Streamlining Inter-operation Memory Communication via Data Dependence Prediction

Andreas Moshovos and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
{moshovos, sohi}@cs.wisc.edu

Abstract

We revisit memory hierarchy design viewing memory as an inter-operation communication agent. This perspective leads to the development of novel methods of performing inter-operation memory communication:

(1) We use data dependence prediction to identify and link dependent loads and stores so that they can communicate speculatively without incurring the overhead of address calculation, disambiguation and data cache access.

(2) We also use data dependence prediction to convert, DEF-store-load-USE chains within the instruction window into DEF-USE chains prior to address calculation and disambiguation.

(3) We use true and output data dependence status prediction to introduce and manage a small storage structure called the Transient Value Cache (TVC). The TVC captures memory values that are short-lived. It also captures recently stored values that are likely to be accessed soon. Accesses that are serviced by the TVC do not have to be serviced by other parts of the memory hierarchy, e.g., the data cache.

The first two techniques are aimed at reducing the effective communication latency whereas the last technique is aimed at reducing data cache bandwidth requirements. Experimental analysis of the proposed techniques shows that: (i) the proposed speculative communication methods correctly handle a large fraction of memory dependences, and (ii) a large number of the loads and stores do not have to ever reach the data cache when the TVC is in place.

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 (higher bandwidth/lower latency) 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 method for this purpose. 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 storage structures used to build memory hierarchy components, and hopefully, to new ways of thinking about such hierarchies. We expect a fair amount of on-chip resources in future processors to be devoted to "non-traditional" storage structures and a fair amount of effort devoted to "non-traditional" methods as we attempt to solve the "memory problem". We describe three such methods in this paper: *Speculative Memory Cloaking* (or simply *cloaking*), *Speculative Memory Bypassing*, and *Transient Value Cache* (TVC). The first two methods are aimed at reducing the effective communication latency whereas the third method is aimed in addition at increasing the effective memory bandwidth.

In speculative memory cloaking we dynamically convert implicitly specified memory communication into an explicit, albeit speculative, form. To do so, we use *data dependence prediction* to explicitly link loads and stores that are likely to be dependent. 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. When the dependent load and store co-exist in the instruction window, further reduction in the communication latency is possible with speculative memory bypassing. In this technique loads and stores that are predicted as dependent are speculatively removed from any DEF-store-load-USE chains that contain them. Values can then flow directly from the actual producer (DEF) to the actual consumer (USE). Since both cloaking and speculative memory bypassing are speculative in nature, the communication performed in this manner has to be eventually verified via the traditional memory name space.

The transient value cache is a novel memory hierarchy component that attempts to capture that part of the memory space through which recently stored values are accessed or where short-lived values reside. In this technique, we use both true and output *data dependence status prediction* (i.e., whether a load or a store have a dependence with a recent store) to selectively redirect memory accesses to the TVC. Such accesses may not have to go to the data cache, consequently reducing the data cache bandwidth demand.

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 communication 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, and in Section 6 we present the TVC technique. We provide a quantitative assessment of the proposed techniques in Section 7. Finally, we comment on related work in Section 8 before we offer concluding remarks in Section 9.

2 Memory as an Inter-operation Communication Agent

Memory communication can be viewed as a two step process where first the dependences are established and then the actual values are communicated. To streamline memory communication we need: (i) to establish the dependences as quickly as possible and then, (ii) to provide storage structures that best meet the communication requirements (e.g., low latency/high bandwidth). Attempts to meet these two goals face obstacles that stem from the way memory communication is specified and from practical and economical restrictions on the amount and the type of storage structures that we can provide.

The traditional implicit specification of memory communication imposes unnecessary delays on the communication since to establish the dependence relationships we have first to carry out an address calculation and then to perform disambiguation, even though the desired value may be available long before these tasks can be performed. An explicit representation in which the identity of the producing store is known to the consuming load and vice versa, does not impose unnecessary delays; the communication can be initiated as soon as the identities of the two instructions become known.

Another concern with the implicit specification of memory communication is that it results in name-centric memory hierarchy design approaches where the emphasis is placed upon the attributes of the memory names. For example, caches are designed to take advantage of empirical observations about the temporal and spatial locality of memory names. However, as the demands placed on memory hierarchies increase, more sophisticated methods are sought (see Section 8). These new methods may be helped by information about the communication itself, in addition to information about the names.

An explicit representation of memory communication may open up new possibilities since it encourages a communication-centric approach to memory hierarchy design. There are cases where viewing memory hierarchy design in terms of the communication itself (i.e., the loads and the stores) rather than in terms of memory names is advantageous. For example, consider a number of dependences which during run-time, and when viewed in isolation, exhibit predictable lifetimes. This lifetime information may be used to decide where in the memory hierarchy to place the associated values. If the corresponding values are mapped to the same name, the resulting behavior in terms of the name may defy prediction. Another example is when a dependence which during run-time exhibits predictable lifetimes uses different memory names over time. In this case, the past behavior of the dependence can be used to predict the behavior of memory names that we may not have yet encountered.

Motivated by the aforementioned observations, in this paper we are primarily concerned with methods of converting the traditional implicit specification of memory communication into an explicit form. Dependence relationships could be determined and expressed in an explicit manner statically. Nevertheless, in this paper we do not consider a static approach since it would require static knowledge of the dependences, and it would also involve changing the program representation completely. Instead, we investigate dynamic approaches.

We utilize *data dependence prediction* to establish and 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 calculation, disambiguation and data cache access. However, since the dependences are speculative the communication performed *eventually* has to be verified via the traditional memory name space. Furthermore, the amount of history information we can record places a bound on the fraction of the memory communication we can convert.

Finally, we also take a first step toward annotating this representation with dynamically collected information that can be used to develop and manage novel memory hierarchies. We do so by using *data dependence status prediction* to annotate memory accesses such that stores whose values that are likely to be killed soon and loads that are likely to access a recently stored value are serviced from a small storage structure and without consuming data cache resources.

3 Memory Traffic Analysis

To motivate the proposed methods we first present an empirical study of the memory inter-operation traffic of the SPECint95[1] benchmarks on a MIPS architecture (the benchmarks, architecture and methodology are detailed in Section 7). To get an estimate (i) of the fraction of the memory operations we can serve with a dependence based mechanism, (ii) of how much the storage might we require for this speculative explicit communication, and (iii) what attributes we might desire of it, we measure: (i) the percentage of loads that read a value created by a preceding store (true dependence), and (ii) the percentage of stores that are killed by a later store (output dependence). We present both characteristics as a function of the *store distance,* which is defined as the number of stores that appear between the dependent instructions in the dynamic instruction stream. We use store distance as our metric since it provides an upper bound on the number of data values that have to be recorded in order to detect and capture the particular dependence.

Part (a) of Figure 1 reports the percentage of dynamic loads that read a data value produced from a preceding store (store distance range shown is 8 to 8K) whereas part (b) reports the percentage of dynamic stores that are killed by a later store. It can be seen that for almost all programs about 50% of all dynamic loads get their value from a store that is at most at store distance 256 (marked by a continuous vertical line) and that about 60% of all stores are killed within a store distance of 256.

These results indicate that a mechanism which can record and detect dependences across the last 256 stores can potentially service around 50% of all loads and cut down about 60% of the store traffic. 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. Later, motivated in addition by the large fraction of store values that are quickly overwritten, in Section 6, we attempt to also the reduce the bandwidth requirements imposed on the data cache. To do so, we introduce a separate but small storage structure and redirect to it those memory locations through which communication of true dependences occurs or recently stored values are killed.

1. We have performed a similar study for the SPECfp95 programs. The results are available via: ftp://ftp.cs.wisc.edu/sohi/micro30-memtraffic.ps

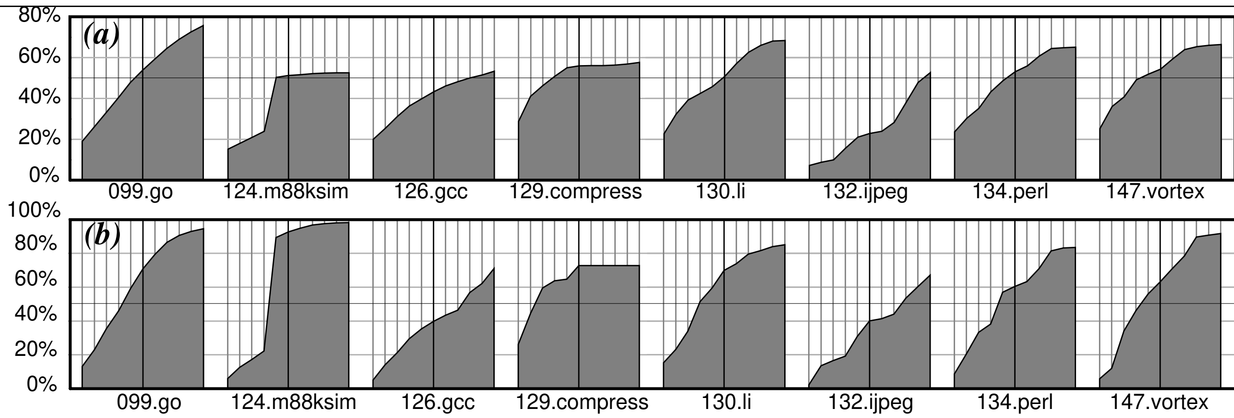

Figure 1. *(a) Distribution of dynamic load/store (true) dependence distances, (b) Distribution of dynamic store/store (output) dependence distances. For both graphs the X axis represents distance in stores. Samples are taken at store distances that are powers of two starting from 8 and ending at 8K. Y axis represents percentage over all loads or all stores accordingly.*

4 Speculative Memory Cloaking

The purpose of cloaking is to streamline memory communication by dynamically converting the implicit specification of dependences into an explicit form. In cloaking, dependence prediction is used to identify loads and stores that are likely dependent. The dependent load and store are then explicitly linked via a new name, a *synonym* which uniquely identifies the dependence (e.g., the synonym can be the (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 in the nature of the association between the name and the instructions that use it. In contrast to a data address, the synonym uniquely identifies the dependent instruction pair so that the load and the store can each derive the synonym based solely on their identity (PC). We use the term *speculative memory cloaking* to signify that memory is hidden since the communication takes place speculatively through a dynamically created name space and without knowledge of the memory location used by the program; no association between the storage used by the synonym and the memory address is built (in contrast to what is done in *memory renaming* (e.g., [5])).

The process of cloaking is illustrated in Figure 2. As shown in part (a), a detection of a load-store dependence results in an association between the load, the store and a function that can be used to derive preferably unique synonyms for future instances of the dependence. When a later instance of the store instruction is brought into the instruction window and the existence of a dependence is predicted, this association results in the generation of a synonym (part (b), action 1) and further, in the allocation of physical storage for that synonym (action 2). Storage for the synonym is preferably provided in the *Synonym File (SF)* which is a small, low latency/high bandwidth storage structure. The storage element is initially set to indicate that the value is not yet available and is updated with the actual value as soon as the latter becomes available (action 3). Finally, when the store computes its data address, the value is also written to the traditional memory system (action 4). When the appropriate instance of the load is brought into the instruction window and the existence of a dependence is predicted, the association is used again to derive the synonym (part (c), action 5) and consequently, to locate the appropriate element in the synonym file (part (c), action 6). Instructions that use the load value may at this point execute speculatively using this value (action 7). When the load data address becomes available, the memory system is accessed to read the actual value (action 8). This 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 mis-speculation occurs, and any instructions that used wrong data have to be re-executed. (In Section 6, we will show that this verification can be hidden from the data cache using the TVC.)

Speculative memory cloaking has the following requirements: (1). predicting dependences, (2). creating synonyms, associating them with the dependent instructions and assigning storage for the communication, and (3). verifying the speculatively communicated values. We next discuss each of the requirements in detail.

4.1 Detection and Prediction of Dependences

If cloaking is to succeed, we have to be able to predict dependences. In [15] we have shown that relatively few static dependences are responsible for the majority of the true dependences observed dynamically and that this set exhibits temporal locality. (Stores with output dependences exhibit similar behavior. In Section 6, we make use of this observation to also cut down on the write traffic.) This observation suggests that we may use dependence history to predict future dependences.

The most straightforward prediction scheme is to record and predict dependences as (load PC, store PC) pairs similarly to what was done in [15]. However, such a scheme may have to predict among many possible dependences since, as we will demonstrate in Section 7, different instances of the same static store often observe dependences with instances of different static loads and vice versa. Furthermore, with such a scheme we may have to predict multiple dependences per dynamic store when its value is used by many loads. For these reasons, it is both conceptually and practically convenient to 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.

In Section 7, we will demonstrate that even simple, counter based predictors can predict the dependence status of instructions with relatively high accuracy. Predicting the actual dependence

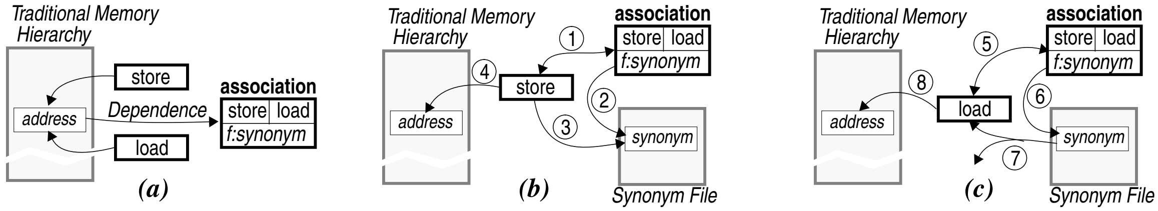

Figure 2. *Streamlining the communication through memory via speculative memory cloaking: (a). a dependence detection results in an association between the dependent load and store instructions, (b). a later instance of the store creates a synonym, (c). a later instance of the load locates the synonym and uses the data speculatively.*

however requires more effort. Although it is conceivable and desirable to design a predictor that attempts to predict directly the actual dependence, we found it sufficient for the purposes of this work to use a scheme which assigns a common tag to all dependences that have common producers (stores) or consumers (loads) and use that tag to identify all these dependences collectively[2]. The processor can then determine which of all the possible dependences is currently observed by a mere inspection of the incoming instruction stream (this is similar to what is done for register dependences).

Since our dependence prediction schemes are based on dependence history, we also need a way of detecting dependences. As the results of Section 3 suggest, to capture a large fraction of the dynamic dependences we need to be able to detect dependences over several stores (e.g., 256), that is, over regions that most likely exceed the instruction window used. This can simply done by keeping a record of the recent stores (e.g., their PC) along with the memory address each touched. There are numerous ways of implementing the dependence detection functionality. A relatively inexpensive and straightforward implementation is via a *Dependence Detection Table* (*DDT*) which is nothing more than a regular but very small cache that records the PC of the store that last touched each recorded address. Note that since the information collected by the detection mechanism is used only for prediction purposes, relatively long detection latencies and detection errors may be acceptable.

4.2 Synonym Generation and Communication

In cloaking, stores initiate the communication by generating a synonym in reaction to the prediction of a dependence. The synonym has a dual role: it identifies the specific instance of the dependence (or dependences in the case of multiple consumers), and it is also used as a handle by the dependent instructions to locate the storage element through which the communication will take place. The exact encoding of the synonym is not important. However, it is desirable for the naming scheme used to provide different synonyms for unrelated communication at any given point of time. The synonym generated has to be associated with all the instructions that will communicate so that they can locate the appropriate storage element using their PCs. This is straightforward given that the dependence prediction mechanism identifies the dependence either explicitly by the (store, load) edge or implicitly via a tag which is associated with the store and the load as described in the previous section. To perform the communication, physical storage has also to be provided for synonyms. The storage elements should provide space for the data value and an indication on whether the value is currently available. Finally, mapping synonyms to storage elements can be done in a variety of ways (e.g., using a direct mapped or a fully associative SF).

4.3 Verification

Because the communication that takes place in cloaking is based on dependence prediction, values so obtained are speculative and have to be verified. This can be done by letting the dependent instructions also communicate via the memory space. The support required for invalidating and re-executing instructions that used incorrect data is no different than that required for dependence or value speculation [12]. Since cloaking requires verification through the memory space it can only reduce the latency of the communication and does not save on bandwidth. As we will describe in Section 6, however, the verification can be done via the TVC, in which case the bandwidth required of the pre-existing memory hierarchy can also be reduced.

4.4 Implementation Aspects

In this section we describe an implementation of the speculative memory cloaking technique and explain its operation by means of an example. Our goal is to demonstrate the feasibility of the required mechanisms and to provide insight about their complexity. We partition the support structures in the following: (a) *dependence detection table* (DDT), (b) *dependence prediction and naming table* (DPNT), and (c) *synonym file* (SF).

As we explained earlier in this section, 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 status 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).

4.5 Working Example

The exact function and use of the support structures is best

---

2. For example in the code: **if** (cond) **then** store_1M**;** **else** store_2M**;** load M the dependences (store_1, load) and (store_2, load) will be both assigned a common tag. The stores and the load will use this tag to link to each other.

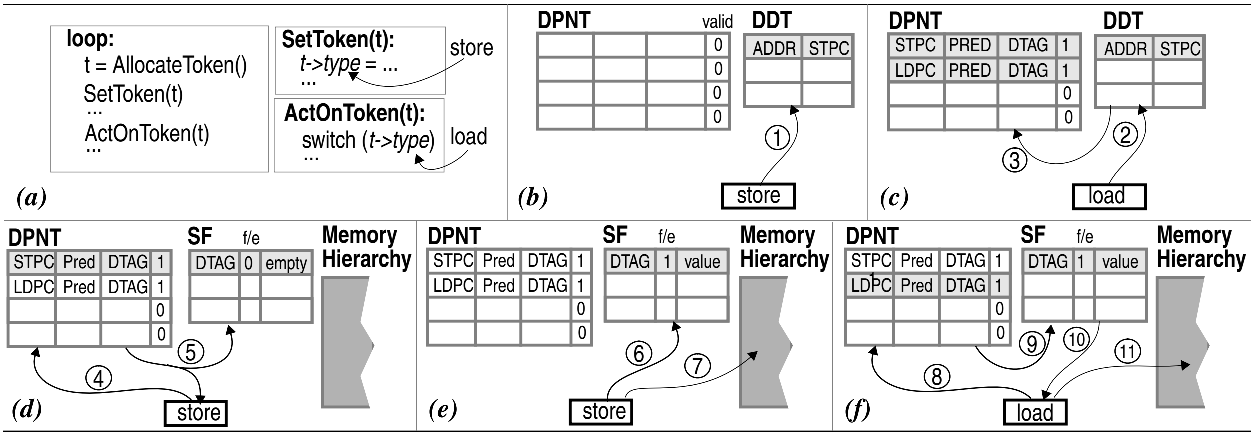

Figure 3. *An implementation of Speculative Memory Cloaking.*

understood by means of an example. In the discussion that follows we use the working example of Figure 3 to demonstrate how an earlier detection of a dependence between a store and a load results in the streamlining of the inter-operation communication the next time the same dependence is encountered. In the discussion that follows we assume that the dynamic dependences result from the execution of the loop shown in part (a). Dynamically, a series of dependences will be observed between instances of the marked load and store. Each of the dynamic dependences will map to a different memory address (we assume that new space is allocated for each token).

In parts (b) and (c) we show the actions that lead to the detection of the dependence. In part (b), the first instance of the store executes and records in the DDT its PC and the data address it updated (action 1). Later on, in part (c), the first instance of the load using its data address probes the DDT (action 2) and determines that a dependence exists with the recorded the store. In reaction to this detection, two entries are allocated in the DPNT one for the load and one for the store (action 3). In addition, a tag is created for the dependence, and it is recorded in both entries. (Since the operation of the DDT has been described in steps 1 through 3, it is not shown in the remaining parts of the figure.)

In parts (d) through (f) the actions that lead to the cloaking of a later instance of the dependence recorded in part (c) are shown. Cloaking is initiated when, as shown in part (d), a later instance of the store enters the instruction window. The PC of the store is used to probe the DPNT for a matching entry (action (4)), and since one is found, its predictor is used to determine whether cloaking should occur. Assuming that the predictor indicates so, a synonym is generated based on the tag recorded in the DPNT entry (for the purposes of this discussion the tag of the DPNT and the synonym are the same), and it is used to allocate space in the SF (action 5). The full/empty bit of the SF entry is set to empty to indicate that the value is not yet available, whereas, the store also records the location of the SF entry since the actual data value, when it becomes available, will have to be written in the SF entry (part (e), action 6). Eventually, the store also accesses the traditional memory hierarchy (part (e), action 7).

When the next instance of the load enters the window (part (f)), as it was done previously with the store, its PC is used to probe the DPNT (action 8). After a match is found and a dependence status prediction is made, the tag recorded in the DPNT entry leads to the generation of the same synonym generated previously for the store. This synonym is used to access the appropriate SF entry (action 9) and to obtain the data left there by the store. At this point the load may use this data to execute speculatively (action 10). Later on, when the data address becomes available, the load accesses the traditional memory hierarchy to obtain the actual data value (action 11). This value is compared against the value read previously from the SF and appropriate action is taken if the two values differ. At this point we may also update the predictors in the DPNT entries for both the load and the store (to locate the DPNT entry for the store the SF entry will have to record the store's PC).

4.6 Other Issues

We now discuss a few issues which relate to the method and the implementation we have described.

4.6.1 Dependences Through Different Data Types

So far we have assumed that the value obtained through a dependence is exactly the one written by a single store. Loads and stores, however, may operate on various data types (e.g., a byte, half word or a full word). So, it is possible to encounter dependences between a store and a load that operate on different data types or, to encounter loads that read a value that is a combination of the values written by many stores. Given a load, there are four possible cases: (1) the dependence is with a single store that operates on the same data type, (2) the loaded value is only part of the value written by a single store, (3) the loaded value is a combination of the values (or, parts of them) written by more than one stores (e.g., the load reads a word whose bytes were each written by a different store), and (4) only part of the value read by the load comes from recent a store (or stores).

The first case does not present a challenge for the mechanisms we have described. For the other three cases we do have the option of not providing support. However, it is desirable to provide support for at least those cases that are relatively frequent or that are critical in terms of performance. For the purposes of this work we base the decision on which types to support solely on frequency. To provide support for the second case we need to be able to determine what part of the store value the load reads. This information cannot be derived from the identity of load only, the actual data address is needed. However, we may employ a simple prediction scheme in which we record (using a 4 bit mask in the DPNT entry for the load) the location of the bytes that the load read last time the dependence was observed and use this information to extract the same part of the store value the next time the dependence occurs (this also allows us to provide support for sign-extension). However, this method

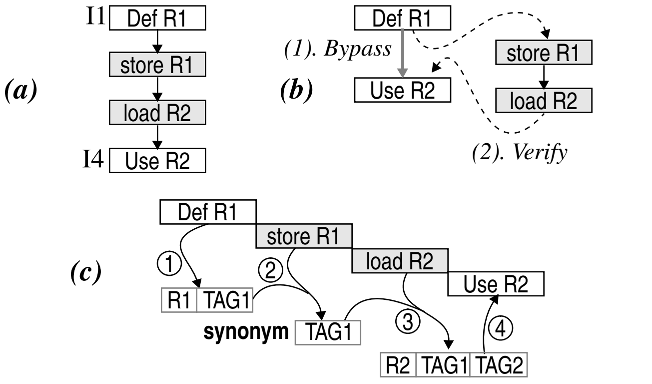

Figure 4. *Speculative Memory Bypassing: (a). Regular communication path through a memory dependence. (b). Communication path with Speculative Memory Bypassing. (c). How are the load and the store taken off the communication path.*

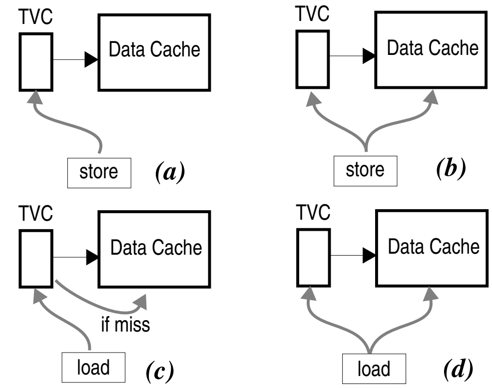

Figure 5. *Transient Value Cache operation. Store: (a). output dependence likely, (b). output dependence unlikely. Load: (c). true dependence likely, (d). true dependence unlikely.*

will fail if over time the load reads different parts of the store value. Even though it is possible to extend our mechanisms to support dependences that fall in the other two categories, we do not consider these extensions here since these cases rarely occur in practice.

4.6.2 Multiple Instances of the Same Dependence

It is possible for multiple instances of the same static dependence (i.e., (load PC, store PC) pair) to be simultaneously active. In this case, it is desirable to create a different synonym for each dynamic dependence. Generating a different synonym every time a new instance of the store is encountered is straightforward and can be done in numerous ways (e.g., using a global counter). However, for communication to occur as planned, the same synonym has to be assigned to the appropriate instance of the load. Doing so is straightforward if, in the original program order, the lifetimes of the dynamic dependences are distinct; all we have to do is remember the most recent version per active dependence (similarly to what is done for register dependences). However, since data addresses are calculated dynamically, the lifetimes of the dynamic dependences may overlap (as for example in the following loop that has a recurrence that spans 3 iterations: `for i = 1 to N do a[i + 3] = a[i] + 1`). In this case, remembering the most recent synonym for the static dependence is not sufficient. Instead, the load has to determine which of all synonyms is the appropriate one. Even though support for regular communication patterns can be provided, further investigation of this issue is beyond the scope of this paper.

5 Speculative Memory Bypassing

With cloaking, values can flow quickly from stores to loads. However, in load/store architectures, stores and loads do not compute values[3] rather they are simply used to pass the values that some other instructions produce to some other instructions that consume them. This occurs when either the compiler was unable to establish the dependence statically or when storage in the register name space was not available. *Speculative memory bypassing* converts DEF-store-load-USE chains into DEF–USE chains whenever the load-store dependence is predicted and the DEF and USE instructions co-exist 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 4, using the *I1–store–load–I4* chain shown in part (a). Even though speculative memory cloaking may allow the value to be speculatively communicated between the store and the load, the value will still have to travel through these two instructions before it can reach I4. However, as shown in part (b) with speculative memory bypassing, the value can be sent directly from I1 to I4. As was the case with speculative memory cloaking, this communication is speculative and has to be verified.

Speculative memory bypassing can be implemented as a simple extension to speculative memory cloaking. We explain the exact process using the working example of Figure 4, part (c). At step (1), instruction *I1* is decoded and register renaming creates a new name *TAG1* for the target register *R1*. At step (2), the store instruction is decoded and determines the current name of its source register *R1*. In parallel, via the use of cloaking, a synonym is created for the memory communication. At this point, we also record in the synonym the current name *TAG1* of store's source register *R1*. At step (3), the load instruction is decoded and register renaming creates a new name *TAG2* for the destination register *R2*. In parallel, via the use of cloaking, the load locates the synonym and hence determines 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 *I1* 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) *I4* 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*, *I4* can link directly to *I1* and execute speculatively as soon as *I4* produces its value. Later on, after the load has accessed the memory the integrity of the communication can be verified. Note that speculative memory bypassing naturally extends for dependence chains that include more than one memory dependence; whenever a store detects that its source register has a speculative name, it can optimistically pass it via the synonym.

6 Transient Value Cache

As we have seen in Section 3, a large fraction of loads get

3. We ignore sign-extension and type conversion issues. The support required is similar to that required for cloaking. However we disallow bypassing over multiple dependences that involve different data types.

their values from a recent store, and most of the values stored to memory are quickly killed. Motivated by these observations we extend the memory hierarchy by introducing a small storage structure, the *Transient Value Cache* (TVC) and use both true and output dependence status prediction to capture in the TVC that part of the memory space through which recently stored values are communicated or killed. As a result, the verification accesses required by cloaking as well as store values that are quickly killed may not have to reach the rest of the memory hierarchy. One may wonder what are the advantages of doing so since the data cache being much larger in size will most likely capture this part of the memory space. However, as ILP processors attempt to execute an ever increasing number of instructions per cycle, as data cache sizes increase, and as wire lengths become a concern, read and write ports to the data cache become an extremely precious and expensive resource [19, 21]. Using a separate, small structure (with separate read/write ports) to service a significant percentage of the loads and stores, not only may lead to reduced data cache read/write port and bandwidth requirements, but may also facilitate shorter access latencies.

In Figure 5 we show how dependence status prediction is used to steer loads and stores. Stores that are likely to be killed soon are initially sent only to the TVC in hope that they will be killed in it before they are forced to go the data cache (part (a)). Other stores are sent to both caches to keep them coherent (part (b)). Loads that are likely to have true dependences with recent stores are initially sent only to the TVC. Such a load is directed to the data cache only if we miss in the TVC (part (c)). In the latter case we *do not* bring the data in the TVC since most likely the dependence status prediction was wrong. Other loads have to access both the TVC and the data cache in parallel (part (d)) since the most recent value may be only in the TVC. Finally, if a dirty block in the TVC needs to be replaced, its contents will have to be written to the data cache.

Similar to true dependence status prediction, output dependence status prediction can be based on the history of previous dependences. However, to do so it is necessary to detect output dependences. The dependence detection table we described in Section 4.1 can provide this functionality by simply reporting the PC of the store recorded in an entry whenever the latter is overwritten.

Finally, note that in a shared memory multiprocessor environment and subject to the consistency model in use, we may have to expose all memory operations to the coherence mechanism. This can be done for example by allowing the TVC to be turned off at the discretion of the operating system or of the program.

7 Experimental Evaluation

In this section, we present experimental evidence in support of the utility of the techniques we propose. The rest of this section is organized as follows: in Section 7.1 we describe our methodology. In Section 7.2, (1) we attempt to gain some insight on the nature of the dependences that are experienced and (2) we demonstrate that speculative memory cloaking can capture a large fraction of the dynamic memory dependences. To do so, we first demonstrate that simple predictors can identify the true dependence status of loads and stores with high accuracy. Then, we look at how complex the actual dependence determination and communication mechanism has to be. We first measure the distribution of the degree of use of store values (i.e., the number of loads that use the value) to determine the number of synonyms a store would have to generate if each dependence was treated separately (similarly to tokens in dataflow processors). Based on this result we continue to consider cloaking schemes in which stores create a single synonym. In this case it is up to the loads to determine the synonym, i.e., to predict the exact dependence. For this reason, we continue by measuring the dependence prediction accuracy that is possible as a function of the number of the dependences that can be remembered per load, i.e., as a function of the *dependence history depth*. We observe that a large fraction of loads experience many different dependences during execution that have to be tracked simultaneously. Based on this observation we continue to evaluate a simple cloaking mechanism in which all dependences that have a common store or load are assigned a common tag as explained in Section 4.1. We then show that this cloaking scheme can communicate correctly a large fraction of the dynamic dependences. In Section 7.3, we first evaluate an output dependence status predictor and then we report a lower bound on the reduction in the number of data cache accesses that can be expected by a 256-entry, fully associative TVC. We conclude the evaluation by measuring the potential performance impact of the proposed techniques. We assume perfect dependence and dependence status prediction over the 256 most recent stores and we show that a cloaking mechanism coupled with a 256 entry fully associative TVC not only improves performance but may also outperform a traditional memory system that has twice as much data cache.

7.1 Methodology

All experiments were performed using the integer programs of the SPEC'95 benchmark suite which were compiled for the MIPS-I architecture [11] by the 2.7.2 version of the GNU GCC compiler (-O3 plus loop unrolling and function inlining). In order to keep the simulation time within reasonable limits we used either the *train* or the *test* input data sets. We used the train data set for *099.go, 132.ijpeg, 134.perl (*jumble*),* and *147.vortex,* whereas we used the test data set for *124.m88ksim, 126.gcc,* and *130.li*. Finally, to obtain a reasonable execution sample for *129.compress* we increased the train input set from 10K characters to 50K. Table 1 reports the resulting dynamic instruction counts and the percentage of loads and stores for the programs used. To simplify the graphs that are subsequently presented we identify the benchmarks using only the first three digits of their name.

Benchmark	***Total***	***Loads***	***Stores***
099.go	553 M	21.3 %	7.9 %
124.m88ksim	458 M	18.9 %	9.5 %
126.gcc	1.49 G	23.4 %	19.4 %
129.compress	150 M	21.7 %	13.5 %
130.li	977 M	29.6 %	17.5 %
132.ijpeg	1.48 G	17.6 %	8.4 %
134.perl	2.21 G	25.5 %	16.4 %
147.vortex	2.82 G	28.7 %	24.7 %

Table 1. *Dynamic instruction count and percentage of load and store instructions per benchmark*

To evaluate cloaking and to provide an estimate on the processor/data cache reduction we can expect with a TVC, we first employ trace based simulation. The memory access traces are generated via the use of a functional simulator and include all but

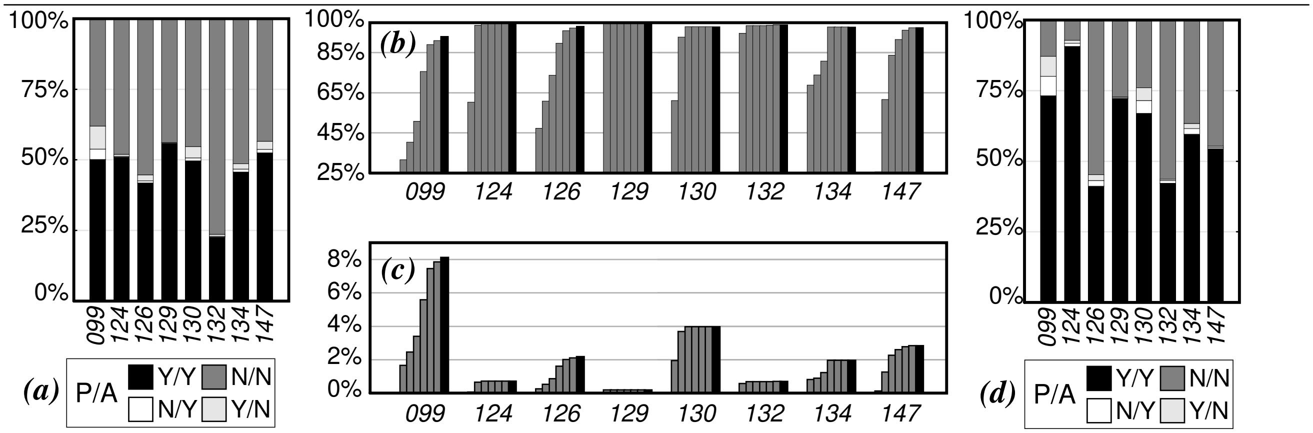

Figure 6. *Predicting the true dependence status of loads and stores within a store window of 256. Parts (a) through (c) are for loads whereas part (d) is for stores.* ***Loads:*** *Infinite prediction resources: (a) prediction breakdown ("P/A" stands for "Predicted/Actual"). Finite prediction resources: (b) Accuracy of prediction for loads with true dependences, (c) Percentage of loads that are incorrectly identified as having true dependences. Results of parts (b) and (c) are as a function of table size. Samples are taken at powers of two and the range is 64 to 2K entries (last bar is with infinite resources).* ***Stores:*** *(d) Prediction breakdown with infinite resources.*

system code data references. (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 a traditional 5 stage (fetch/decode/execute/access/writeback) pipeline with out-of-order execution characteristics. Up to 64 instructions can be in-flight at any given point of time. Functional units are fully pipelined and have a latency of 1 cycle except for multiplication and division which take 4 and 12 cycles respectively. For control prediction purposes we use a GSHARE predictor [13] with 64K 2-bit saturating counters. The base memory system comprises four read/write ports, a store queue with 16 entries, a non-blocking 32 kilobyte/16 byte block/8-way interleaved/2-way set associative data cache with an access latency of 2 cycles and a miss latency of either 16 or 24 cycles (depending on the configuration simulated) for the first word plus 1 cycle for each additional word, and finally an instruction cache with the same characteristics (contention in the memory bus shared by the two caches is also modeled). We also assume perfect memory disambiguation for all configurations in order not to give an unfair advantage to the configuration which uses cloaking (cloaking may also be used to schedule unresolved dependences similarly to what was done in [15]).

For the configurations that use cloaking and a TVC we assume perfect dependence and dependence status prediction within the 256 most recent stores[4]. However, a dependence is not predicted unless it has been seen at least once. We also assume a 256 entry fully-associative TVC. Furthermore, for each benchmark we simulate 100 million instructions after we have skipped the first 100 million instructions.

7.2 Accuracy of Speculative Memory Cloaking

In this section we demonstrate that: (i) relatively simple predictors can be used to identify the dependence status of loads and stores, (ii) a significant fraction of stores would have to generate multiple synonyms if each dynamic dependence was treated separately, (iii) predicting and tracking more that one dependence per static load is important, and (vi) a simple cloaking scheme can capture and communicate a large fraction of the dynamic dependences for most programs.

7.2.1 True Dependence Status Prediction

We evaluate predictors that associate a saturating counter with each relevant static load or store instruction via the instruction address. We experimented with various counter based predictors, and here we report the results for the those that performed best: 2-bit counter with threshold of 1 for loads and 1-bit counter for stores (i.e., last status seen). To isolate problems with finite storage we first evaluate infinite structures. However, to demonstrate the feasibility of the mechanisms we also evaluate finite prediction structures of various sizes with LRU replacement policy. In the discussion that follows we are first concerned with predicting the dependence status of loads. Later we consider stores also.

In Figure 6, part (a) we report the breakdown of the dynamic predictions for a true dependence predictor with infinite prediction entries. Since dependence prediction is a binary decision there are four possible outcomes: (a) we may correctly predict that a load has a true dependence (category *Y/Y*), (b) we may fail to predict that a load has a dependence (category *N/Y*), (c) we may incorrectly predict that a load has a true dependence whereas it does not (category *Y/N*) and finally, (d) we may correctly predict that a load has no dependence (category *N/N*). The loads of category *Y/Y* are candidates for cloaking and speculative memory bypassing. Loads in the second category *(N/Y)* could potentially benefit from either of the proposed techniques, however in practice will fail to do so. Loads in *Y/N* may be incorrectly used to perform cloaking and bypassing if a matching synonym is found. Finally, the loads of category *N/N* will neither benefit nor get penalized by any of the proposed techniques.

In parts (b) and (c) of Figure 6, we show the effects of finite prediction storage. We report results for prediction tables of entry counts that are powers of 2 in the range of 64 to 2K. In part (b), we report the *true dependence prediction accuracy*, which is defined as the percentage of the correctly predicted dynamic

4. Note that these predictors are pessimistic models of perfect predictors which use dependence information that is collected via a 256 fully-associative DDT since, for example, some of the 256 most recent stores may be writing to the same memory location. In this case the described DDT may capture dependences whose store distance is more than 256.

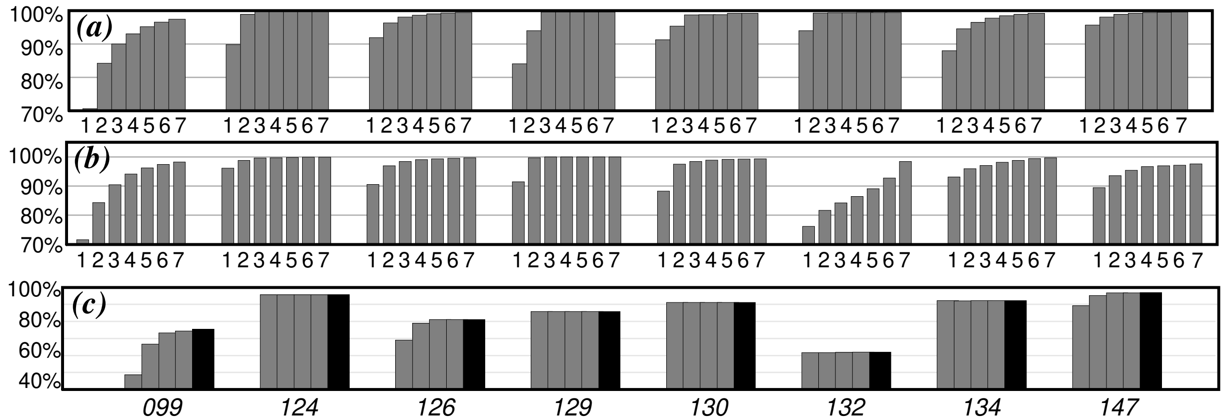

Figure 7. *(a). Cumulative distribution of the degree of use of store values. (b). Percentage of true dependences that can be predicted as a function of the dependence history depth kept per static load. (c). Percentage of true dependences communicated correctly via cloaking. Dark bar is for infinite DPNT, gray bars are for 512, 1K, 2K and 4K entries.*

loads with true dependences over the total number of the dynamic loads that have true dependences (i.e., *Y/Y* over *Y/Y + N/Y*). It can be seen that for all benchmarks, there is virtually no difference between a 2K prediction table and an infinite one (the dark bar shows the results with infinite entries). Furthermore, there is little or no difference between the predictors with 512 and 2K entries. In part (c), we report the percentage of the dynamic loads that are incorrectly identified as having true dependences (category *Y/N)*. It can be seen that only few loads are incorrectly identified.

Part (d) of Figure 6 reports the prediction breakdown for an infinite true dependence status predictor for stores. We omit the results for finite prediction tables by noting that no significant degradation in accuracy or no difference at all is observed for a 512 entry prediction table.

7.2.2 A Speculative Memory Cloaking Mechanism

The cumulative distribution of the degree of use of store values is shown in part (a) of Figure 7 up to a degree of use of 7 (percentage is over all stores whose value is used). It can be seen that a significant fraction of store values are used at least twice. Consequently, if we were to provide a different synonym per dependence we often would have to generate two or more synonyms. Based on this result and for the purposes of this evaluation we restrict our attention to cloaking schemes in which a store instance uses a common synonym to communicate with all loads that may be dependent on it. In part (b) or Figure 7, we report the number of dynamic dependences that can be potentially predicted as a function of the number of dependences we can remember per static load. It can be seen that a significant fraction of dependences cannot be predicted unless we record two or more of the most recent dependences per static load.

Based on the insight gained from the previous two experiments we evaluate a cloaking mechanism that associates a common tag to all dependences that have a common store or a load as these are detected (we also evaluated a scheme that records and predicts the most recent store per load and found it to be inferior). As explained in Section 4.1, this scheme attempts to provide support for loads that experience dependences with more than one store. For all experiments, we assume a 256, fully associative SF (no notable difference was observed with a direct mapped SF). In part (c) of Figure 7 we report the percentage of the dynamic true dependences whose value is correctly communicated through speculative memory cloaking (note that the number of true dependences and the number of the loads with a true dependence is considered to be the same for the purpose of this evaluation) when an infinite DPNT is used (dark bar) and when the number of entries is restricted to 512, 1K, 2K and 4K (gray bars). It can be seen that the majority of all dynamic dependences is correctly communicated. An investigation of the relative importance of each potential source of failures is beyond the scope of the paper. (The percentage of all loads that get their value from cloaking can be derived by multiplying the Y/Y prediction accuracies from part (a) of Figure 6 and the cloaking accuracies reported).

7.3 Reduction of Data Cache Accesses

In Figure 8, part (a) we report the breakdown of the dynamic predictions of an output dependence predictor with infinite entries. Again there are four possible outcomes based on the predicted and the actual dependence status of a store. Overall, the dependence status of the majority of the dynamic stores is correctly predicted. We do not present the results for finite prediction tables since the trends are similar to those observed in true dependence status prediction. However we note that virtually no difference was observed when the number of entries was restricted to 512. Finally, in part (b) of Figure 8, we report a lower bound on the reduction of data cache accesses that can be expected by a 256-entry fully associative TVC given the true and output dependence status predictors we simulated of 512 entries each. We report a lower bound on the percentage of dynamic loads and stores that would hit or get killed, respectively, in this TVC (i.e., these accesses will not reach the data cache).

7.4 Potential Impact on Performance

In this section we attempt to get an estimate on the performance impact the proposed mechanisms may have. The base case used in these experiments is an ILP processor with a traditional memory system with a 32K data cache. In Figure 9, we report the speedups observed when our mechanisms (as described in Section 7.1) is used or when the data cache size is doubled to 64K. We present two sets of measurement, one with for miss latency of 16 cycles and one for miss latency of 24 cycles. It can be seen that the proposed mechanisms have the potential to improve performance even when compared to the system that has twice as much data cache.

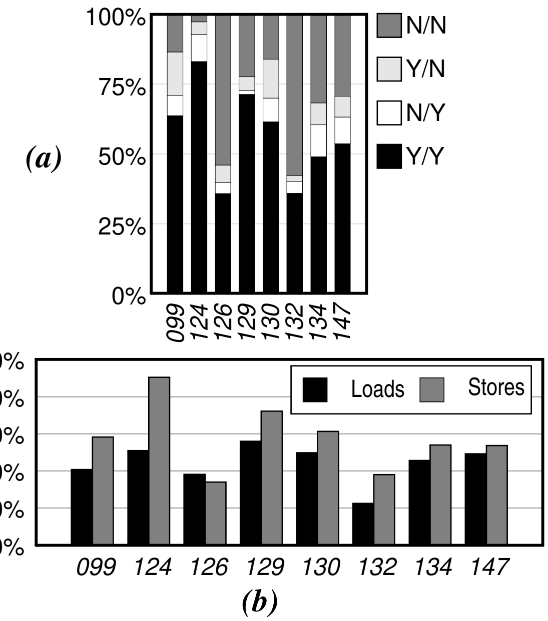

Figure 8. *(a) Output dependence prediction breakdown given infinite prediction resources. (b) Lower bound on the percentage of all load and store instructions that will either hit or will be killed in a 256-entry, fully associative TVC. We assume counter based true and output dependence status prediction tables, each of 512-entries.*

8 Related Work

Dependence prediction has been introduced in [15] where it was used to improve the accuracy of dependence speculation. However, to the best of our knowledge, no previous work exists in attempting to dynamically establish direct links between dependent instructions for the purposes of streamlining the inter-operation communication; nor has there been previous work in using dependence status prediction to manage the storage within a traditional memory hierarchy.

Numerous techniques that attempt to predict the data addresses of loads and stores have been proposed and used to reduce the access latency of loads both in hardware and in software [e.g., 1,2,3,4,6,18]. Even though no attempt is made to establish explicit links between dependent instructions, these techniques may, as a side effect, reduce the latency of the communication of load-store dependences, provided that the data address accessed by the load is correctly predicted and that the store has executed (i.e., both the data address and value are available). Cloaking may streamline the communication even if the access pattern defies prediction.

In this work we were motivated by the large fraction of memory accesses that correspond to dependences with a recent store and by the fraction of memory values that are killed soon after they are created. A number of studies have also looked at the memory referencing behavior of programs for the purpose of optimizing the memory hierarchy. McNiven and Davidson [14] 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. They also formalized efficient memory systems [8,9]. The TVC brings a traditional memory hierarchy closer to the idealized efficient memory hierarchy since many of the values that are quickly killed are handled by a small storage without ever getting exposed to the rest of the hierarchy.

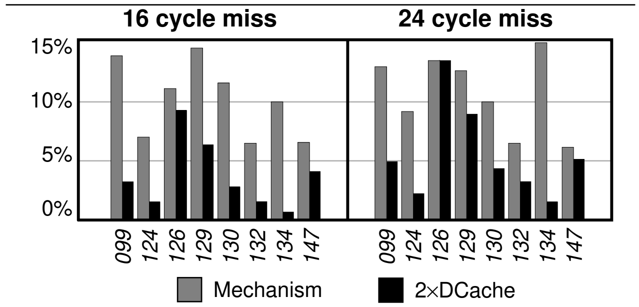

Figure 9. *Potential performance impact of our mechanisms. Speedups are relative to a base machine that has a 32K data cache.* "Mechanism" *uses in addition all the three techniques we propose.* 2×DCache *is the base system with 64K data cache. The configurations are described in detail in Section 7.1.*

Methods for making cache management decisions based on access characteristics have also been suggested. Tyson, Farrens, Matthews and Pleszkun have used the miss behavior of loads to selectively bypass the data cache [20]. Rivers and Davidson [17], and Johnson and Hwu [10] used the reuse behavior of memory addresses to make cache management decisions. González, Aliagas and Valero use a dynamic scheme to determine the spatial and temporal locality characteristics of memory addresses and use it to manage two separate caches [7]. Finally, Pomerene, Puzak, Rechtschaffen and Sparacio [16] propose the shadow cache as an improvement over LRU replacement. The TVC approach is orthogonal to all aforementioned cache management methods and differs in that dependence status prediction is used to redirect accesses and not the characteristics of the actual data address or the miss behavior of the instruction.

Wilson, Olukotun and Rosenblum suggested the use of a Line Buffer to cache recently accessed data in order to reduce the processor–data cache bandwidth and read/port requirements [21]. The Line Buffer is placed in front of the data cache, and all accesses have to go through it (for this reason its size is limited by timing considerations). Furthermore, in contrast to the TVC, all loads accesses cause the corresponding data to be cached into the Line Buffer.

Finally, value speculation may effectively reduce the latency of memory communication independently of whether the load has a true dependence or not [12]. 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.

9 Summary and Conclusions

We revisit memory communication and consider techniques that use dynamically collected information to alleviate the drawbacks associated with the traditional way of expressing and performing memory communication. In doing so, we make the following contributions:

(1) We show that the data dependence status of most memory operations can be predicted with high accuracy on a per instruction basis and based solely on the history of previous data dependences.

(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 the explicit specification of memory communication to take the address calculation, the disambiguation, the data cache access and whenever possible, the load and store instructions themselves off the communication path.

(4) We propose the Transient Value Cache a dependence status prediction managed storage structure that can reduce the contention for data cache resources.

We demonstrated that a large percentage of the inter-operation memory communication can be streamlined via cloaking and hidden via the TVC. Furthermore, we showed that a large percentage of the store values that are quickly killed can also be hidden by a TVC.

Several directions for further research exist. Although effective, the implementations we proposed are preliminary. Accordingly, further investigation may help in: (i) improving accuracy and performance of the proposed mechanisms, (ii) developing better implementations and (ii) determining the relative importance of each of the mechanisms. To improve the accuracy of the dependence prediction more sophisticated predictors may be sought. The relative importance of the mechanisms is expected to vary as the assumptions about the processor and the memory system change. Further investigation is required to determine how the proposed mechanisms perform as instruction windows and memory latencies increase or as other speculation methods are incorporated.

A more exciting research direction however is to study the impact a communication centric approach may have on memory hierarchy design and management. The methods we propose make a first step toward this direction. More general schemes may be possible. For example we may attempt to collect and predict various kinds of communication attributes and to associate this information with the corresponding loads and stores. This information may be useful in making memory hierarchy management decisions. It may also be utilized to develop novel storage structures that are optimized toward different communication patterns. What kinds of information might be useful in this context, whether they can be collected and predicted dynamically or statically, whether it is best to associate them with dependences rather than names, and how they might be used, are open questions. As a starting point we may consider a number of communication characteristics (e.g. lifetime, inter-reference times, how important a value is in terms of performance) and study how these characteristics vary from the point of view of the dependences and of the memory names.

Acknowledgements

We are thankful to Scott Breach, Andy Glew, Babak Falsafi, Stefanos Kaxiras, Amir Roth, Avinash Sodani and T. N. Vijaykumar for helpful discussions and for their valuable comments. We also thank Mark Hill for comments on an earlier version of this paper. Scott Breach also provided the simulators used in this study. He, and T. N. Vijaykumar have patiently read and commented on several versions of this text.

This work was supported in part by NSF Grant MIP-9505853, by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346, by a donation from Intel and an equipment donation from Sun Microsystems. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data access with 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*, pages 176–186, 1991.

[4] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. In *IBM journal on research and development,* pages 37(4):547–564, July 1993.

[5] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[6] M. Golden and T. Mudge. Hardware support for hiding cache latency. *CSE-TR-152-93, University of Michigan, Dept. Of Electrical Engineering and Computer Science*, Feb. 1991.

[7] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tunned to different types of locality. In *Proc. of the 1995 Conference on Supercomputing*, June 1995.

[8] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proc. ASPLOS-VII*, Oct. 1996.

[9] A. S. Huang and J. P. Shen. A limit study of local memory requirements using value profiles. In *Proc. MICRO-28*, Dec. 1996.

[10] T. L. Johnson and W.-M. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. ISCA-24*, June 1997.

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

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

[13] S. McFarling. Combining branch predictors. *DEC WRL Technical Report TN-36*, June 1993.

[14] G. D. McNiven and E. S. Davidson. Analysis of memory referencing behavior for design of local memories. In *Proc. ISCA-15*, May 1988.

[15] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In Proc. *ISCA-24*, June 1997.

[16] J. Pomerene, T. R. Puzak, R. Rechtschaffen, and F. Sparacio. *Prefetching Mechanism for a high-speed buffer store*. Referred to as "Patent Pending 1984", in *High Performance Computer Architecture*, H. S. Stone, 3rd Edition, Addison Wesley, 1993.

[17] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proc. of the 1996 Conference on Parallel Processing*, June 1995.

[18] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. In *Proc. MICRO-29*, Dec. 1997.

[19] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proc. ASPLOS-IV*, Apr. 1991.

[20] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proc. MICRO-28*, Dec. 1996.

[21] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Proc. ISCA-23*, May 1996.