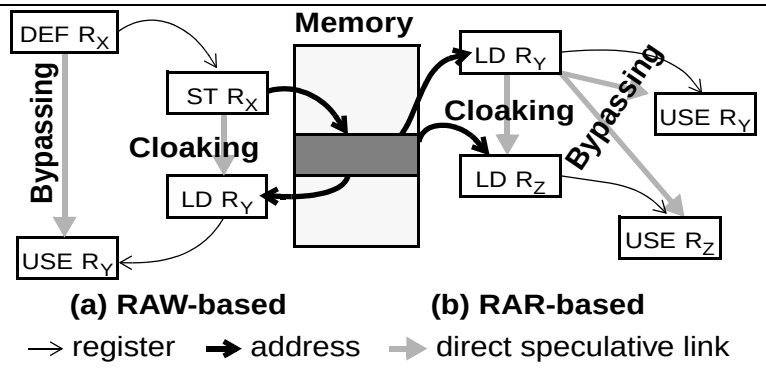

Figure 1: *Speculative Memory Cloaking and Bypassing. (a) Exploiting RAW dependences. (b) Exploiting RAR dependences.*

graph. Consequently, the first load can propagate its value to the consumers of all its RAR dependent loads.

In our previous work we have focused on applications from traditional, general purpose (SPECINT95) and numerical application domains (SPECFP95). These applications included compilers, interpreters, CPU simulators, scripting languages, a database system, and simulators of physical phenomena. With the probable exception of the physical phenomena simulators, the aforementioned applications typically exhibit relatively fairly high rates of short-distance (in time) memory dependences. However, we are currently experiencing a proliferation of multimedia oriented applications which are emerging as an equally important class of applications. Accordingly, in this work, we investigate memory dependence behavior and cloaking and bypassing on a suite of multimedia programs. Specifically, we focus on the *Mediabench* application suite [8] and on a 3D gaming application (*doom* from idSoftware). The Mediabench applications include speech, video and image coders/encoders, a postscript interpreter and a set cryptography applications.

We demonstrate that the memory dependence stream of these multimedia applications is also very regular. Moreover, we show that combined, cloaking and bypassing can offer often significant performance benefits.

The rest of this paper is organized as follows: In Section 2, we provide additional information on the benchmarks we used. In Section 3 we demonstrate that these multimedia programs exhibit regular RAW and RAR memory dependence streams. In Section 4 we comment on the rationale for cloaking and bypassing. In Section 5 we review related work. In Section 6 we evaluate the accuracy and performance cloaking and bypassing. Finally, in Section 7 we summarize our findings. For clarity we use the terms *dependence* and *memory dependence* interchangeably.

2 Benchmarks

We have used benchmarks from the Mediabench suite [8] and the 3D action game DOOM from idSoftware. All programs were compiled for the MIPS-I architecture using GNU's *gcc* compiler version 2.7.2 (flags: -O2 -funroll-loops -finline-functions). Table 2.1 reports the dynamic instruction count ("IC" column) per program, along with the fractions of loads and stores. For clarity, we use abbreviated names when referring to individual benchmarks. These names are shown under the "Ab." column. Adpcm, epic, g721 and gsm are all audio processing programs. For all of them, two versions are provided, one for encoding and one for decoding. Ghostscript, is a postscript interpreter. Jpeg is an image encoder/decoder. Mpeg2 is a video encoder/decoder. Pegwit is an encryption/decryption application. Finally, rasta is a feature extraction for speech recognition. Doom, is a 3D action game. It takes a 3D model of a virtual world along with user input and produces a first-

person view of the worlds based on the player's actions. Along with the player, several other computer-controlled players (often called *bots*) along with various other moving objects are simulated. Texture mapping is used extensively in Doom. While Doom is an interactive application, it provides a demo facility. We exploited this option during our simulations.

For most programs we have used the data inputs provided with the Mediabench suite. For jpeg encode and decode we used a different image to increase the dynamic instruction count. For ghostscript we used the BIT output device as opposed to the default PPM device. PPM outputs a '0' or an '1' character per printed pixel. As a result a significant fraction of the execution time is spent in printing these characters. The BIT produces raw binary output which is closer to what a regular printer or graphics device would require. For mpeg2 encode, we reduced the input to 3 frames in order to obtain reasonable simulation times. For DOOM, we have simulated video memory writes by allocating a non-cacheable memory array. Moreover, we have removed all sound related code as it translates to direct calls to DMA hardware which we do not currently simulate.

Program	*Ab.*	*IC*	*Loads*	*Stores*	*SR*
Mediabench					
adpcm decode	***adD***	5.4	6.8%	2.7%	N/A
adpcm encode	***adE***	6.6	6.7%	1.1%	N/A
epic decode	***epD***	6.1	15.0%	13.4%	N/A
epic encode	***epE***	53.0	12.4%	1.6%	N/A
g721 decode	***g7D***	256.9	15.0%	6.8%	1:2
g721 encode	***g7E***	271.7	14.5%	6.6%	1:2
ghostscript	***gs***	92.1	24.5%	12.1%	N/A
gsm decode	***gmD***	73.1	7.4%	4.1%	N/A
gsm encode	***gmE***	231.1	17.0%	4.5%	1:2
jpeg decode	***jpD***	33.7	18.6%	12.0%	N/A
jpeg encode	***jpE***	15.3	24.1%	8.4%	N/A
mpeg2 decode	***mgD***	170.6	23.5%	4.6%	1:1
mpeg2 encode	***mgE***	410.7	28.1%	2.8%	1:3
pegwit decode	***pwD***	19.2	20.7%	6.9%	N/A
pegwit encode	***pwE***	33.9	18.7%	6.7%	N/A
rasta	***ras***	31.7	34.3%	8.4%	N/A
Other					
doom	***dom***	343.5	29.2%	8.2%	1:2

Table 2.1: *Benchmark Execution Characteristics. Instruction counts ("IC" column) are in millions. (The "SR" column reports the timing to functional sampling ration we used during our timing experiments. See Section 6 for a detailed description.)*

3 Memory Dependence Stream Regularity Analysis

In this section, we demonstrate that the dependence stream of the multimedia programs described earlier is regular. Specifically, we show that most loads exhibit temporal locality in their RAR- and RAW-dependence streams. That is, once a load (as identified by its instruction address) experiences a RAR or RAW dependence, chances are that it will experience the same RAR or RAW dependence again soon. Moreover, we demonstrate that the working set of dependences per load is relatively small. These properties enable history-based prediction of RAR and RAW dependences. Our goal here is not to propose a specific optimization that exploits this regularity. Rather, we are interested in demonstrating that this regularity exists. In the Sections that follow we present

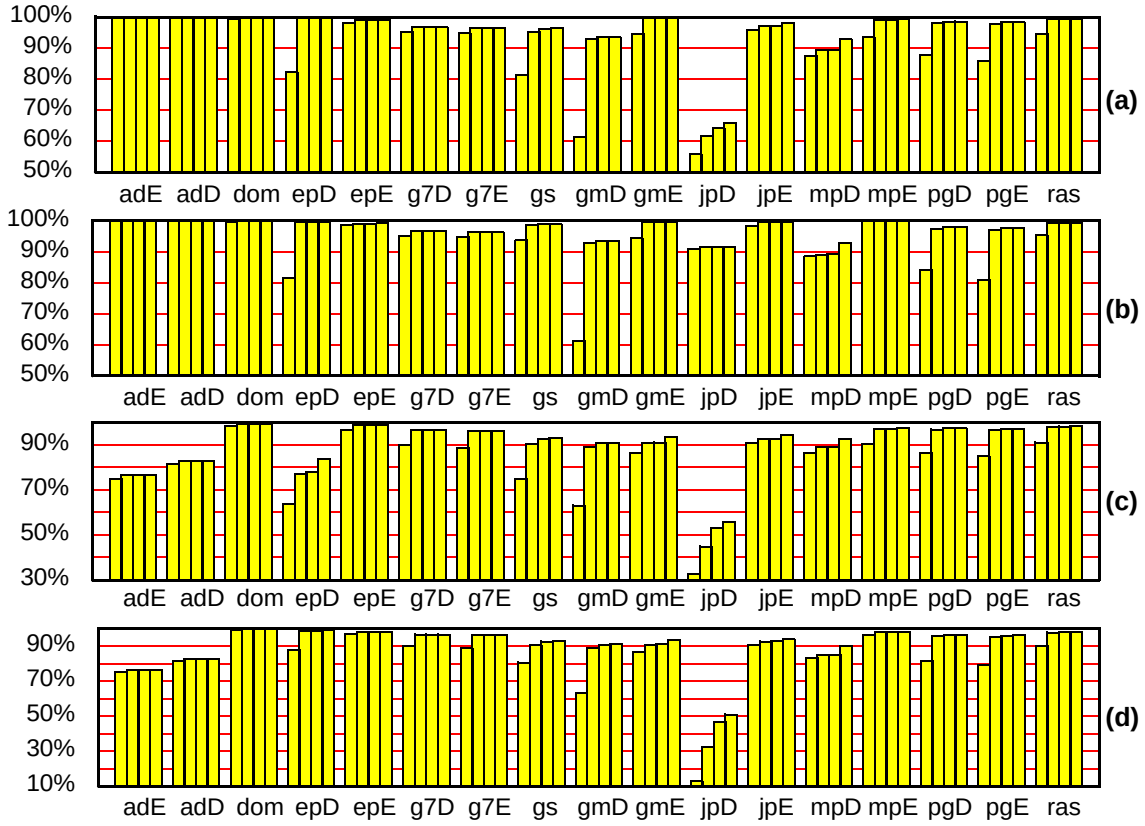

Figure 2: *Memory Dependence Locality of RAR (parts (a) and (b)) and RAW (parts (c) and (d)) dependences (range: 1 to 4). Address window size: (a)&(c) Infinite, (b)&(d) 4K entry.*

two techniques that exploit this regularity to reduce memory latency. Other ways of exploiting this regularity in multimedia program behavior may be possible.

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 4 as it allows us to keep track of a single RAR dependence per executed load (ignoring data granularity). We use a similar methodology for RAW dependences, which we represent as (STORE PC, LOAD PC) pairs. If a load has both RAW and RAR dependences (e.g., in the sequence ST, LD_1,LD_2, LD_2 has both a RAW and RAR dependence) we account for both dependences separately. Finally, in our experiments we take into account data access granularity. For example, we will keep track of all four memory dependences in the sequence ST_{BYTE}A, ST_{BYTE}A+1, ST_{BYTE}A+2, ST_{BYTE}A+3, LD_{WORD}A (each of the four stores writes a separate byte of the four-byte wide word read by the last load).

To show that dependence streams are regular we measure the *memory dependence locality* of loads with RAR dependences and that of loads with RAW dependences. We define *memory-dependence-locality-RAR(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. We define *memory-dependence-locality-RAW(n)* similarly. We should emphasize that memory-dependence-locality is a relative metric measured as a percentage of loads that experience dependences of a specific type. It is not a percentage over *all* loads.

Figure 2, part (a) shows locality results for sink loads of RAR dependences. 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 (unless they access a wider data type). 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. With the exception of gsm-decode (gmD) and jpeg-decode (jpD), more than 80% of all loads with a RAR dependence experience the same RAR dependence as last time they were executed. Locality is also strong in gsm-decode when the last 2 RAR dependences per load are considered. Figure 2, part (c) shows locality results for loads with RAW dependences. With the exception of jpeg-decode, locality is above 60% when only the last RAW dependence is considered. When the last 4 RAW dependences per load are considered locality improves to above 80%.

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 for RAR dependences or store for RAW dependences. Such a restriction is interesting from the perspective of history-based prediction as we need a mechanism to detect RAR and RAW 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), RAR dependence locality is high, in some cases higher than it was when all accesses were considered. This implies that shorter RAR-dependences seem to be more regular that distant ones. For example, RAR-dependence locality in jpeg-decode is now close to 90%. The same trend applies to RAW dependences as can be seen in part (d). The only exception is jpeg-decode where locality(1)s drops with the shorter address window (the absolute number of RAW dependences also drops). In this program, short-distance memory dependences are less regular than long-distance ones.

4 Speculative Memory Cloaking and Bypassing

We start by briefly reviewing cloaking and bypassing. Additional information can be found in [15,16,17].

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 for 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. Further reduction in communication latency is possible if we observe that 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. The effect of RAW-based cloaking and bypassing is shown in Figure 1, part (a)

Following a similar line of thinking, we can 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. As with memory communication, an explicit representation of data-sharing can eliminate the aforementioned overheads. The effect of RAR-based cloaking and bypassing is illustrated in Figure 1, part (b).

4.1 Speculative Memory Cloaking

In this section we review how we use RAW memory dependence prediction to streamline memory communication. Our method works as follows: the first time a RAW dependence is encountered, the identities of the dependent store and load 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 store to the load without having to wait for address calculation and disambiguation. We illustrate the exact process with the example of Figure 3 where we show how an earlier detection of a RAW dependence between STORE and LOAD is used the second time these instructions are encountered to provide a speculative value for LOAD. The first step is detecting the RAW 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 STORE 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 STORE. At this point we have detected a RAW dependence between the two instructions. As a result, we associate a preferably unique name, a *synonym*, with both instructions. This association is recorded in the *Dependence Prediction and Naming Table (DPNT)* (action 1). This is a PC-indexed table and two entries are created one for STORE and one for LOAD. When a later instance of STORE is encountered (part (b)), its PC is used to access the DPNT predicting whether a RAW 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 STORE receives its input data, the value to be written to memory is also written into the SF marking the entry as full (action 4). When LOAD is encountered, its PC is used to access the DPNT and predict the RAW dependence (action 5). Using the DPNT provided synonym LOAD 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 misspeculation occurs. While we assumed that STORE receives its data value before LOAD is encountered, this technique is useful even when this is not so. RAR-based cloaking is similar to RAW-based cloaking with an earlier load being treated similarly to a store [16].

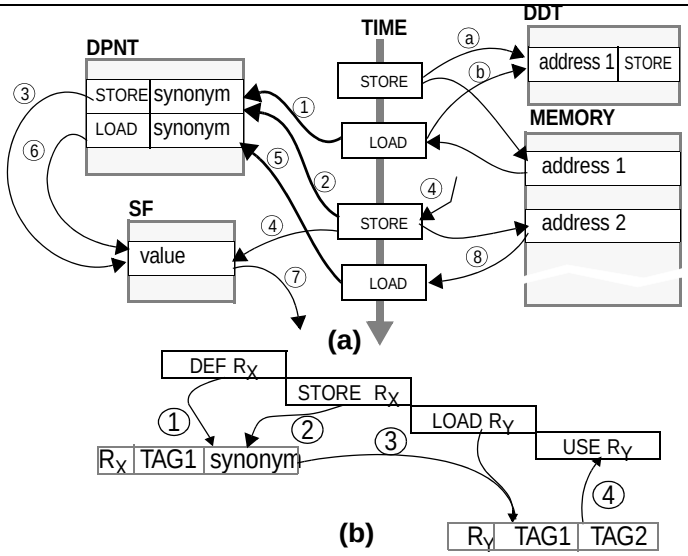

Figure 3: *RAW-based speculative memory cloaking (part (a)) and bypassing (part (b)).*

4.2 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 exact process is illustrated in Figure 3, part (b). First, and as part of register-renaming, DEF_X obtains a physical register TAG1 for its target register R_X (action 1). Then, and again as part of register-renaming, the store locates this physical register while, through cloaking also obtains a synonym (action 2). A record of both the synonym and the physical register TAG1 is kept in the *synonym renaming table* (SRT). When the load is encountered it also obtains the same synonym through cloaking. Using this synonym, the load can the access the SRT and obtain TAG1 the physical register associated with R_X. TAG1 is then associated with the physical register assigned to R_Y, the load's target register. Finally, when USE R_Y passes through register renaming it will be informed of TAG1. This will allow it to directly obtain the value produced by DEF R_X as soon as this happens. RAR-based bypassing permits a similar optimization for the consumers of RAR-dependent loads. In this case, the first load that reads a value from memory, broadcasts this value to all consumers of all RAR-dependent loads.

5 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., [1,2,4,3,5,9,19]. 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 [6,11]. 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 value sets.

Moshovos, Breach, Vijaykumar and Sohi introduced RAW memory dependence prediction for scheduling loads [14]. Tyson and Austin [21] and Moshovos and Sohi [15,17] 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 [7] 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 [18]. RAR memory dependence prediction and RAR-based cloaking and bypassing were introduced by Moshovos and Sohi [16]. All aforementioned studies of cloaking and bypassing focused on programs from the SPEC benchmark suite. To the best of our knowledge, this is the first study of cloaking and bypassing on a set of multimedia programs.

6 Experimental Evaluation

This section is organized as follows: In Section 6.1 we describe our methodology. The first step in using cloaking is building dependence history. Accordingly, in Section 6.2 we measure the fraction of memory dependences observed as a function of DDT size. In Section 6.3 we investigate an aggressive cloaking mechanism and study its accuracy. In Sections 6.4 through 6.6 we present a characterization of the speculated loads by considering their address distribution and locality, and value locality characteristics. In Section 6.7, we measure the performance impact of a combined cloaking and bypassing mechanism.

6.1 Simulation Methodology

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.

To attain reasonable simulation times we used sampling for some programs. Table 2.1 reports the sampling ratio per program. *We used sampling only for the timing experiments of section 6.7.* 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.

Finally, we did not provide explicit support for dependences between instructions that access different data types. The original RAW-based cloaking and bypassing proposal discusses potential support for such dependences [15]. However, an investigation of the utility of such support is beyond the scope of this paper.

6.2 Dependence Detection Mechanisms

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 4 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 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).

While some programs exhibit relatively high fractions of memory dependences, others do not. For example, in mpD about 80% of all loads have a RAW of RAR dependences detected with a 256-entry DDT. On the other hand, in mpE only about 30% of loads have dependences detected even when a 2K DDT is used. This behavior is quite different than that of SPEC95 programs where in virtually all cases, at least 80% of all loads had dependences detected with a 2K DDT [16]. Nevertheless, for most programs studied the faction of loads with dependences detected is significant.

No clear trend is observed with respect to the relatively contribution of RAW and RAR dependences. For some programs, RAR dependences account for virtually all dependences detected (e.g., adE and adE). For others, RAW dependences clearly dominate (e.g., g7D and g7E). In general, using larger DDTs results in more dependences being detected. However, for the majority of programs, most of the dependences detected are visible even with the smaller DDTs (e.g., 128 entries). However, some programs exhibit a steady increase, almost linear, with DDT size (e.g., pgD and jpD).

The frequency of RAW dependences never decreases with larger DDTs. However, in some cases the frequency of RAR dependences decreases when a larger DDT is used. For example, in epE we observe about 60% loads having RAR dependences with the 32-entry DDT whereas only 20% of loads have a RAR dependence detected with a 64-entry DDT. This is caused by an increased frequency of RAW dependences: 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 significant fraction of loads. The actual range varies significantly from as little as 20% (e.g., mpE) to as high as 90% (e.g., ras). For the rest of the evaluation we focus on configurations that use a 128-entry DDT. We do so, since for the majority of the programs studied such a

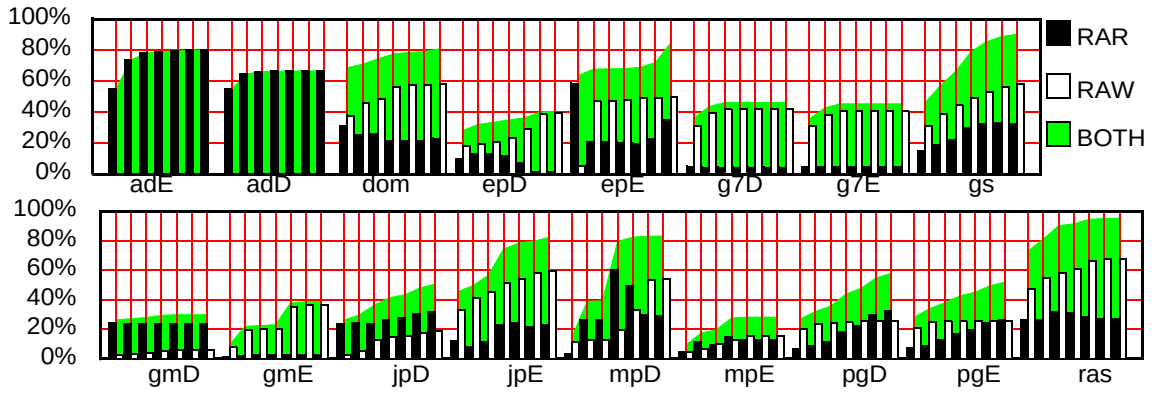

Figure 4: *Fraction of loads with RAW or RAR dependences as a function of DDT size. Range is 32 to 2K in power of 2 steps.*

table detects most of the dependences that are detectable with the larger DDT we studied (i.e., 2K-entry). On average, an 128-entry DDT detects dependences for 49.7% of all loads.

6.3 Cloaking Accuracy

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 5 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. For most programs, the majority of loads with dependences detected get a correctly predicted value from cloaking. Two notable exceptions are adE and adD where only about 30% and 20% of loads are correctly predicted, even though, 80% and 60% of loads have dependences detected (see Figure 4). This behavior can be attributed to different instances of the same static dependences whose lifetimes overlap [17]. For example, this would be the case in a loop containing a "*a[i] = a[i - 2]*" statement. In this case, the load to "*a[i - 2]*" from iteration *i* has a RAW dependence with the store to "a[i]" from iteration *i - 2*. Our cloaking predictor will fail to correctly predict the dependence as another instance of the store instruction (that of iteration *i-1*) appears in between the dependent instructions. We have observed a similar phenomenon for the SPEC95 programs [17].

The relative importance of RAW and RAR dependences varies per program. For example, in adE and adD, all correctly predicted loads have a RAR dependences, while in gmE the vast majority of correctly predicted loads have a RAW dependence. With the exception of adE and adD, only relatively 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. On average the adaptive predictor reduces misspeculations by almost an order of magnitude compared to the non-adaptive predictor. For most programs misspeculation rates with

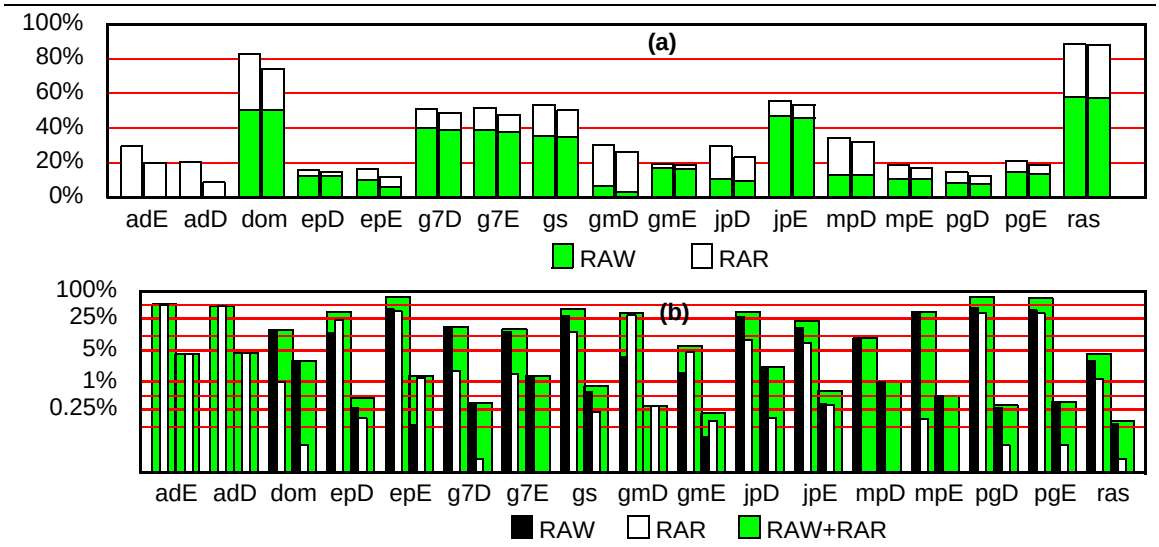

Figure 5: *Breakdown of cloaking accuracy per dependence type: (a) coverage, and (b) misprediction rates (logarithmic Y axis). Two predictors are shown per program (see text). Percentages are over all loads.*

the adaptive predictor are on or below 1%. In the worst case observed (adD), about 4% of all loads get an incorrect value from cloaking. Unfortunately, this a relatively high misspeculation rate if one considers that about 10% of loads get a correct value from cloaking for the same program. As we will show in Section 6.7, no performance benefits are observed in this case (however, no performance degradation is observed either). In the rest of the evaluation we restrict our attention to the adaptive predictor. On the average, 33.3% of all loads get a correct value with this predictor.

6.4 Address Space Breakdown

In this section we present a breakdown of the memory traffic handled by cloaking in terms of the address space eventually accessed by the corresponding loads. Figure 6 shows a breakdown of loads that get a value from cloaking in terms of the address segment that the load accesses. We split loads to those the access the data, heap and stack segments. Two bars are shown per benchmark. The one on the left is for correctly predicted loads, while the one of the right is for misspeculated loads. It can be seen that while a large fraction of the correctly predicted values belongs to the data and stack segments, many heap values are also correctly handled by cloaking. In fact, compared to the SPEC95 programs, heap accesses account for a significantly larger fraction of correctly predicted values. In some programs, many of the misspeculations are from accesses to the stack. Usually, this is caused by recursive functions that give rise to dependences whose lifetimes overlap. As we explained in the previous section, the specific cloaking predictor cannot handle such dependences.

6.5 Address Locality

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. The left bar represents the fraction of all loads that exhibit address 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. For some pro-

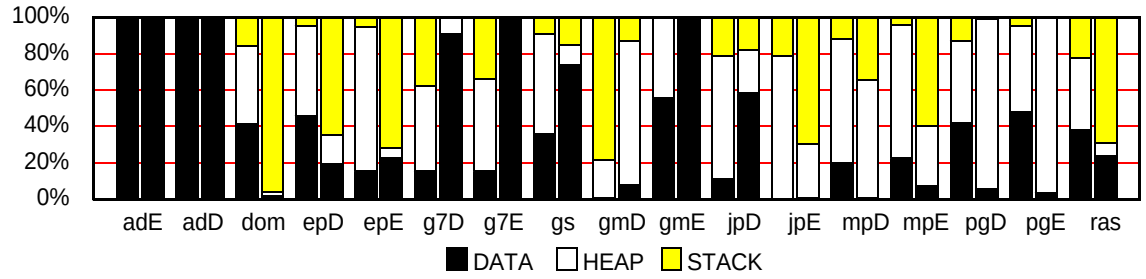

Figure 6: *Address space breakdown. Left bar: Correctly communicated values. Right bar: Misspeculated values.*

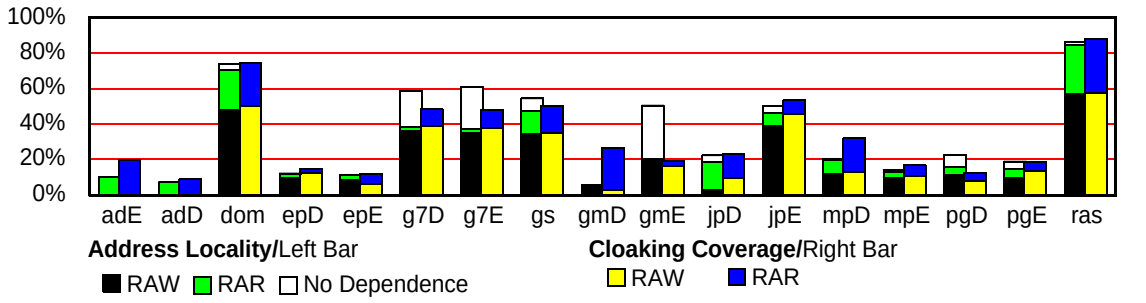

Figure 7: *Address Locality breakdown.*

grams, cloaking correctly predicts a significant fraction of loads that do not exhibit address locality (e.g., gmD or mpD). For others, significant fractions of loads exhibit address locality but are incorrectly predicted by cloaking (e.g., g7D and gmE). Nevertheless, in general address locality and cloaking coverage appear to be correlated for most programs. In some programs, and when we take the actual presence of memory dependences into account, we can observe relatively noticeable fractions of loads that are correctly predicted by cloaking and that do not exhibit address locality. For example, in g7D address locality and cloaking coverage are roughly 60% and 50% respectively. However, only 40% of all loads with address locality have a dependence detected. Other programs were similar phenomena are observed include pgE, jpD, g7E, and jpE. We should note that locality provides just an indication of the accuracy of an address-based predictor. For example, in the next section we will see that an actual value predictor does not correctly predict all loads that exhibit value locality. While we have seen that there is high-correlation between address-locality and memory-dependence-locality in these multimedia applications, the correlation was not as strong in the SPEC95 programs [17].

6.6 Value Locality and Prediction

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 load values, possibly earlier than cloaking would allow. Figure 8 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. Again, there is no common trend. In some programs, value locality is stronger than cloaking coverage (e.g., epD) while in others the opposite is true (e.g., ras). However, for most programs, cloaking correctly predicts more of the loads with dependences. For example, in jpE cloaking coverage and value locality are both about 50%. However, only 30% of all loads have a dependence and exhibit value locality.

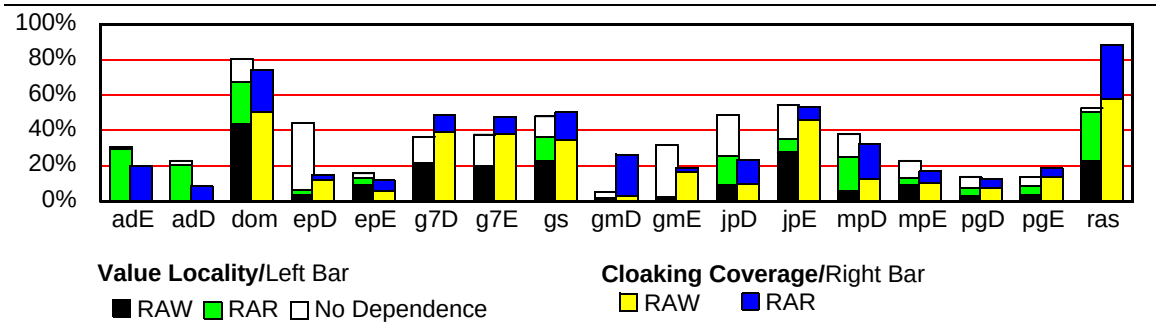

Figure 8: *Value Locality breakdown.*

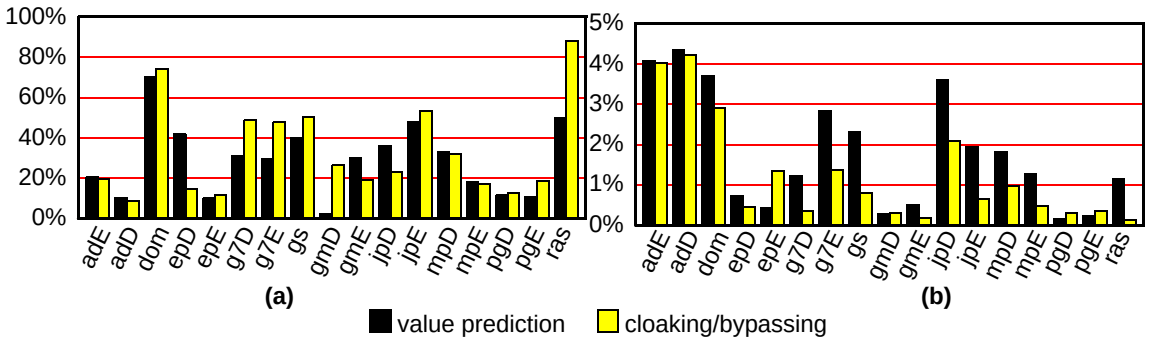

Figure 9: *Comparing a value predictor and a cloaking/bypassing predictor: (a) Coverage. (b) Misspeculation rates.*

To better understand how value prediction and cloaking/bypassing relate, we next compare an actual value predictor with cloaking/bypassing. 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. Figure 9 reports the results. Part (a) reports coverage, while part (b) reports misspeculation rates. Again, no prediction method outperforms the other one for all cases. However, we can observe that not all loads with value locality are correctly predicted. For example, while value locality was somewhat higher in doom, with the actual predictor cloaking is slightly better. As seen in part (b), for most programs, the mispeculation rate for cloaking is somewhat lower than that of value prediction. This is true even for programs where cloaking coverage greatly exceeds that of value prediction (e.g., ras). However, there are cases where cloaking mispeculation rates are significantly greater than that of value prediction even though cloaking coverage is not significantly greater (e.g., epE),

Aggregate metrics such as coverage do not provide sufficient insight on the actual mix of loads that are correctly predicted by cloaking and value prediction. While either technique may correctly predict, say 50% of all loads, the actual loads predicted may be totally different for each technique. For this reason, 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 6.1. We also present a breakdown of the values obtained via cloaking/bypassing in terms of the dependence type. As it can be seen, for most programs, there is a significant number of loads that are predicted by only one of the two. For example, in epD, 10% of correctly predicted loads by cloaking are not correctly predicted (could be not predicted at all) by value prediction. The fraction of loads correctly predicted by value prediction for the same program is 37%. Again, there is no clear trend on which

technique covers more loads. These observations suggest a potential synergy of the two techniques. While context-based value predictors could be used to improve value prediction coverage, cloaking offers a concise way of representing information for prediction purposes for a large fraction of loads.

	Cloaking/Bypassing			VP
	RAW	RAR	Total	
adD	*0.06*	*0.04*	0.11	0.65
dom	*0.05*	*0.04*	0.09	1.58
adE	*6.67*	*0.62*	7.29	3.21
epD	*8.97*	*1.91*	10.87	37.77
epE	*3.57*	*1.01*	4.58	2.34
g7D	*21.17*	*1.05*	22.23	4.38
g7E	*21.21*	*1.06*	22.26	3.93
gs	*18.09*	*1.04*	19.11	8.85
gmD	*1.42*	*23.35*	24.77	0.78
gmE	*15.25*	*1.66*	16.91	28.24
jpD	*1.89*	*1.48*	3.37	16.14
jpE	*16.40*	*0.27*	16.68	11.42
mgD	*7.80*	*0.55*	8.35	9.11
mgE	*1.84*	*2.00*	3.83	4.87
pwD	*5.52*	*1.46*	6.99	5.91
pwE	*11.18*	*1.06*	12.25	4.16
ras	*37.12*	*2.52*	39.64	1.12

Table 6.1: *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).*

6.7 Execution Time Measurements

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 6.7.1 we describe the cloaking/bypassing mechanism we simulated. In Section 6.7.2 we measure how performance varies when cloaking/bypassing is used.

6.7.1 Integrating Cloaking/Bypassing into the Pipeline

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 10 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.

Misspeculations are signalled only when an instruction has actually read an incorrectly speculated value. *Selective invalidation* is used to recover from misspeculations. This mechanism re-

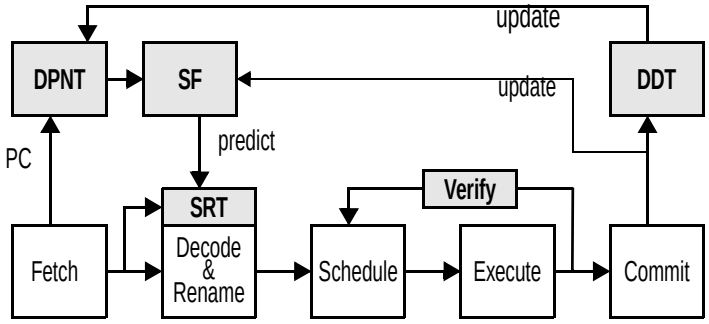

Figure 10: *Integrating cloaking/bypassing into a pipeline.*

executes only those instructions that used incorrect data. We have found that *squash invalidation* (i.e., invalidating all instructions starting from the one that was misspeculated) typically results in performance degradation even with relatively low misspeculation rates.

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 [20]. 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.

6.7.2 Performance

Figure 11, part (a) shows performance improvements when cloaking/bypassing is used. Speedup are measured over the base processor that uses no cloaking/bypassing. Two bars are shown. The one on the left is with a total cache access latency of 2-cycles, while the one on the right is for 3-cycle cache latency. For some programs virtually no performance improvements are observed. Speedups of 5% or more are observed only for doom, gs, jpE and ras. Performance improvements are generally higher as cache latency increases from 2 cycles to 3 cycles. However, in some cases this is not true, as for example, in mpD. This can be attributed mostly to loads with RAR dependences. Upon receiving a response from the cache, the base processor will search for any outstanding loads to the same address in the load/store queue and service them without issuing additional cache accesses. When cache latency is increased, subsequent loads may end up calculating their address before the first load gets its value from the cache. On the average, performance improves by 2.6% and 3.75% for the 2-cycle and 3-cycle cache systems respectively.

Figure 11, part (b) reports the fraction of all instructions that are executed speculatively as the result of cloaking/bypassing. The loads that trigger cloaking/bypassing are not included in this metric. For most programs, a relatively large fraction of instructions are executed speculatively. However, there is not always a direct correlation with performance improvements. For example, even though roughly 40% of speculative instructions are executed in gmD, no noticeable performance improvements result. Ultimately, performance improves only when the speculated instructions are part of the critical path.

In Figure 12 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. For this experiment we restrict our attention to the 2-cycle cache. It can be seen that in most cases speedups are significantly higher (sometimes double) compared to Figure 11 where the base processor uses memory

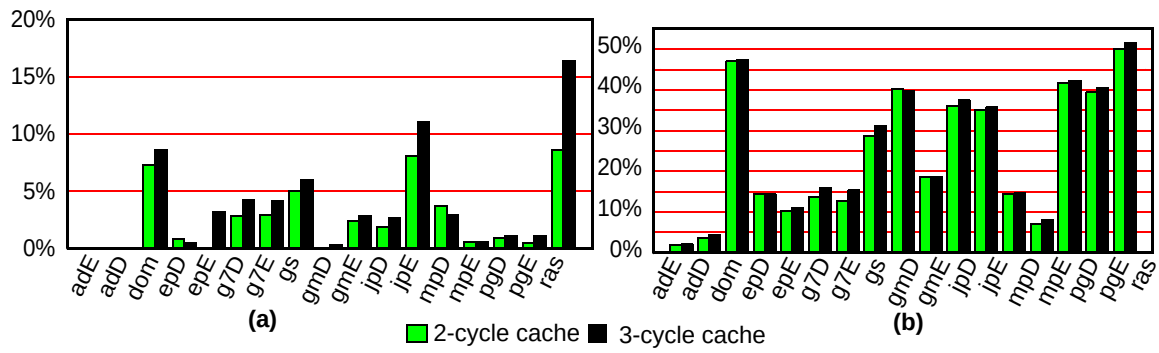

Figure 11: *(a) Performance of cloaking/bypassing for two different cache access latencies. (b) Fraction of speculatively executed instructions due to cloaking/bypassing.*

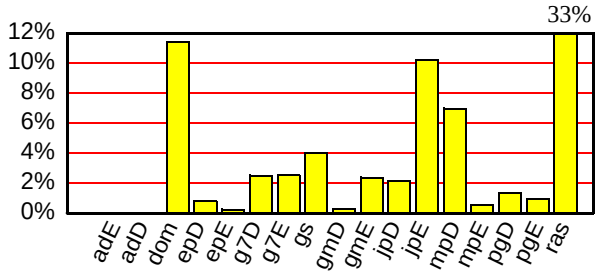

Figure 12: *Performance of cloaking/bypassing when no memory dependence speculation is used.*

dependence speculation (see Section 6.1). 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.

7 Conclusion

We have studied the memory dependence behavior of a set of multimedia applications. We have also studied the utility of cloaking/bypassing for this set of applications. We have shown that both the RAR and RAW memory dependence stream of these applications is highly regular. This observation suggested that history-based prediction of memory dependences is possible. However, we have observed that some of these applications do not exhibit relatively high rates of short distance memory dependences.

We have found that for most of these programs, memory dependence prediction can be used to support cloaking/bypassing. We have found that a relatively simple predictor can be used to predict most values for significant fractions of loads with dependences. The actual fractions varied from as little as 10% of all loads to as much as 90%. We have studied the address space distribution of these values and found that while data and stack accesses account for a significant portion, many heap accesses are also correctly predicted. We have found that, for the most part. there is some correlation between address locality and cloaking coverage. We also found that cloaking correctly predicts a significant fraction of loads that a load value predictor doesn't and vice versa. We measured the performance impact of a cloaking/bypassing mechanism. For most programs, performance improvements were relatively small. On the average, performance improved by 3.75% for a system with a 2-cycle cache access latency. However, for four of the programs studied performance improved significantly and as much as 16%. We also showed that the performance benefits of cloaking/bypassing increase when cache latency is also increased.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853, by an equipment donation from Intel Corporation and by a research grant from Northwestern University. We also thank the anonymous reviewers for their helpful comments.

References

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

[3] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *International Symposium on Computer Architecture*, 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*, Dec. 1998.

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

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

[7] 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*, Dec. 1998.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[9] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Proc.International Conference on Supercomputing*, 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. Annual International Symposium on Microarchitecture*, Dec. 1996.

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

[13] A. Moshovos and G. S. Sohi. *Memory Dependence Speculation Tradeoffs in Centralized, Dynamically-Scheduled Superscalar Processors*. In *Proc. International Conference in High Performance Computer Architecture,* Jan. 2000.

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

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

[16] A. Moshovos and G. S. Sohi. Read-after-read memory dependence prediction. In *Proc. Annual International Symposium on Microarchitecture*, Nov. 1999.

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

[18] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying Load and Store Instrictions for Memory Renaming. In *Proc. International Conference on Supercomputing, June 1999*.

[19] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proc. Annual International*

Symposium on Microarchitecture, Dec. 1997.

[20] 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.

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