Speculative Memory Cloaking and Bypassing

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

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

Abstract

We revisit memory hierarchy design viewing memory as an inter-operation communication mechanism. We show how dynamically collected information about inter-operation memory communication can be used to improve memory latency. We propose two techniques: (1) Speculative Memory Cloaking, and (2) Speculative Memory Bypassing. In the first technique, we use memory dependence prediction to speculatively identify dependent loads and stores early in the pipeline. These instructions may then communicate prior to address calculation and disambiguation via a fast communication mechanism. In the second technique, we use memory dependence prediction to speculatively transform DEF-store-load-USE *dependence chains within the instruction window into* DEF-USE *ones. As a result, dependent stores and loads are taken off the communication path resulting in further reduction in communication latency.*

Experimental analysis shows that our methods, on the average, correctly handle 40% (integer) and 19% (floating point) of all memory loads. Moreover, our techniques result in performance improvements of 4.28% (integer) and 3.20% (floating point) over a highly aggressive, dynamically scheduled processor implementing naive memory dependence speculation. We also study the value and address locality characteristics of the values our methods correctly handle. We demonstrate that our methods are orthogonal to both address and value prediction.

1 Introduction

Programs execute operations which produce values for other operations; these values must be stored while they are waiting to be consumed by the later operations. This inter-operation communication is commonly implemented by providing register and memory name spaces coupled with an agreed upon communication convention: the producer binds its value to a name within the name space, and the consumer(s) access the value by using the same name. Faster processing requires faster inter-operation communication.

In this paper we are concerned with inter-operation communication carried out through the memory name space, or simply *memory communication*. Caches have been used extensively to implement more efficient memory communication. Caches perform *memory name presence speculation*: a given memory name could reside in a variety of storage structures that are typically either fast but small or slow but large. A processor implicitly speculates that a desired name will be present in faster storage (cache), and attempts to access it from there, going to slower storage only if speculation fails. To verify the speculation, the desired memory name and the memory names stored in the given storage structure are compared; speculation succeeds only if a match occurs.

In this paper we revisit memory communication by observing that the traditional, implicit form of memory communication where the store does not directly know the identity of the consuming load(s) and vice versa, is not the only possible form. Explicit forms in which the stores and loads are linked to one another are not only possible but may lead to new forms of speculation, to new memory hierarchy components, and hopefully, to new ways of thinking about such hierarchies. In this work we focus on two such methods: *Speculative Memory Cloaking* (or simply *cloaking*) and *Speculative Memory Bypassing*. Both are memory latency reduction techniques.

The effect of our techniques is illustrated in Figure1. In speculative memory cloaking we dynamically convert implicitly specified memory communication into an explicit, albeit speculative form. To do so we use history-based *memory dependence prediction* to explicitly link loads and stores. These loads and stores can then communicate via a dynamically created name space without incurring the overhead of address calculation, disambiguation and data cache access. Speculative memory bypassing further reduces latency when the dependent load and store co-exist in the instruction window. This technique converts DEF-store-load-USE dependence chains into DEF-USE ones. As a result, values can then flow directly from the actual producer (DEF) to the actual consumer (USE). Both techniques are speculative and any communication performed in this manner has to be eventually verified. However, when speculation is successful performance may improve as memory *appears* faster to the consumers of a speculated load.

The rest of this paper is organized as follows: We start our discussion of the problem and approach by looking at inter-operation memory communication in more detail in Section 2. Here we describe the rationale for our proposed approach. We continue with a brief quantitative assessment of inter-operation memory com-

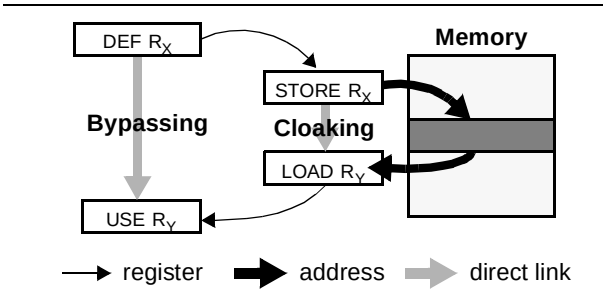

Figure 1. *Speculative Memory Cloaking and Bypassing*

munication in Section 3. We use the quantitative data, along with our rationale, to describe the requirements for cloaking in Section 4. In Section 5 we describe speculative memory bypassing. We provide a quantitative assessment of both techniques in Section 6. Finally, we comment on related work in Section 7 before we offer concluding remarks in Section 8. For clarity we use the terms *cloaking* and *bypassing* to refer to speculative memory cloaking and to speculative memory bypassing respectively.

2 Memory as an Inter-operation Communication Agent

Much of research in improving memory performance has focused on exploiting properties of the address stream. This is natural as memory appears as a storage mechanism accessed via addresses. In this work we take a different, yet orthogonal approach to improving memory performance. We observe that the memory interface is really a primitive which is often used to synthesize inter-operation communication. Separating specification from implementation, we observe that while we have chosen to specify memory communication via an address-based interface we do not have to perform it in the exact same way. In this section we discuss how the traditional specification impacts memory communication and advocate that a dynamically created explicit specification of memory communication could be used to further improve memory performance.

Memory communication is currently expressed implicitly. The load, the store or the address used provide no *a priori* indication of the communication that has to happen. As a result, detecting communication requires significant effort. To detect the communication and establish the communication link both the store and the load have to calculate an address and go through disambiguation. The latter action entails comparing the addresses of stores and loads taking program order into account; a store and a load communicate if (1) they access the same address, and (2) no intervening store accesses the same address. Both address calculation and disambiguation introduce overheads as the value may be available long before either action completes. An example is shown in Figure 2 where communication is to take place between the STORE and LOAD instructions of the code fragment of part (a). Part (b) shows a possible event sequence. Initially the store is fetched, its address is calculated, and at some later point the store's data becomes available. Later on, the load is encountered. At this point both communicating instructions have been encountered and the value is available. Yet, communication is delayed until LOAD has calculated its address and has passed through disambiguation. This is necessary to establish the dependence with STORE. Depending on whether memory dependence speculation is used, accessing the value may be delayed even further until it is established that no intervening store writes to the same address. For example, LOAD may get delayed until $STORE_1$ also calculates its address and goes through disambiguation.

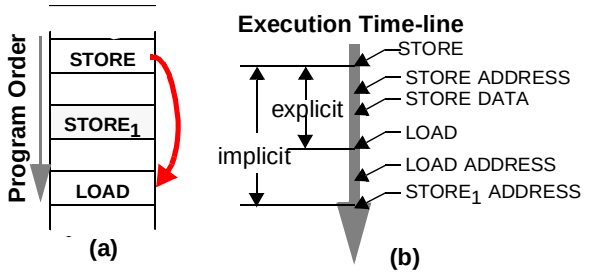

Figure 2. *An example of inter-operation memory communication. (a) Program segment with a store and a load that will communicate. (b) Time-line of execution. With an implicit specification, communication cam take place after address-calculation and disambiguation. With an explicit specification communication can take place as soon as the two instructions are encountered and the value is available.*

Even when the value becomes available after both the load and the store complete address-calculation and disambiguation, we will still observe the latency associated with accessing the value through the memory hierarchy, i.e., through the store buffer or the data caches. Unfortunately, current memory hierarchies cannot distinguish between memory communication and other memory accesses. As such, they have to be large enough to service as many accesses as possible.

The aforementioned overheads can be eliminated if we opt for an explicit representation of memory communication. In an explicit representation the producing store and the consuming load both know that communication will take place and can locate each other directly. As a result, communication can take place as soon as the two instructions are encountered and the value becomes available. Since both instructions can locate each other, there is no need for address-calculation and disambiguation. Moreover, as the results of Section 3 suggest, a rela-

tively small storage structures can be used to service a large fraction of memory communication activity.

Inter-operation communication gives rise to true (RAW) dependences. An explicit representation of memory communication requires a representation of the corresponding dependences. We could attempt to determine and specify these memory dependences statically as, for example, was done in dataflow machines [7, 27]. Even though this is an interesting option, we will not consider it further for two reasons. First, a static representation would involve changing the program representation. This would create legacy for future processor implementations and provide no benefit for legacy software. Second, identifying inter-operation communication statically may not be possible either because the dependences cannot be determined (i.e., they are ambiguous) or because they are transient (i.e., do not occur every time). For these reasons, we opt for a dynamic approach in which the conversion is done while the program is running using architecturally invisible structures. While our approach entails higher hardware costs than a pure software approach, it can be used at will and only when justified by technological trade-offs. This avoids software incompatibilities and legacy issues.

In our approach we utilize *memory dependence prediction* to explicitly express dependences dynamically as follows: we use dynamically collected dependence history information to predict future dependences. We then use these speculative dependences to create a dynamic name space through which the dependent loads and stores can communicate without incurring the overhead of address-based communication.

3 Memory Traffic Analysis

Before we delve into describing our methods it is best if we consider their potential coverage. To do so we present an empirical study of memory communication traffic using the SPEC95 benchmarks on a MIPS-I like instruction set architecture (the benchmarks, architecture and methodology are detailed in Section 6).

To get an estimate (i) of the fraction of the memory operations we can serve with a dependence-based mechanism, and (ii) of how much the storage might we require for this speculative explicit communication in this section measure the percentage of loads that read a value created by a preceding store (true dependences). We measure this characteristic as a function of the *unique store address distance* or simply *store distance.* This we define as the number of unique addresses stored to between the dependent instructions in the dynamic instruction stream. This metric provides an upper bound on the number of accesses that have to be recorded in order to detect the particular dependence.

Part (a) of Figure 3 reports the cummulative distribution of dynamic loads as a function of store distance. Percentages are measured over all dynamic loads. The measured distance range is 32 (leftmost) to 2K (rightmost) in power of two steps. Benchmarks are identified using the first three numbers of their name (see Table 1 in Section 6.1). Part (b) reports averaged results. From the average results it can be seen that 50% (integer) and 20% (floating-point) of dynamic loads get a value through a dependence at a store distance less than 128. On a per program basis, we can observe that most integer program exhibit high volumes of short store distance memory communication. This is not so for most floating programs. These programs are dominated by long running loops with little intra- and inter-iteration memory communication (these loops write large arrays using other arrays as inputs).

These results suggest that detecting dependences even over short distances (e.g., 128) has the potential to service a large fraction (50%) of all dynamic loads for integer codes and to a lesser extent for floating-point codes (20%). Motivated by the large fraction of loads that get their value through a dependence with a recent store, in Sections 4 and 5, we propose techniques that attempt to reduce the latency of this communication by explicitly linking the dependent instructions.

4 Speculative Memory Cloaking

Cloaking aims at streamlining memory communication by dynamically converting the implicit specification of communication into an explicit form. In cloaking memory dependence prediction is used to identify dependent loads and stores with high probability. Once a dependence is deemed predictable, the dependent load and store are explicitly linked via a new name, a *synonym* which uniquely identifies the dependence. For example, the synonym can be a (load PC, store PC) pair. One may wonder how using a different name may help in streamlining the actual communication. After all, data addresses and synonyms are just names that the dependent instructions use to link to each other. The answer lies (1) in the nature of the association between the name and the instructions that use it, and (2) in the information associated with the existence of the name itself. In contrast to an address, the synonym is intended to uniquely identify the dependent instruction pair. This allows the load and the store to derive the synonym based solely on their identity (PC). This in turn allows them to locate the appropriate value without having first to perform an address calculation and go through disambiguation. Furthermore, the mere association of a synonym with a load or a store is intended to indicate that the instruction is involved in short distance inter-operation communica-

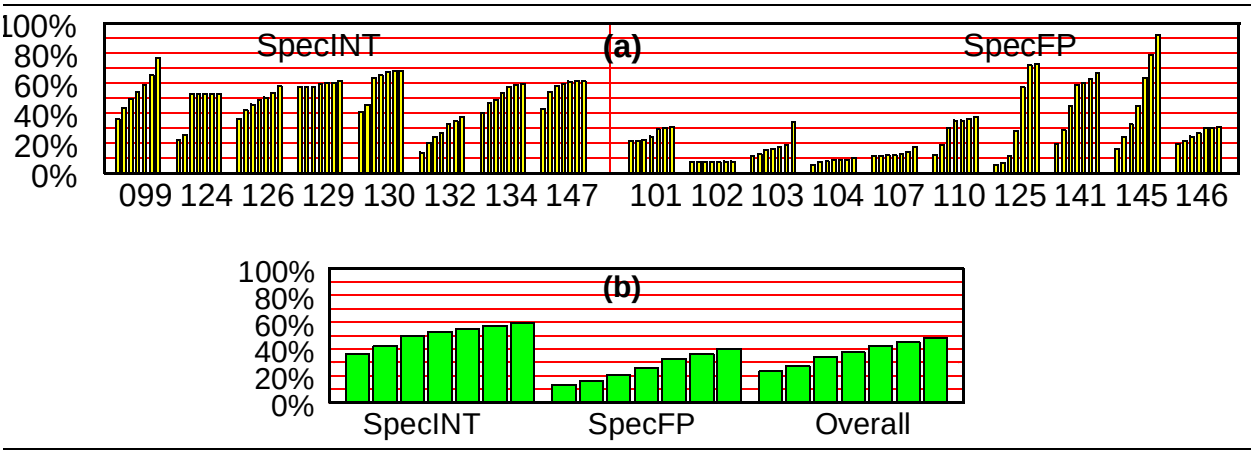

Figure 3. *Distribution of dynamic store-to-load dependence distances. (a) Per program results. (c) Averaged results. Range shown in 32 to 2K in power of 2 steps.*

tion.

The process of cloaking is illustrated in Figure 4. As shown in part (a), detecting a load-store dependence results in an association among the load, the store and a synonym. When a subsequent instance of the store is encountered and a dependence is predicted (part (b), action 1), this association results in the generation of a new version of the synonym (action 2). Storage for this synonym is preferably provided in the *Synonym File (SF)* which is a small, low-latency/high-bandwidth storage structure. The storage element is initially marked as empty as no value is yet available. Upon value reception the synonym file entry is updated and marked as full (action 3). Finally, when the store computes its address it accesses memory as it normally would (action 4). When the appropriate instance of the load is encountered memory dependence prediction is used. Provided that prediction is correct, the same synonym is derived (part (c), action 5). This synonym can now be used to locate the appropriate SF element (part (c), action 6). Instructions consuming the load's value may at this point execute speculatively using the value read from the synonym file (action 7). When the load's address becomes available, the memory system is accessed to read the actual value (action 8). The memory value is compared with the value obtained earlier via the cloaking mechanism. If the two values are the same, cloaking was successful and no further action is required. Otherwise, data value mispeculation occurs, and any instructions that used wrong data have to be re-executed. It should be noted that the above discussion covers one possible sequence of events. Other sequences are possible in practice. For example, the load may be encountered before the store writes a value in the SF. In any case, cloaking still provides the benefit of establishing a communication link early without requiring address-calculation and disambiguation.

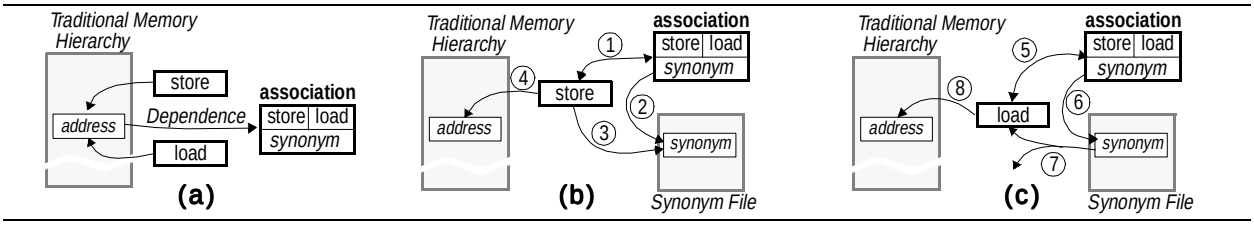

Figure 4. *Streamlining memory communication via speculative memory cloaking: (a) Detecting a dependence results in an association between the dependent load and store instructions and in the creation of a synonym for the dependence, (b) A later instance of the store creates a new version of the synonym, (c) A later instance of the load locates the synonym and uses the data speculatively.*

To perform cloaking we need to be able to: (1) predict dependences, (2) create synonyms, associate them with the dependent instructions and assign storage for the communication, and (3) verify the speculatively communicated values. In Sections 4.1 and 4.3 we discuss each of these requirements in detail. Finally, we present an implementation of cloaking in Section 4.4.

Before we proceed with our description we should note that in the discussion that follows we make the assumption that the value read by a load is always produced by a single store. However, since loads and stores may operate on different data types, this might not be always the case. While support for memory communication among loads and stores that operate on different data

types might be possible [21] we do not consider such options here.

4.1 Detection and Prediction of Dependences

For the purposes of this work we use a history-based memory dependence predictor. This predictor works by detecting memory dependences through the memory address space and by using this information to predict future dependence behavior. Regarding memory dependence detection there are two considerations: (1) how dependences should be reported, and (2) what is the desired scope of this mechanism (i.e., how many loads and stores it should detect dependences for).

Since we want to initiate predictions early in the pipeline we require that the dependence detection mechanism reports dependences as (store PC, load PC) pairs. Regarding the scope of the detection mechanism and as the analysis of Section 3 suggests to capture a large fraction of dependences we need to be able to detect dependences over several addresses (e.g., 128). This is possible if we maintain a record of recent stores (e.g., their PC) along with the memory address each touched in a *Dependence Detection Table* (*DDT*). Dependence detection could be initiated when loads access memory. Using its address a load can locate the last store that updated the particular memory location. At this point the identities of both the load and the store are known: the store PC is recorded in the DDT entry while the load's PC is readily available.

With a dependence detection mechanism in place, the next step is devising a history-based dependence prediction scheme. The most straightforward prediction scheme is to record and predict dependences as (load PC, store PC) pairs. Unfortunately, we found [21] that such a scheme often will have to: (1) predict among many possible dependences (e.g., a load has many possible producing stores that alternate in the execution stream), and (2) predict multiple dependences at the same time (e.g., a store has many consuming loads). For these reasons, we treat dependence prediction as a two step process. In the first step, a prediction is made on whether the given load or store *has* a dependence (i.e., the *dependence status* of the instruction), and in the second step, a prediction is made to decide with which load or store the dependence is with.

We have found that the dependence status of loads and stores rarely changes making simple counter-based predictors highly accurate [19]. Instead of predicting the exact dependence we found it sufficient to use a level of indirection in representing all possible dependences a store or a load has. To do so we use a scheme which assigns a common synonym to all dependences that have common producers (stores) or consumers (loads). This synonym is used to identify all these dependences collectively. We can then determine which of all the possible dependences is currently observed by a mere inspection of the incoming instruction stream as done for register dependences.

We perform assignment of synonyms to dependences using an incremental method which we explain using the example code fragment of Figure 5, part (a). This code has two read-after-write dependences: (STORE_1, LOAD) and (STORE_2, LOAD). We next explainn how eventually both dependences are assigned the same synonym. During the first iteration, one of the dependences, for example the (STORE_1, LOAD), is detected. A new synonym is allocated and associated with both STORE_1 and LOAD. At a later iteration the (STORE_2, LOAD) dependence is detected. We now associate the same synonym with STORE_2 also (the synonym is readily available as it is associated with LOAD).

When the aforementioned method is used it is possible to detect a dependence between store and a load that have different synonyms already assigned to them. Consider for example the code fragment of Figure 5, part (b). Dependences can be encountered in the following order: first (STORE_1, LOAD_1), then (STORE_2, LOAD_2), and finally, (STORE_1, LOAD_2). When the first two dependences are detected, different synonyms are assigned to them as they share no instructions. As a result, when the third dependence is encountered, STORE_1 and LOAD_2 have different tags. At this point it is often desirable to merge all dependences together by assigning one common synonym to all four instructions. This is desirable in our example since which excatly store feeds a load depends on the current control path. If a different synonym is assigned to each store, then a load will have to predict among multiple synonyms. There are cases where we should not assign a common synonym. However, we have found that if one policy is to be used for all dependences, *always* assigning a common synonym is better than *never* doing so. To assign a common synonym to all such dependences can be done by replacing all instances of one synonym with the other. This is the method we originally used [21]. Doing so would probably require a broadcast mechanism, an undesired feature. Alternatively, we could use the approach suggested by Chrysos and Emer in the context of speculation/synchronization [6]. In their approach, the smallest synonym is assigned to both instructions. As a result, eventually all relevant loads and stores are assigned the same synonym. We have found that this method offers virtually the same accuracy as our original full-merge method.

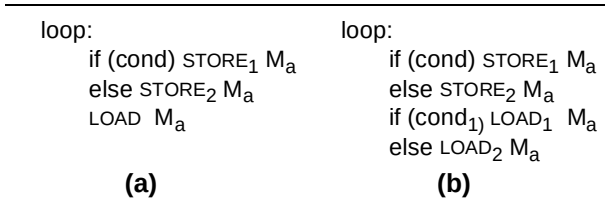

Figure 5. *Code fragments that have multiple true dependences.*

4.2 Synonym Generation and Communication

In cloaking, stores initiate the communication by creating a new version for the synonym when a dependence is predicted. The exact encoding of the synonym is not important. However, it is desirable to provide different versions of the same synonym for unrelated communication at any given point of time. This may require generating different versions for different instances of the same static dependence. This is the case when these instances are simultaneously active (e.g., the values have not been consumed yet). This problem is somewhat similar to register renaming. However, in contrast to register dependences, the lifetimes of instances of a static memory dependence may overlap. For example, this would be the case in a loop containing an "a[i]=a[i - **2**] + doo" statement. An example is shown in Figure 6. While a general solution might be possible, we restrict our attention to the most frequent case where given a store-load dependence no other instance of the store appears in between the dependent instructions. That is, we do not handle dependences generated by statements of the form "a[i] = a[i - **2**] + bee", while we handle dependences generated by statements of the form "a[i] = a[i - **1**] + doo". This simplification allows us to use methods similar to those developed for register renaming. That is, we can use a synonym directly as an index in synonym file provided that the corresponding store has committed. If the corresponding store has not yet committed, we use a small structure, the *synonym rename table*, to associate its synonym with the reservation station where the store resides. Given that stores represent only a small fraction of all dynamic instructions, even a relatively small synonym rename table (SRT) should be sufficient. It should be understood that this is a performance optimization and not a correctness issue. We could simply ignore overlapping instances of the same static dependence at the expense of reduced coverage or accuracy. Moreover, we should note that accessing the SRT can be done as soon as the PC or a load or a store is known possibly before the actual instruction is fetched.

4.3 Verification

Because the communication that takes place in cloaking is based on dependence prediction, any values so obtained are speculative and have to be verified. This can be done by letting the dependent instructions also communicate via memory. This does not necessarily imply that we have to access the memory system [11, 19]. In fact, another memory dependence prediction based technique, the transient value cache will service most relevant accesses hiding them from the rest of the memory hierarchy [21].

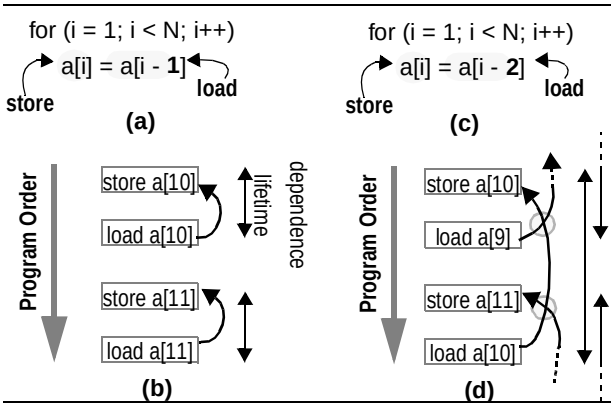

Figure 6. *Examples illustrating dependences whose lifetimes do not overlap (parts (a) and (b)) or do overlap (parts (c) and (d)). Dependences are marked with thick arrows. Dependence lifetimes are marked with thin arrows.*

The support required for invalidating and re-executing instructions that used incorrect data is no different than that required for memory dependence or value speculation [16]. Two options have been proposed to date: (i) squash, and (ii) selective invalidation [16, 15, 26]. In squash invalidation, which is also used on branch mispredictions, all instructions after the mispeculated one are invalidated and re-executed. In selective invalidation only those instructions that used incorrect data are re-executed. While squash invalidation requires no more hardware than what is typically found in modern processors (it is also used to support control speculation) its performance penalty is relatively high. Selective invalidation on the other side offers relatively low performance penalty at the expense of added hardware cost and complexity. In fact, support for selective invalidation is in our opinion still in an experimental phase and whether such mechanisms are practically possible has yet to be demostrated.

4.4 Implementation Aspects

In this section we describe an implementation of the speculative memory cloaking technique. We partition the support structures in the following: (a) *dependence detection table* (DDT), (b) *dependence prediction and*

naming table (DPNT), (c) *synonym file* (SF) and (d) synonym rename table (SRT). As we explained earlier, the DDT is used to detect dependences. An entry of this table consists of the following fields: (1) Data Address (ADDR), (2) Store PC (STPC) and (3) a valid bit. This information identifies the store that last updated the given word data address. The DPNT is used to identify, through prediction, those loads and stores that have dependences. It also provides the tags that are used to create synonyms for the dependences. An entry of this table comprises the following fields: (1) instruction address (PC), (2) dependence predictor (PRED), (3) dependence tag (DTAG), and (4) a valid bit. The instruction address identifies the load or the store this entry corresponds to. The purpose of the dependence predictor field is to provide an indication on whether a dependence exists. Finally, the dependence tag field is used to identify the dependences of this instruction. The SF is used to provide storage for synonyms. SF entries have the following fields: (1) name, (2) value, (3) full/empty bit, (4) valid bit. Based on the exact configuration used, some of the fields may not be required (e.g., we may not use a name field in a direct mapped SF) and some structures can be combined (e.g., we can merge the DPNT and the SF, or the register file and the SF). An SRT entry contains 3 fields: (1) synonym, (2) reservation station tag, and (3) valid. If the valid flag is set, the entry maps the given synonym to the reservation station where the producing store resides. [21, 19] provide examples illustrating the operation of the support structures. We omit such a description due to space limitations.

Figure 7 illustrates how the support structures can be integrated into the pipeline of a dynamically scheduled processor. Loads and stores access the DPNT upon entering the pipeline to obtain a prediction and a synonym. If no dependence is predicted no further action is taken. Stores create an SRT entry for the provided synonym. When a store commits it updates the DDT, releases the SRT entry and writes its value into the SF. Loads using the predicted synonym access the SRT to find where the value resides. If an entry is found it points to the reservation station for the producing store. Otherwise, the synonym resides in the SF. On commit time, loads probe the DDT to detect dependences. This information along with information about the success of any speculation attempts on this load are used to update DPNT. Verification and speculation resolution are done when loads access memory. A description of the speculation resolution mechanism can be found in [19].

5 Speculative Memory Bypassing

In typical load/store architectures, stores and loads do not compute values. Loads and stores are simply used to pass the values that some other instructions produce to some other instructions that consume them. If we knew that a store and a load are used for inter-operation communication we could have eliminated them altogether. This is exactly the goal of s*peculative memory bypassing*. This technique converts a DEF-store-load-USE chain into a DEF–USE chain whenever the load-store dependence is predicted and the DEF and USE instructions coexist in the instruction window. In this case, the value can speculatively flow directly from the actual producer (DEF) to the actual consumer (USE). This concept we illustrate in Figure 8 using the *DEF–store–load–USE* chain shown in part (a). Even though cloaking may allow communication between the store and the load, the value will still have to travel through these two instructions before it can reach USE. However, as shown in part (b) with bypassing, the value can be sent directly from DEF to USE. As was the case with cloaking, this communication is speculative and has to be verified via the traditional memory name space. Note that bypassing is different than cloaking only when the dependent load and store co-exist within the instruction window.

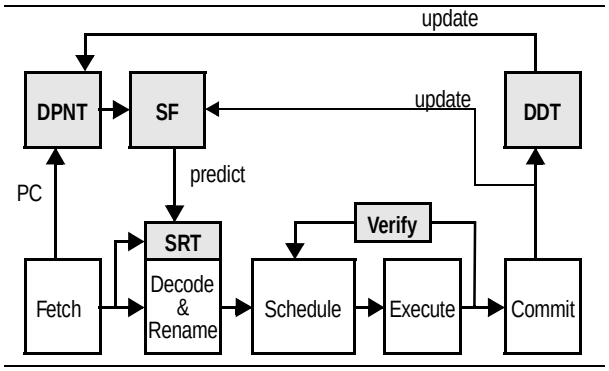

Figure 7. *An out-of-order processor pipeline with a cloaking mechanism*

Speculative memory bypassing can be implemented as a straightforward extension to cloaking. We explain the exact process using the working example of Figure 8, part (c). At step (1), instruction *DEF* is decoded and register renaming creates a new name, *TAG1*, for the target register *R1*. At step (2), the store instruction is decoded and as part of register renaming it locates *TAG1* the current name of its source register *R1*. In parallel, via the use of cloaking, a synonym is created. To perform bypassing, at this point we associate the synonym with the store's source register *R1* name TAG1. This association is be done by recording TAG1 in the SRT entry. At step (3), the load instruction is decoded and register renaming creates a new name *TAG2* for the destination register *R2*. In parallel, the load locates the synonym through cloaking. Using the synonym the load may now

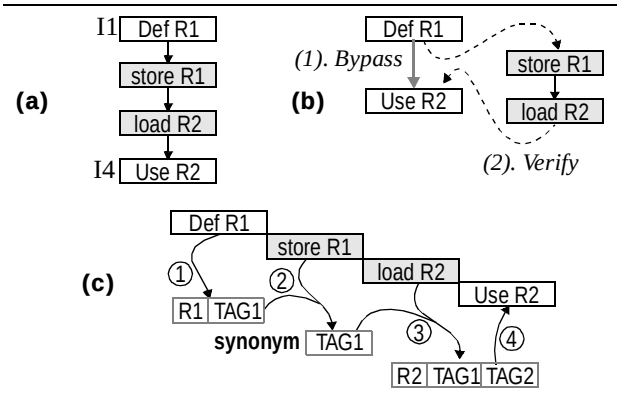

Figure 8. *Speculative Memory Bypassing: (a) Communication path through a load-store dependence. (b) Communication path with speculative memory bypassing. (c) How are the load and the store are removed.*

determine the name, *TAG1*, of the store's source register *R1*. In doing so, the load has determined the storage (e.g., physical register or reservation station) where the actual producer *DEF* will place or has placed the value. This name is speculatively associated with the target of the load *R2*. This way, when at step (4) *USE* is decoded, it can determine that its source register *R2* has two names: one actual *TAG2* and one speculative *TAG1*. By using the speculative name *TAG1*, *DEF* can link directly to *use* and execute as soon as *DEF* produces its value. Later on, after the load has accessed memory, the integrity of the communication can be verified.

Note that bypassing naturally extends for chains that include multiple memory dependences; whenever a store detects that its source register has a speculative name, it can optimistically pass it via the synonym. However, we do not study such an extension in our evaluation. Moreover, speculative memory bypassing becomes more attractive when a store has multiple dependences as it may help in further reducing latency compared to cloaking when register write-back bandwidth is limited. In this case, the speculative value will be propagated to all consumers of all the dependent loads as soon as the actual producers writes its target register. If no bypassing is used, then each of the dependent loads will have to propagate the speculative value to their consumers individually. Finally, bypassing can also be used to eliminate the need for an explicit synonym file at the expense of reduced coverage. In such a design, prediction will have to be restricted to only those dependences that are visible from within the instruction window. In this case, no synonym file is required as bypassing associates synonyms with pre-existing storage elements (i.e., physical registers or reservation stations).

6 Experimental Evaluation

In this section we demonstrate the effectiveness of cloaking and bypassing. Initially we investigate our techniques ignoring timing considerations. This approach allows us to make observations on the nature of memory communication and on its predictability. We then simulate an aggressive dynamically scheduled processor and show that our techniques can improve its performance.

The rest of this section is organized as follows. We start by describing our methodology in Section 6.1. In Section 6.2 we study the accuracy of a cloaking mechanism. In this experiments we assume infinite prediction structures and study the effects of: (1) DDT's of various practical sizes, and (2) of two confidence mechanisms. In Section 6.3 we present various characteristics of the load values predicted by cloaking. We study the base-register, address-space, address-locality and value-locality characteristics of those loads. A detailed description of the characteristics we consider along with a justification is given in that section. In Section 6.4, we measure the performance impact of a combined cloaking and bypassing mechanism under two mispeculation models.

6.1 Methodology

In our experiments we used the SPEC'95 programs which we compiled for the MIPS-I architecture [12] using the 2.7.2 version of the GNU gcc compiler (flags: -O2 -funroll-loops -finline-functions). We translated FORTRAN codes to C using AT&T's f2c compiler. To keep simulation times reasonable we: (1) modified the standard *train* or *test* inputs, and (2) used sampling [28,22,4]. Table 1 reports the dynamic instruction count, the fraction of loads and stores and the sampling ratios per program. While we used relatively shorter inputs we note that virtually no variation was observed in cloaking accuracy compared to the standard SPEC inputs. A description of the modified inputs can be found in [19]. We used sampling only for the timing experiments of section 6.4. The observation size used is 50,000 instructions. The sampling ratios are reported under the "SR" columns as "timing:functional" ratios. These ratios resulted in roughly 100M instructions being simulated in timing mode. We did not use sampling for 126.gcc, 130.li and 147.vortex as cloaking was sensitive to its use. During the functional portion of the simulation the following structures were simulated: I-cache, D-cache, and branch prediction. In the rest of the evaluation we will refer to the benchmarks by using the first numbers of their name shown in Table 1.

We employ both trace and execution-driven timing simulation. Trace based simulation is used for Sections 6.2 and 6.3. Timing simulation is used for Section 6.4. Traces are generated via a functional simulator. All but

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

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

system code references are included. System calls are handled by trapping to the OS of the simulation host. To investigate the potential impact of the proposed techniques, we model a realistic, 8-way superscalar processor with out-of-order execution characteristics. Up to 128 instructions can be in-flight at any given point of time. The processor is pipelined and it takes 5 cycles for an instruction to be fetched, decoded and placed into the 128-entry re-order buffer for scheduling. It takes one cycle for an instruction to read its input operands from the register file once issued. Functional units are fully pipelined and have a latency of 1 cycle except for multiplication and division which take 4 and 12 cycles respectively.

A 128-entry load/store scheduler (load/store queue) is also included. This scheduler is capable of scheduling up to 4 loads and stores per cycle. It takes at least one cycle after a load has calculated its address to go through the load/store scheduler. An important parameter is the use of *naive memory dependence speculation* [20]. That is: (1) a load will access memory even if the addresses of preceding stores are unknown, (2) a load will wait for preceding stores to the same address, and (3) stores post their address to loads even their data is not yet available. It has been shown that memory dependence speculation can have a significant impact on base performance with minimal hardware support [20, 6]. Not including this technique in our base configuration would inflate the performance benefit of our techniques (and of any value speculative technique). Moreover, not using memory dependence speculation impacts the observed critical path artificially inflating the importance of predicting most load values. We note that for our continuous instruction-window processor model naive memory dependence speculation is virtually identical to ideal memory dependence speculation [19].

The base memory system comprises: (1) a 128-entry write buffer, (2) a non-blocking 32Kbyte/16 byte block/4-way interleaved/2-way set associative L1 data cache with 2 cycle hit latency, (3) a 64K/16 byte block/8-way interleaved/2-way set-associative L1 instruction cache with 2 cycle hit latency, (4) a unified 4Mbyte/8-way set-associative/128 byte block with 10 cycle hit latency, and (5) an infinite main memory with 50 cycles miss latency. Miss latencies are for the first word accessed. Additional words incur a latency of 1 cycle (L2) or 2 cycles (main memory). Memory system is event-driven. 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 predictor [17].

6.2 Cloaking Accuracy

The first step in cloaking is dependence detection. The results presented in Section 3 show the fraction of loads that would have dependences detected for DDTs of various sizes. We next measure the fraction of loads that get a value from a cloaking mechanism. For this experiment we assume infinite DPNT and SF structures and vary the size of the DDT. Moreover, we use an non-adaptive predictor: once a dependence is detected cloaking will be used for all subsequent instances of the corresponding load. We consider two metrics: (1) coverage, and (2) mispeculation rate. We define *coverage* as the fraction of dynamic loads that get a *correct* value via cloaking. We define *mispeculation rate* as the fraction of dynamic loads that get an *incorrect* value from cloaking. These results are shown in Figure 9 part (a). Four measurements are taken per benchmark for the following DDT sizes: 32, 128, 512 and 2K shown from left to right. The white bars report cloaking coverage while the diamonds report mispeculation rates. The thick lines report the fraction of loads that have a dependence detected (taken from Figure 3). We can observe that the vast majority of loads with detected dependences get a correct value from cloaking. Not all loads with dependences, however, get a value from cloaking. We found that there is high correlation between unit-distance dependences (i.e., static dependences whose dynamic instance lifetimes do not overlap) and cloaking coverage.

A major source of non-unit distance dependences are recursive functions. Another major source of non-unit distance dependences are arrays which are written to by one loop and read by a later loop (very common in floating-point programs). In this cases, the static dependence is insufficient in representing the dynamic communication relationships. Interestingly, in some cases, cloaking coverage exceeds the fraction of loads with dependences detected. Dependences whose store distances fluctuate are the cause. Once a dependence is detected, it can be correctly predicted even if later the store and the load are so far apart as to escape detection by the DDT. While increasing the size of the DDT generally increases cloaking coverage in some cases a decrease is observed (e.g., 132.ijpeg from 512 to 2K). Infrequent large store distance dependences are the cause. To understand why this is so consider the following sequence: store a[i], store a[j] and load a[i]. A dependence exists between store a[i] and load a[i] every time this sequence is encountered. However, occasionally when i equals j a dependence exists between store a[j] and load a[i]. Our greedy approach to assigning synonyms will incorrectly give the same synonym to all three instructions. Such infrequent dependences often appear at large store distances. Finally, it can be seen that the use of an non-adaptive predictor results in very high mispeculation rates. For this reason we next consider an adaptive predictor.

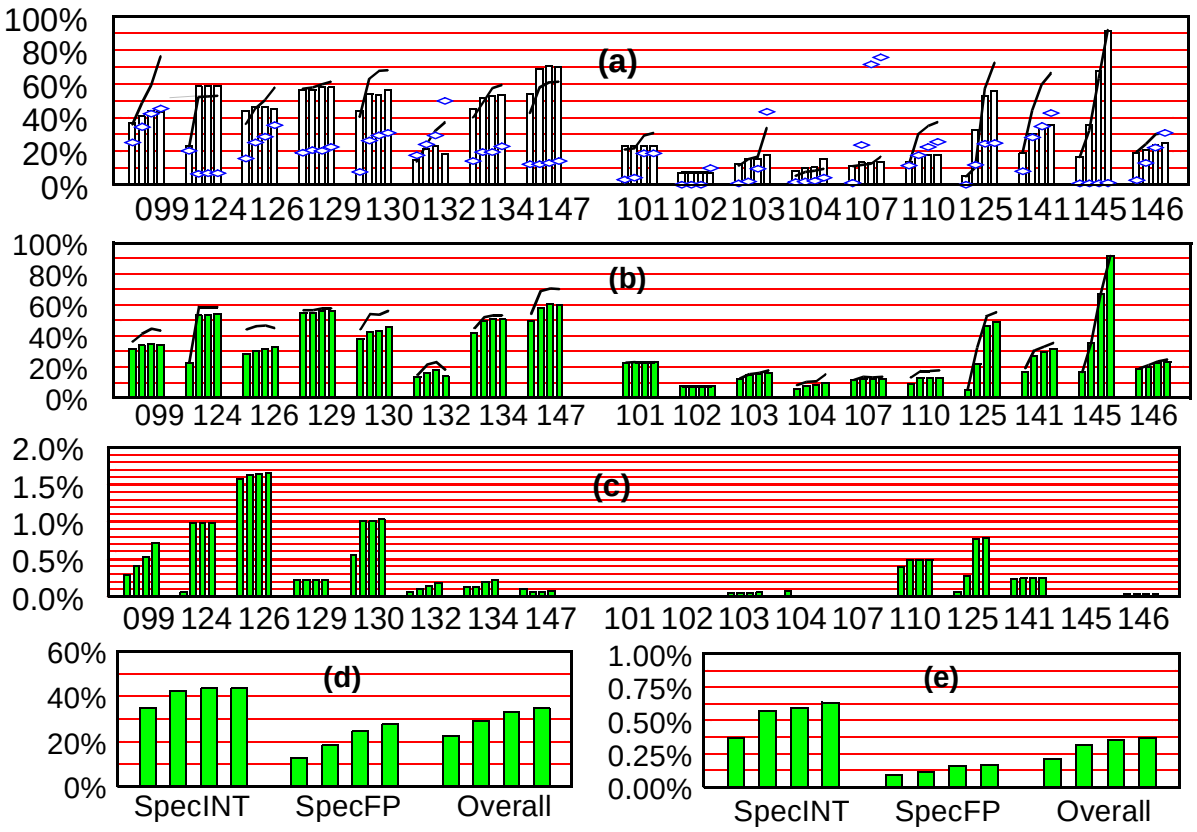

Figure 9. *Accuracy of Cloaking: (a) A non-adaptive cloaking predictor. (b) - (e): An adaptive cloaking predictor. Measurements are as a function of DDT size. DDT sizes included are: 32, 128, 512 and 2k.*

Parts (b) and (c) of Figure 9 report coverage and mispeculation rates respectively with an adaptive cloaking predictor. The lines in part (b) report the coverage of the non-adaptive predictor of part (a). Our adaptive predictor uses a 4 stage automaton per DPNT entry. Once a dependence is detected cloaking is used the next time around. However, upon mispeculation it takes 2 correct predictions for cloaking to be used again. For virtually all programs coverage is very close to that of the non-adaptive predictor of part (a). Moreover, mispeculation rates are drastically lower and often barely noticeable (for some programs the mispeculation rate is not visible since it is below 0.01%). Parts (d) and (c) report average coverage and mispeculation rates over the integer, the floating-point and all benchmarks. It can be seen that cloaking offers high coverage with relatively low mispeculation rates. For example, with a DDT of 128 words (512 bytes) about 45% (integer) and 19% (floating-

point) of all loads get a correct value via cloaking. For the same DDT, only about 0.60% (integer) and 0.12% of all loads get an incorrect value.

While we omit these results we note that a cloaking mechanism with a 4K DPNT and a 1K SF resulted in virtually identical coverage and mispeculation rates. For the rest of this evaluation we use cloaking mechanisms that use a 128 word DDT.

6.3 Characteristics of Predicted Loads

In this section we present a characterization of the loads that get a value from cloaking. In Section 6.3.1 we present a breakdown of predicted loads in terms of the base-register and the address-space used. We do so to provide additional insight on the type of predicted loads. In Section 6.3.2 we measure the address locality of loads and how it is distributed among those predicted by cloaking. In Section 6.3.3 we measure the value locality characteristics of predicted loads and compare cloaking with a last-value load value predictor [16]. The measurements of Sections 6.3.2 and 6.3.3 are included to provide insight on the interaction of cloaking with address prediction and value prediction based schemes.

6.3.1 Base-Register and Address-Space Distribution

Table 2 reports a breakdown of loads that were predicted by cloaking in terms of the base register and the address space used. Percentages are over all dynamic loads. Two rows are shown per benchmark. The top row reports the base-register breakdown while the bottom row reports the address-space breakdown. Both correctly ("R" columns) and incorrectly predicted ("W" columns) loads are included. It can be seen that while the stack and the stack pointer are major contributors to correctly predicted loads other parts of the address space and other base-registers are also correctly predicted. In fact, large fractions of the loads that access the heap of the data segment are correctly predicted. Note that the heap accesses in floating-point programs are an artifact of our compilation process.

6.3.2 Address Locality

Memory latency could be reduced if we could predict the address a load will access. For this reason in this section we consider the address locality characteristics of the values predicted by cloaking. Figure 10 shows a breakdown of loads that exhibit address locality in terms of whether they have a dependence detected or not. We measure *address locality* by counting the fraction of dynamic loads that access the same address as the last instance of the same static load. Address locality provides an upper bound on the accuracy of an infinite, last-address based address predictor with no hysteresis. The lower, dark bar reports the fraction of loads that have a

	SP		GP		OTHER	
	W	R	W	R	W	R
	STACK		DATA		HEAP	
	W	R	W	R	W	R
099	0.01	24.41	0.01	4.97	0.39	4.42
	0.07	25.43	0.34	8.37	0.00	0.00
124	0.00	19.82	0.00	12.53	0.99	21.45
	0.01	19.98	0.98	33.61	0.00	0.22
126	1.49	19.95	0.00	1.55	0.14	8.81
	1.55	25.94	0.02	3.28	0.06	1.09
129	0.00	6.75	0.00	46.61	0.20	1.20
	0.00	6.75	0.21	47.81	0.00	0.00
130	0.54	27.29	0.00	11.89	0.48	3.34
	0.72	27.81	0.00	11.90	0.30	2.82
132	0.00	10.01	0.00	0.01	0.10	5.95
	0.04	13.62	0.00	0.16	0.06	2.19
134	0.11	33.19	0.00	8.53	0.03	7.70
	0.11	33.32	0.01	8.62	0.02	7.48
147	0.01	43.57	0.00	1.70	0.05	13.10
	0.05	54.87	0.00	1.71	0.01	1.78
101	0.00	0.61	0.00	0.41	0.00	21.99
	0.00	22.42	0.00	0.46	0.00	0.13
102	0.00	6.59	0.00	0.00	0.00	0.78
	0.00	6.59	0.00	0.78	0.00	0.00
103	0.00	4.27	0.00	2.88	0.04	7.61
	0.04	10.49	0.00	3.09	0.00	1.18
104	0.00	3.94	0.00	2.18	0.01	1.67
	0.01	4.18	0.00	2.80	0.00	0.82
107	0.00	11.55	0.00	0.05	0.00	0.14
	0.00	11.59	0.00	0.14	0.00	0.00
110	0.00	10.64	0.00	0.01	0.49	1.95
	0.19	12.26	0.30	0.35	0.00	0.00
125	0.00	4.58	0.00	0.00	0.27	17.09
	0.25	19.84	0.03	1.83	0.00	0.00
141	0.00	19.91	0.00	0.25	0.24	6.63
	0.00	24.56	0.24	2.11	0.00	0.13
145	0.00	29.85	0.00	0.05	0.01	5.27
	0.00	30.09	0.01	5.08	0.00	0.00
146	0.00	7.48	0.00	0.19	0.03	12.66
	0.00	7.92	0.03	12.40	0.00	0.00

Table 2. *Base-Register and Address-Space distribution of cloaked loads. Two rows are shown per benchmark. The top reports the percentage of cloaked loads whose base register is the stack pointer (SP), global pointer (GP) or none of these two (OTHER). For each base-register type two numbers are given: correctly predicted loads (R) and incorrectly predicted loads (W). Percentages are over all dynamic loads. The second row reports a breakdown of cloaked loads in those that access the stack, the data or the heap address spaces.*

dependence detected and also exhibit address locality. The upper part reports loads that have no dependence detected and exhibit address locality. The diamonds report the fraction of loads that get a correct value from cloaking. It can be seen that often loads that get a value from cloaking do not exhibit address locality. Furthermore, many loads that do exhibit address locality do not get a value form cloaking. The results of this section suggest that cloaking and address prediction are orthogonal and could be combined for improved coverage. However, such an investigation is beyond the scope of this work

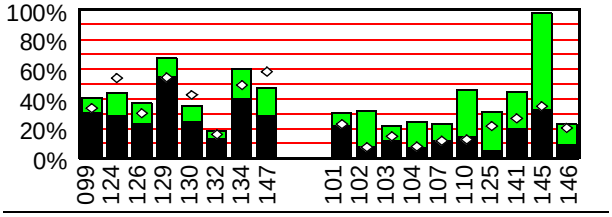

Figure 10. *Address-locality and Cloaking. Dark bar: loads with dependence detected that exhibit address locality. Light bar: loads that exhibit address locality but have no dependence detected. Diamonds: loads that get a correct value from cloaking.*

6.3.3 Value Locality

Memory latency could be also reduced by predicting load values directly. For this reason in this section we compare a load value predictor with a cloaking mechanism. For this experiment we simulate a last-value predictor with 16K entries. The cloaking mechanism we use has an 8K DPNT, a 128-entry DDT and a 2K synonym file. All structures are fully-associative. Figure 11 reports coverage (part (a)) and mispeculation rates (part (b)). Value prediction results are shown by the light left bar, while cloaking prediction results are shown by the right dark bar. In terms of coverage the results are mixed. None of the two techniques appears to have a clear advantage. However, in terms of mispeculation rates, cloaking is superior suggesting that dependence behavior is more stable than value behavior.

Coverage and mispeculation rates are insufficient for comparing value prediction and cloaking. A better comparison considers which loads cloaking predicts that value prediction doesn't and vice versa. These results are shown in Table 3. Column VP reports the fraction of loads that are correctly predicted by value prediction and not by cloaking. Column CLOAK reports the fraction of loads correctly predicted by cloaking but not by value prediction. These results demonstrate that while some overlap exists between load value prediction and cloaking, the two techniques also cover rather large fractions of different loads. One could argue that more complex value predictors may be used to potentially cover all

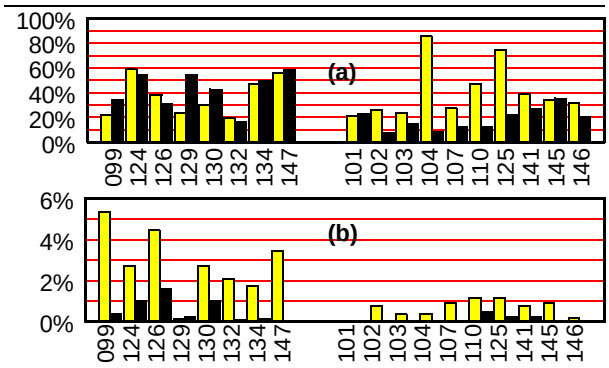

Figure 11. *Comparing a last-value load value predictor and a cloaking predictor. (a) Correctly predicted loads. (b) Incorrectly predicted loads. Gray bars: load value prediction. Dark bars: cloaking.*

loads that cloaking predicts. However, and besides that better cloaking predictors may be possible, we note that, if so desired, cloaking could be viewed as a value prediction enhancing technique.However, it exploits regularities in the dependence stream and not the value stream.

Bench.	VP	CLOAK	Bench.	VP	CLOAK
099	11.72	23.63	**101**	7.79	10.22
124	19.01	13.68	**102**	24.84	6.43
126	24.08	16.63	**103**	16.01	7.13
129	10.00	40.85	**104**	80.54	3.01
130	15.20	29.43	**107**	18.22	2.36
132	12.61	9.12	**110**	37.43	2.69
134	19.42	21.92	**125**	54.99	2.02
147	26.67	29.34	**141**	20.88	9.08
			145	28.38	29.59
			146	21.29	10.12

Table 3. *Comparing Cloaking and Load Value Prediction. Column VP reports the fraction of loads that are correctly value predicted but not correctly cloaked. Column CLOAK reports the fraction of loads that are correctly cloaked but not correctly value predicted.*

6.4 Performance Impact

Having shown that program behavior is such that cloaking could predict large fractions of all loads in this section we measure its performance impact. We evaluate a combined cloaking/bypassing mechanism that includes: (1) a 4K 2-way set associative DPNT, (2) a 1K 2-way set associative SF, (3) a 128 word fully-associative DDT, and (4) a 128 SRT. Up to 4 loads or stores can access each prediction structure simultaneously. Prediction updates occur at commit time. Moreover, mispeculations are signaled on the consumers of loads. As a result, no mispeculation is signalled on incorrectly predicted

load values that have not been used by any instruction.

Compared to Figure 9 and for most programs this delay rarely changed prediction coverage. However, mispeculation rates increased. Noticeable differences in coverage were observed for 126.gcc and 134.perl. In 126.gcc coverage was up by 7% and for 134.perl it was down by 9%. For mispeculation handling we studied three models: (1) squash invalidation, (2) selective invalidation, and (3) oracle. The first two were described in Section 4.3. The oracle model does not use incorrect predictions. We do not include performance results for the oracle model here. However, we note that performance was virtually identical to that of selective. In fact, in some cases using an incorrectly predicted value proved beneficial. This is the case for some values that were partially correct. It turns out that some computations depend only on part of their input values (e.g., for AND X, 0xf only the lower 4 bits of X are important). When mispeculation is detected, only the immediate consumers of the mispeculated value are notified. They will notify their consumers only if after they are re-executed and the value produced is different than that they produced with the mispeculated value. More information on the speculation resolution mechanism we used is given in [19].

Figure 10 shows the relative performance of cloaking. The gray bars report *speedups* with selective invalidation. The white and dark bars report *slowdown*s and *speedups* respectively with squash invalidation. It can be seen that cloaking with squash invalidation is not robust and rarely beneficial. On average (harmonic mean) this mechanism leads to slowdowns of 5.63% (integer), 1.59% (floating-point) and 3.43% (all programs). Cloaking with selective invalidation, however, is robust and in most cases beneficial. It never reduced performance. On average this mechanism offers speedups of 4.28% (integer), 3.20% (floating-point) and 3.68% (all programs).

As we have seen in Section 6.2, cloaking mispeculation rates are often very small. Nevertheless, we see that squash invalidation more than often leads to slowdown. This should not come as a surprise. Cloaking mispredictions often occur when most of the instruction window is full and control prediction is correct. Under these conditions, flushing part of the window proves highly disruptive and has a large impact on performance. A similar result has been reported for memory dependence mispeculations where it was shown that even very small mispeculation rates have great impact on performance [20].

Part (b) of Figure 12 shows a breakdown of predicted loads in (1) those that get a value through cloaking (gray bar), and (2) those that get a value through bypassing (dark bar). There is no definite pattern here. In some cases cloaking covers most of the values (e.g., 099.go) and in some cases bypassing accounts for most of the

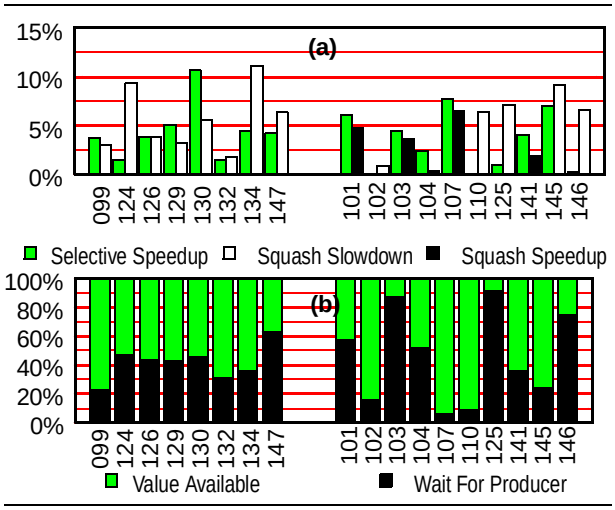

Figure 12. *(a) Relative performance of a cloaking mechanism. Grey bars are with selective invalidation. These bars report SPEEDUP. White and dark bars are with squash invalidation. White bars report SLOWDOWNS, dark bars report SPEEDUP.*

predictions (e.g., 125.turb3d). Two primary forces are at work: the observed critical path and the distance in instructions between dependent loads and stores. Both are program specific attributes. Bypassing can be applied only when both the dependent store and load appear within the window. Even when they do, the store value may be available long before the load is decoded.

The results of this section suggest that even over a highly aggressive base configuration cloaking can offer respectable performance improvements. However, benefits are limited by the control prediction accuracy and by the optimistic assumptions about load/store queue and data cache latencies (2 cycles and 3 cycles respectively). As instruction windows and load/store queues grow, as control prediction accuracy improves and as wire lengths start dominating it should expected that the latency of memory communication will increase. In such environments it is likely that the performance impact of cloaking and bypassing will be higher.

7 Related Work

Memory dependence prediction was introduced in [20] where it was used to improve the accuracy of memory dependence speculation in the context of a split-window processor.

Numerous techniques that attempt to predict the addresses of loads and stores have been proposed both in hardware and in software [e.g., 1,2,3,8,9,25,5]. Address prediction provides no explicit information about communication. A mechanism may be required to compare any predicted addresses for reducing communication latency. Moreover, address prediction relies in regulari-

ties in the address stream. Cloaking is orthogonal to such techniques and may streamline memory communication even if the access pattern defies prediction.

In this work we were motivated by the large fraction of short-distance memory communication. Numerous memory referencing behavior studies for the purpose of optimizing the memory hierarchy exist. Two are most relevant to this work: McNiven and Davidson [18] analyzed memory referencing behavior and suggested using compiler hints to identify values that are killed in order to reduce traffic between adjacent levels of the memory hierarchy. Huang and Shen studied the minimal bandwidth requirements of current processors, as a function of instruction issue rate, memory capacity and memory bandwidth [10].

Value speculation may effectively reduce the latency of memory communication independently of whether the load has a true dependence or not [16]. The success of this approach relies on the ability to track and predict the actual values. In cloaking we do not directly predict the load value, rather we predict its producer.

Cloaking and bypassing were originally reported in [21]. Simultaneously, Tyson and Austin proposed *memory renaming* a technique similar to cloaking [26]. They combined memory dependence prediction with value prediction and studied their combined effect. They utilized a last-producing-store prediction scheme for prediction purposes. A restricted form of cloaking is *alias prediction* [14]. In this technique loads predict a write buffer entry where their producing store resides. This optimization alleviates the need for an SRT.

A mechanism similar to bypassing was proposed by Jourdan, Ronen, Bekerman, Shomar and Yoaz [11]. In their proposal, bypassed loads do not necessarily have to access memory and as result memory bandwidth requirements are also reduced. The Transient Value Cache has a similar effect [21]. They also combined address prediction with cloaking/bypassing. A software guided approach to cloaking was investigated by Reinman, Calder, Tullsen, Tyson and Austin [24]. In their approach, new instructions are introduced that allow the compiler to communicate speculative memory dependences to the hardware. Reinman and Calder also performed a comparative study of load value prediction, memory dependence speculation/synchronization [20], of a variation of the Memory Renaming technique of Tyson and Austin [23] and of address-prediction-based techniques.

While we have investigated cloaking in the context of sequential programs, cloaking could be applied to explicitly parallel programs also. Kaxiras and Goodman have proposed such a mechanism [13].

8 Contributions and Future Directions

We revisit memory design observing that memory is often used as a communication mechanism. From this perspective we identify a number of overheads introduced by traditional address-based memory communication. We propose techniques to alleviate these overheads. Our techniques are architecturally transparent as they utilize dynamically collected dependence information. Our contributions are:

(1) We show that highly-accurate history-based memory dependence prediction is possible.

(2) We show that the traditional implicit specification of memory communication can be dynamically converted into a explicit, albeit speculative form.

(3) We propose speculative memory cloaking and its extension speculative memory bypassing which utilize a dynamically created explicit specification of memory communication to reduce memory latency.

We conclude by commenting on a research direction regarding prediction-based techniques. Prediction-based techniques already empower most modern high-performance processors. Examples include branch prediction, caching and memory dependence speculation. We believe that such techniques will play an increasingly important role in the future. In this context, cloaking and bypassing represent a step toward a class of new prediction techniques that in addition to regularities in the *outcomes* of program execution (e.g., values, addresses and branch directions) also exploit regularities in the *actions* programs take to produce these outcomes. As outcome-based prediction techniques are perfected reaching a point of diminishing returns, action-based prediction techniques represent a promising direction for continuous improvement.

References

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

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

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

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

[5] B.-C. Cheng, D. A. Connors, and W.-M. Hwu. Compiler-directed early load-address generation. In *Proc. MICRO-31*, Dec. 1998.

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

[7] J. Dennis. Data Flow Supercomputers. *IEEE Com-*

puter, Nov. 1980.
[8] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. In *IBM journal on research and development, 37(4)*, July 1993.
[9] M. Golden and T. Mudge. Hardware support for hiding cache latency. In *CSE-TR-152-93, University of Michigan, Dept. Of Electrical Engineering and Computer Science*, Feb. 1991.
[10] A. S. Huang and J. P. Shen. A Limit Study of Local Memory Requirements Using Value Reuse Profiles. In *Proc. MICRO-28*, Dec. 1995.
[11] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proc. MICRO-31*, Dec. 1998.
[12] G. Kane. *MIPS R2000/R3000 RISC Architecture*. Prentice Hall, 1987.
[13] S. Kaxiras and J. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proc. HPCA-5*, Feb. 1999.
[14] M. H. Lipasti. *Value Locality and Speculative Execution*. Ph.D. thesis, Carnegie Mellon University, Pitsburgh, PA 15213, Apr. 1997.
[15] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. on MICRO-29*, Dec. 1996.
[16] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proc. ASPLOS-VII*, Oct. 1996.
[17] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corp., WRL, June 1993.
[18] G. D. McNiven and E. S. Davidson. Analysis of Memory Referencing Behavior for Design of Local Memories. In *Proc. ISCA-15*, May 1988.
[19] A. Moshovos. *Memory Dependence Prediction*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, Dec. 1998.
[20] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. ISCA-24*, June 1997.
[21] A. Moshovos and G. Sohi. Streamlining inter-operation communication via data dependence prediction. In *Proc. MICRO-30*, Dec. 1997.
[22] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. In *IEEE Computer, 31(5)*, May 1998.
[23] G. Reinman and B. Calder. Predictive Techniques for Aggresive Load Speculation. In *Proc. MICRO-31*, Dec. 1998.
[24] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided load marking for memory renaming. Technical Report CS98-593, University of California, San Diego, July 1998.
[25] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. In *Proc. MICRO-29*, Dec. 1996.
[26] G. S. Tyson and T. M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. MICRO-30*, Dec. 1997.
[27] A. H. Veen. Dataflow Machine Architectures. *ACM Computing Surveys, vol. 18*, Dec. 1986.
[28] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Processors. In *Proc. ISCA-23*, May 1996.