CHAPTER 15
Exploiting Load/Store Parallelism via Memory Dependence Prediction

Since memory reads or loads are very frequent, memory latency, that is the time it takes for memory to respond to requests can impact performance significantly. Today, reading data from main memory requires more than 100 processor cycles while in 'typical’ programs about one in five instructions reads from memory. A naively built multi-GHz processor that executes instructions in-order would thus have to spent most of its time simply waiting for memory to respond to requests. The overall performance of such a processor would not be noticeably better than that of a processor that operated with a much slower clock (in the order of a few hundred MHz). Clearly, increasing processor speeds alone without at the same time finding a way to make memories respond faster makes no sense. The memory latency problem can be attacked directly by reducing the time it takes memory to respond to read requests. Because it is practically impossible to build a large, fast and cost effective memory, it is impossible to make memory respond fast to all requests. However, it is possible to make memory respond faster to some requests. The more requests it can process faster, the higher the overall performance. This is the goal of traditional memory hierarchies where a collection of faster but smaller memory devices, commonly referred to as caches, is used to provide faster access to a dynamically changing subset of memory data. Given the limited size of caches and imperfections in the caching policies, memory hierarchies provide only a partial solution to the memory latency problem. Figure 0.1 parts (a) through (c) illustrate how memory latency impacts performance and how reducing it as do caches reduces execution time.

An alternative, yet orthogonal way of attacking the memory latency problem seeks to tolerate memory latency. The goal here is to send loads to memory earlier, as far in advance from the instructions that need the memory data in effect overlapping the memory request with other useful work. With this technique, memory still takes the same amount of time to respond, but because the data is needed later, to the consum-
ing instructions it appears that memory is faster (i.e., they have to wait for less time or not at all). Sending loads to memory as early as possible requires moving loads up in the execution order, placing them in a position that might be different than the one implied by the program. Figure 0.1 part (d) shows an example of this technique. Because memory writes (stores) are also fairly frequent (about one in ten instructions is a store) moving loads up the execution order includes the ability to move them before otherwise preceding stores. To do so we have to be able extract and exploit load/store parallelism and to execute instructions out-of-order. Techniques for exploiting instruction level parallelism have been developed since the 60’s. However, these techniques work only for instructions that access registers and not memory. Extracting load/store parallelism is much harder since the memory locations accessed are not fixed at compile time as are the register accesses per instruction. While an instruction always accesses the same registers every time it is executed, loads and stores may access a different address every time they are executed since the address calculation is done at run time.

In this chapter we review previously proposed static and dynamic methods of executing memory operations out-of-order, and demonstrate that higher performance is possible if memory dependence speculation is used. In memory dependence speculation, a load may execute before a preceding store on which it may be data dependent (i.e., the store may be writing the data needed by the load). We discuss the trade-offs involved in using memory dependence speculation and explain that care must be taken to balance the benefits of correct speculation against the net penalty incurred by erroneous speculation. We demonstrate that as dynamically-scheduled ILP processors are able to schedule instructions over larger regions, the net performance loss of erroneous memory dependence speculation (misspeculation) can become significant.

The rest of this chapter is organized as follows: in Section 15.1 we motivate the need for exploiting load/store parallelism and discuss the challenges raised by ambiguous (i.e., temporarily unknown) memory dependences. We briefly discuss static and hybrid methods for exploiting load/store parallelism in sections 15.1.1 and 15.1.2 respectively. We explain the difficulties raised in exploiting load/store parallelism dynamically in section 15.1.3. We use this discussion to motivate memory dependence speculation which we discuss in Section 15.2. Here we review how memory dependence speculation is being used today and provide qualitative arguments on why, techniques to improve the accuracy of memory dependence speculation might
be useful. In Section 15.3 we discuss a number of memory dependence speculation policies and argue for memory dependence speculation and synchronization (speculation/synchronization for sort), a policy that aims at mimicking what is ideally possible. In Section 15.4, we discuss the requirements of speculation/synchronization. An implementation framework for our proposed technique we present in Section 15.5. We review related work in Section 15.6. We provide experimental evidence in support of the utility of our proposed technique and of our observations in section 15.7. Finally, we summarize our findings in Section 15.8.

15.1 Using Load/Store Parallelism To Improve Performance

The most wide-spread programming model in use today assumes sequential execution semantics. In sequential execution programs are written with an implied, total order where instructions are meant to execute one after the other and in the order specified by the program. Even parallel programs assume sequential execution semantics within individual threads. However, closer inspection of sequential execution semantics reveals that there are many execution orders that produce exactly the same results. Key in determining whether a particular execution order will produce valid results is the discovery of dependences (see [33] for detailed explanation of dependence types). Of particular importance are true (read-after-write or RAW) dependences. A true dependence exists between two instructions A and B, if B reads the data produced by A. Whenever, a true dependence exists between two instructions they must execute in the implied order, otherwise they can execute in any order, possibly in parallel. This ability is also useful in tolerating slower memory devices by overlapping the processing of load requests with other useful computation. Moving loads as far ahead of the instructions that read their data, and in general exploiting instruction-level parallelism can be done statically, dynamically or via a combination of both static and dynamic methods.

15.1.1 Static Methods

The first class of methods for exploiting load/store parallelism relies on static rearrangement or scheduling of memory accessing instructions during compile time. The key difficulty here is knowing whether it is safe to move a load earlier in the schedule and specifically before, an initially preceding store. A load should not be scheduled before a store if it is going to access the same address. At the core of all software-based load scheduling techniques are static disambiguation or alias analysis techniques. The goal of these methods is to determine whether a given load and store will be data independent during execution. Many techniques have been proposed. Initially research focused primarily on array variables [4,22,10], while recently methods have been proposed for dynamically allocated data types [23, 97]. With these methods, a load would be scheduled before an initially preceding store only if it can be proven that they two instructions will always be independent. Pointers and the inconvenience of multipass compiling (alias analysis across functions declared in separate files requires whole program analysis) are the two most important limitations of such static methods.

It was observed that while many loads are independent of preceding stores it was not possible to prove (with existing methods) this independence during compile time. To allow such loads to execute early Nicolau proposed run-time disambiguation [65], a software only approach to memory dependence speculation. In his technique, loads can be speculatively scheduled before a preceding store without knowing for sure that no dependence exists. Code is inserted after the store to detect whether a true dependence is violated (this is done by comparing the addresses accessed by the store and the load), and repair code is also inserted to

1. Strictly speaking, program semantics are maintained so long as instructions read the same value as they would in the original program implied order. This does not necessarily imply that a dependent pair of instructions executes in the program implied order. We avoid making this distinction in the discussion of this chapter for clarity.
recover from memory dependence violations. Another software only approach was proposed by Moudgill and Moreno [64]. Their approach differs from Nicolau’s in that they compare values rather than addresses to detect violation of program semantics. Code bloat and the execution overhead of checking and repairing in case of a violation are important considerations with these techniques. Figure 15.2 shows an example of run-time disambiguation.

15.1.2 Hybrid Static/Dynamic Methods

In purely static speculation, any benefits gained by scheduling loads early had to be balanced against the execution overhead of checking for memory dependence violations. A major advantage of static speculation techniques however is that they require no hardware support. The next class of memory dependence speculation techniques introduces minimal additional hardware in order to reduce the checking code overhead. Specifically, Gallagher, Chen, Mahlke, Gyllenhaal and Hwu [28, 16] proposed the Memory Conflict Buffer (MCB), a software-hardware hybrid approach. In their technique, load motion and mispeculation recovery are done in software while mispeculation detection is done in hardware. Two copies of each speculated load are executed, one at the original program order (non-speculative) and the other as early as desired (speculative). Speculative loads record their addresses in the MCB. Intervening stores also post their addresses to the MCB, so that dependence violations can be detected. The non-speculative load checks the appropriate entry in the MCB (the target register of the load is used as a handle), and if any dependence was violated, control is transferred to recovery code. Figure 15.3 illustrates how the MCB method works. MCB-like solutions have been implemented in Transmeta’s Crusoe processors [18] and in the Intel/HP EPIC architecture [76]. Huang, Slavenburg and Shen proposed speculative disambiguation [36] another hybrid approach to memory dependence speculation. In their technique multiple versions of the same code are generated, one with speculation enabled and another with speculation disabled. These versions are then scheduled together using predication. Hardware similar to that used for boosting [78, 77] is used to invalidate all but the appropriate path during execution.

15.1.3 Dynamic Methods

Typical modern dynamically-scheduled ILP processors, exploit instruction-level parallelism by forging ahead into the execution stream, building an instruction window, a set of instruction to execute. These processors, then attempt to convert the total, program implied order within this set into a partial order. The shape of the partial order and for that the performance improvements so obtained are heavily influenced by the processor’s ability to uncover the true data dependences among the instructions currently under consid-
In the case of loads, the performance improvements obtained are determined by the processor’s ability to send load requests to memory as early as possible without, however, allowing a load to access memory before a preceding store with which a true data dependence exists. One way of doing so is to first determine the true dependences a load has and then use that information to schedule its execution. With this approach, we ensure that no true dependences are violated in the resulting execution order. In the case of loads and stores the process of determining the data dependences they have is commonly referred to as disambiguation.

Determining the data dependences among the instructions in the instruction window requires inspection of the addresses they access. Unfortunately, these addresses are not necessarily available immediately. This is typical for stores and loads which have to perform an address calculation. As a result, at any point during execution, memory dependences may be unambiguous (i.e., a load consumes a value that is known to be created by a store preceding it in the total order) or ambiguous (i.e., a load consumes a value that may be produced by a store preceding it in the total order). During execution, an ambiguous dependence gets eventually resolved to either a true dependence, or to no dependence. We will use the term false dependence to refer to an ambiguous dependence that eventually gets resolved to no dependence. As we explain next, false dependences present a challenge to the out-of-order execution of load instructions.

Ambiguous memory dependences may obscure some of the parallelism that is present. To maintain program semantics a load has to wait for a store with which an ambiguous dependence exist only if a dependence really exists. If the ambiguous dependence is a false dependence, any execution order is permissible, including ones that allow the load to execute before the store. This latter case, represents an opportunity for parallelism and for higher performance. Unfortunately, the mere classification of a dependence as ambiguous implies the inability to determine whether a true dependence exists without actually waiting for the addresses accessed by both instructions to be calculated. Worse, in the absence of any explicit memory dependence information (the common case today), a dynamically scheduled ILP processor has to assume that ambiguous dependences exist among a load and any preceding store that has yet to calculate its address (provided that no intervening store accesses the same address and has calculated its address).

Figure 15.3: The Memory Conflict Buffer approach to memory dependence speculation. A load is scheduled for execution before an originally preceding store. A special load instruction, load.speculative is used. Upon its execution, and using its target register Rx as a handle an entry is allocated in the memory conflict buffer. The entry records the address accessed by the load.speculative (Addr1 in our example). When the store executes, it posts its address to the memory conflict buffer. If a matching entry is found (that is if Addr1 equals Addr2) a flag is set in the corresponding entry. Later, a check.speculative instruction is executed where one of the arguments is the same register used as target by the preceding load.speculative. The check.speculative instruction is essentially a conditional branch that succeeds if the corresponding memory conflict buffer entry has its flag clear. This means that no store after the load.speculative has accessed the same address and thus the load.speculative did not violate a memory dependence. If the flag is set, then a memory dependence violation occurred and the memory read must be repeated.

\[
\begin{align*}
\text{load.speculative } & \text{ Rx, Mem[Addr2]} \\
& \ldots \\
\text{store } & \text{ Ry, Mem[Addr1]} \\
& \ldots \\
\text{check.speculative } & \text{ Rx, OK} \\
\text{load } & \text{ Rx, Mem[Addr2]} \\
& \text{OK: use Rx} \\
& \ldots
\end{align*}
\]
As we will demonstrate in the evaluation section, significantly higher performance is possible if we could make loads wait only for those ambiguous dependences that get resolved to true dependences. Moreover, we demonstrate that this performance difference widens as the size of the instruction window increases. To expose some of the parallelism that is hindered by ambiguous memory dependences, memory dependences speculation can be used. This technique is the topic of the next section.

15.2 Memory Dependence Speculation

Memory dependence speculation aims at exposing the parallelism that is hindered by ambiguous memory dependences. Under memory dependence speculation, we do not delay executing a load until all its ambiguous dependences are resolved. Instead, we guess whether the load has any true dependences. As a result, a load may be allowed to obtain memory data speculatively before a store on which it is ambiguously dependent executes. Eventually, when the ambiguous dependences of the load get resolved, a decision is made on whether the resulting execution order was valid or not. If no true dependence has been violated, speculation was successful. In this case, performance may have improved as the load executed earlier than it would had it had to wait for its ambiguous dependences to be resolved. However, if a true dependence was violated, the speculation was erroneous (i.e., a mispeculation). In the latter case, the effects of the speculation must be undone. Consequently, some means are required for detecting erroneous speculation and for ensuring correct behavior. Several mechanisms that provide this functionality, in either software and/or hardware, have been proposed [39,26,27,28,37,56,65,71]. The hardware techniques used today work by invalidating and re-executing all instructions following the mispeculated load. We will use the term squash invalidation to refer to this recovery method.

Figure 15.4: Using memory dependence speculation may affect performance either way. (a) Code with an ambiguous memory dependence. Continuous arrows indicate register dependences. Parts (b) through (d) show how this code may execute in a dynamically-scheduled ILP processor capable of executing two instructions per cycle. We assume that due to other dependences, the store may execute only after two cycles have passed. (b) Execution order when no memory dependence speculation is used. (c) Memory dependence speculation is used and the ambiguous dependence gets resolved to no dependence. (d) Memory dependence speculation is used, and the ambiguous dependence gets resolved to a true dependence.

Though memory dependence speculation may improve performance when it is successful, it may as well lead to performance degradation when it is wrong. We demonstrate either possibility with the example of Figure 15.4. The reason is that a penalty is typically incurred on mispeculation. The penalty includes the following three components: (1) the work thrown away to recover from the mispeculation, which in the case of squash invalidation, may include unrelated computations, (2) the time, if any, required to perform the inali-
dation, and finally (3) the opportunity cost associated with not executing some other instructions instead of
the mispeculated load and the instructions that used erroneous data. Consequently, in using memory depend-
ence speculation care must be taken to balance the performance benefits obtained when speculation is cor-
rect against the net penalty incurred by erroneous speculation. To gain the most out of memory dependence
speculation we would like to use it as aggressively as possible while keeping the net cost of mispeculation as
low as possible. Ideally, loads would execute as early as possible while mispeculations would be completely
avoided.

The first dynamically scheduled processors did not use memory dependence speculation because in their
relatively small instruction windows (10-20 instructions) there was often little to be gained from extracting
load/store parallelism. When instruction windows became larger (few tens of instructions) naive memory
dependence speculation was used where a load was executed immediately after its address was calculated.
Naive memory dependence speculation was very appropriate since the probability of a true memory depen-
dence being violated within the relatively small instruction windows was very small. In most cases, naive
memory dependence speculation offers superior performance compared to having to wait until ambiguous
dependences are resolved (i.e., no speculation). Moreover, the benefits of memory dependence speculation
increase as the size of the instruction window also increases. More importantly, further performance
improvements are possible if we could avoid misspeculations. The latter is possible via more accurate mem-
ory dependence speculation methods. We present such methods in the next chapter.

15.3 Memory Dependence Speculation Policies

The ideal memory dependence speculation mechanism not only avoids misspeculations completely, but
also allows loads to execute as early as possible. That is, loads with no true dependences (within the instruc-
tion window) execute without delay, while loads that have true dependences are allowed to execute only
after the store (or the stores) that produces the necessary data has executed. It is implied that the ideal mem-
ory dependence speculation mechanism has perfect knowledge of all the relevant memory dependences.

An example of how the ideal memory dependence speculation mechanism affects execution is shown in
Figure 15.5. In part (b), we show how the code sequence of part (a) may execute under ideal memory depen-
dence speculation and in part (c) we show how the execution may progress under naive memory dependence
speculation. The example code sequence includes two store instructions, ST-1 and ST-2, that are followed by
two load instructions, LD-1 and LD-2. Ambiguous dependences exist among these four instructions as indi-
cated by the dotted arrows. During execution, however, only the dependence between ST-1 and LD-1 is
resolved to a true dependence (as indicated by the continuous arrow). Under ideal dependence speculation,
LD-2 is executed without delay, while LD-1 is forced to synchronize with ST-1.

In contrast to what is ideally possible, in a real implementation, the relevant memory dependences are
often unknown. Therefore, if we are to mimic the ideal memory dependence speculation mechanism, we
have to attempt: (1) to predict whether the immediate execution of a load is likely to violate a true memory
dependence, and if so, (2) to predict the store (or stores) the load depends upon, and, (3) to enforce synchro-
nization between the dependent instructions.

However, since this scheme seems elaborate, it is only natural to attempt to simplify it. One possible sim-
plication is to use selective memory dependence speculation, i.e., carry out only the first part of the ideal
tree-part operation. In this scheme the loads that are likely to cause mispeculation are not speculated.
Instead, they wait until all their ambiguous dependences are resolved; explicit synchronization is not per-
formed. We use the term selective memory dependence speculation (or selective speculation for short) to sig-
nify that we make a decision on whether a load should be speculated or not. In contrast, in ideal dependence
speculation, we make a decision on when is the right time to speculate a load. While selective memory
dependence speculation may avoid mis-speculations, due to the lack of explicit synchronization, this prediction policy may as well make loads wait longer than they should and for this reason may negatively impact performance. This case we illustrate with the example shown in part (d) of Figure 15.5. In this example, \( LD-2 \) is speculated, whereas \( LD-1 \) is not, since prediction correctly indicates that \( LD-2 \) has no true dependences while \( LD-1 \) does. However, as shown \( LD-1 \) is delayed more than necessary as it has to wait not only for \( ST-1 \) but also for \( ST-2 \). In practice, selective data dependence speculation can lead to inferior performance when compared to naive speculation (part (c) of Figure 15.5) even when perfect prediction of dependences is assumed. While this policy avoids mis-speculations it often fails to delay loads only as long as it is necessary.

Another possible simplification, the store barrier has been proposed by Hesson, LeBlanc and Ciavaglia [33,3]. In this technique a prediction is made on whether a store has a true dependence that would normally get mis-speculated. If it does, all loads following the store in question are made to wait until the store has posted its address for disambiguation purposes. While the store barrier policy can be successful in (1) eliminating mis-speculations, and (2) delaying loads that should wait only as long as it is necessary, it may as well lead to inferior performance since it may unnecessarily delay other unrelated loads that have no true dependences that can be mis-speculated. While, in the example of Figure 15.5, the store barrier policy is shown to perform better than selective speculation, the opposite can also be true (for example, if other loads, following \( LD-1 \) existed they would too get delayed under the store barrier policy, while they wouldn’t under the selective policy). The store barrier method has been shown to perform worse than other memory dependence speculation methods we present in the next section [17]. However, it is an attractive alternative due to its simplicity.
In the next section, we present dynamic memory dependence speculation/synchronization, a technique that utilizes memory dependence prediction to identify those store-load pairs that ought to be synchronized in order to avoid memory dependence violations while delaying load execution only as long as it is necessary.

15.4 Mimicking Ideal Memory Dependence Speculation

To mimic the ideal data dependence speculation system, we need to implement all the three components of the ideal system as described in the previous section. That is, we must: (1) dynamically identify the store-load pairs that are likely to be dependent and whose normal execution will result in a memory dependence violation, (2) assign a synchronization mechanism to dynamic instances of these dependences, and (3) use this mechanism to synchronize the store and the load instructions.

To identify the store-load pairs that need to be synchronized we may use history-based memory dependence prediction. With this scheme, naive memory dependence speculation is initially used for all loads. A load is initially allowed to execute as soon as its address is calculated and memory resources are available. With this policy, as execution progresses misspeculations will be encountered. Instead of discarding the information available when a misspeculation occurs (as we would under naive memory dependence speculation), we collect information about the instructions involved. For example, we may record the static dependence that was violated, that is a (store PC, load PC) pair. The next time a load or a store that has previously incurred a mispeculation is encountered, we can use the recorded information to predict whether synchronization has to take place in order to avoid a mispeculation. In the predictors we present mispeculation history is associated with the static loads and stores using their PC. For history-based memory dependence prediction to be possible, it is imperative that past mispeculation behavior to be indicative of future dependences that ought to be synchronized. Fortunately, this is true or typical programs. This can be seen in figure 15.6 where we report the dependence set locality for loads and stores in parts (a) and (b) respectively (see section 15.7.1 for our methodology). We define “dependence set locality (n)” of a load instruction to be the probability that once it experiences a memory dependence then this dependence is within the last n unique dependences it experienced. Dependence set locality is defined similarly for stores. A high dependence set locality for small values of n suggests that loads experience repeatedly the same dependences. In turn this is direct indication that past dependence behavior is indeed strongly correlated to future dependence behavior and thus history-based prediction should be possible. We report locality measurements in the range of 1 to 4 and as a fraction over all executed loads or stores that experience a RAW dependence. Memory dependence locality is very strong. In most programs, nearly 80% or more of all loads with RAW dependences experience the same exact dependence as the last time they were encountered. Locality is also strong for stores but not as strong as it is for loads. This is because store values are typically read more than once, hence stores tend to have more dependences than loads.

With a mechanism to predict whether a load or a store needs to be synchronized we next need: (1) a synchronization mechanism, and (2) a method of having the appropriate dynamic instances of loads and stores locate each other through the synchronization mechanism. In the rest of this section we first discuss a synchronization mechanism. Then, we consider how load and store instances locate each other through this synchronization mechanism.

An apt method of providing the required synchronization dynamically is to build an association between the store-load instruction pair. Suppose this dynamic association is a condition variable on which only two operations are defined: wait and signal, which test and set the condition variable respectively. These operations may be logically incorporated into the dynamic actions of the dependent load and store instructions to achieve the necessary synchronization.
This concept we illustrate with the example of Figure 15.8 where we assume that some method exists to dynamically associate store-load instruction pairs with condition variables (we discuss these means later in this section). As shown in part (a), an earlier mispeculation results in the association of a condition variable with a subsequent dynamic instance of the offending store-load instruction pair. With the condition variable in place, consider the sequence of events in the two possible execution sequences of the load and store instructions. In part (b), the load is ready to execute before the store. However, before the load executes, it tests the condition variable; since the test of the condition variable fails, the load waits. After the store executes, it sets the condition variable and signals the waiting load, which subsequently continues its execution as shown. No mispeculation is observed, and the sequential order is preserved. In part (c), the order of execution is a store followed by a load. After the stores executes, it sets the condition variable and records a signal for the load. Before the load executes, it tests the condition variable; since the test of the condition variable succeeds, the load continues its execution as shown (the condition variable is reset at this point). One may wonder why synchronization is provided even when the execution order follows the program order (i.e., store followed by load). This scenario represents the case where dependence prediction correctly indicates that a dependence exists but fails to detect that the order of execution has changed. The order of execution may change, for example, either (1) in response to external events whose behavior is not easy or desirable to track and predict, such as cache misses or resource conflicts, or (2) because of the successful synchronization of another, unrelated dependence. Synchronization is desirable even in these cases since, otherwise, the corresponding load will be delayed unnecessarily.

Once condition variables are provided, some means are required to assign a condition variable to a dynamic instance of a store-load instruction pair that has to be synchronized. If synchronization is to occur as planned, the mapping of condition variables to dynamic dependences has to be unique at any given point of time. One approach is to use just the address of the memory location accessed by the store-load pair as a handle. This method provides an indirect means of identifying the store and load instructions that are to be
synchronized. Unless the store location is accessed only by the corresponding store-load pair, the assignment will not be unique.

Alternatively, we can use the dependence edge as a handle. The static dependence edge may be specified using the (full or part of) instruction addresses (PCs) of the store-load pair in question. (Compared to using addresses, a potential advantage of this approach is that PC information is available earlier in the pipeline. This property could be exploited to reduce the effective latency of synchronization by having stores initiate synchronization in parallel or prior to the completion or initiation of their memory access.) Unfortunately, as exemplified by the code sequence of Figure 15.7 part (b), using this information may not be sufficient to capture the actual behavior of the dependence during execution; the pair (PC\textsubscript{ST}, PC\textsubscript{LD}) matches against all four edges shown even though the ones marked with dotted arrows should not be synchronized. A static dependence between a given store-load pair may correspond to multiple dynamic dependences, which need to be tracked simultaneously.

To distinguish between the different dynamic instances of the same static dependence edge, a tag (preferably unique) could be assigned to each instance. This tag, in addition to the instruction addresses of the store-load pair, can be used to specify the dynamic dependence edge. In order to be of practical use, the tag must be derived from information available during execution of the corresponding instructions. A possible source of the tag for the dependent store and load instructions is the address of the memory location to be accessed, as shown in Figure 15.7 part (c). An alternate way of generating tags is to have a load synchronize with the closest preceding instance of the store identified by the static dependence. While this scheme may delay a
load more than it should (as in our example, where \( \text{LD}_{a[c+0]} \) will wait for \( \text{ST}_{a[0+1]} \)), the performance impact of this delay may not be large.

In this work, and as our focus is on a distributed, split-window execution model where instructions are not fetched in order, we use an alternate way of generating instance tags. The scheme we use is an approximation of the scheme shown in part (d) of Figure 15.7, where dynamic store and load instruction instances are numbered based on their PCs. The difference in the instance numbers of the instructions which are dependent, referred to as the dependence distance, may be used to tag dynamic instances of the static dependence edge (as may be seen for the example code, a dependence edge between \( \text{ST}_i \) and \( \text{LD}_{i+\text{distance}} \) is tagged - in addition to the instruction PCs - with the value \( i+\text{distance} \)). We approximate this scheme by using the distance in processing units between the instructions that are mispredicted. Though all the aforementioned tagging schemes strive to provide unique tags, each may fall short of this goal under some circumstances (for example, the dependence distance may change in a way that we fail to predict, or the address accessed may remain constant across all instances of the same dependence).

In the rest of the discussion we restrict our attention to second scheme where the dependence distance is used to tag dependences. We note from a practical perspective, several inconveniences exist in the scheme we have just described (For example, how to track and predict multiple dependences per store or load). In the discussion that follows and for clarity, we initially ignore these issues and present an implementation framework in Section 15.5.

### 15.5 Implementation Framework

As we discussed in the previous section, in order to improve the accuracy of data dependence speculation, we attempt: (1) to predict dynamically, based on the history of mispredictions, whether a store-load pair is likely to be mispredicted and if so, (2) to synchronize the two instructions. In this section, we describe an implementation framework for this technique. For the purposes of this section we assume a centralized implementation, ignore the possibility of multiple dependences per load or store, and assume fully-associative structures. In Section 15.5.2, we address these issues.

We partition the support structures into two interdependent tables: a memory dependence prediction table (MDPT) and a memory dependence synchronization table (MDST). The MDPT is used to identify, through prediction, those instruction pairs that ought to be synchronized. The MDST provides a dynamic pool of condition variables and the mechanisms necessary to associate them with dynamic store-load instruction pairs to be synchronized. In the discussion that follows, we first describe the support structures and then proceed to explain their operation by means of an example. We present the support structures as separate, distinct components of the processor. Other implementations may be possible and desirable.

**MDPT:** An entry of the MDPT identifies a static dependence and provides a prediction as to whether or not subsequent dynamic instances of the corresponding static store-load pair will result in a misprediction (i.e., should the store and load instructions be synchronized). In particular, each entry of the MDPT consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), (4) dependence distance (DIST), and (5) optional prediction (not shown in any of the working examples). The valid flag indicates if the entry is currently in use. The load and store instruction address fields hold the program counter values of a pair of load and store instructions. This combination of fields uniquely identifies the static instruction pair for which it has been allocated. The dependence distance records the difference of the instance numbers of the store and load instructions whose misprediction caused the allocation of the entry (if we were to use a memory address to tag dependence instances this field would not have been necessary). The purpose of the prediction field is to capture, in a reasonable way, the past behavior of mispredictions for the instruction pair in order to aid in avoiding future mispredictions or
unnecessary delays. Many options are possible for the prediction field (for example an up-down counter or
dependence history based schemes). The prediction field is optional since, if omitted, we can always predict
that synchronization should take place. However, we note that in our experimentation we found that it is
better if synchronization is enforced only after a load has been mispeculated a couple of times (e.g., three
times).

**MDST:** An entry of the MDST supplies a condition variable and the mechanism necessary to synchronize
a dynamic instance of a static instruction pair (as predicted by the MDPT). In particular, each entry of the
MDST consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store
instruction address (STPC), (4) load identifier (LDID), (5) store identifier (STID), (6) instance tag
(INSTANCE), and (7) full/empty flag (F/E). The valid flag indicates whether the entry is, or is not, in use.
The load and store instruction address fields serve the same purpose as in the MDPT. The load and store
identifiers have to uniquely identify, within the current instruction window, the dynamic instance of the load
or the store instruction respectively. These identifiers are used to allow proper communication between the
instruction scheduler and the speculation/synchronization structures. The exact encoding of these fields
depends on the implementation of the OoO (out-of-order) execution engine (for example, in a superscalar
machine that uses reservation stations we can use the index of the reservation station that holds the instruc-
tion as its LDID or STID, or if we want to support multiple loads per store, a level of indirection may be
used to represent all loads waiting for a particular store). The instance tag field is used to distinguish
between different dynamic instances of the same static dependence edge (in the working example that fol-
lows we show how to derive the value for this field). The full/empty flag provides the function of a condition
variable.

### 15.5.1 Working Example

The exact function and use of the fields in the MDPT and the MDST is best understood with an example.
In the discussion that follows we are using the working example of Figure 15.9. For the working example,
assume that execution takes place on a processor which: (1) issues multiple memory accesses per cycle from
a pool of load and store instructions and (2) provides a mechanism to detect and correct mispeculations due
to memory dependence speculation. For the sake of clarity, we assume that once an entry is allocated in the
MDPT it will always cause a synchronization to be predicted.

Consider the memory operations for three iterations of the loop, which constitute the active pool of load
and store instructions as shown in part (a) of the figure. Further, assume that child->parent points to the same
memory location for all values child takes. The dynamic instances of the load and store instructions are
shown numbered, and the true dependences are indicated as dashed arrows connecting the corresponding
instructions in part (a). The sequence of events that leads to the synchronization of the ST2-LD3 dependence
is shown in parts (b) through (d) of the figure. Initially, both tables are empty. As soon as a mispeculation
(ST1-LD2 dependence) is detected, a MDPT entry is allocated, and the addresses of the load and the store
instructions are recorded (action 1, part (b)). The DIST field of the newly allocated entry is set to 1, which is
the difference of the instance numbers of ST1 and LD2 (1 and 2 respectively). As we noted earlier, we
approximate the instance numbers using the distance in processing units (incidentally this is identical to the
instance distance in our example). As a result of the mispeculation, instructions following the load are
squashed and must be re-issued. We do not show the re-execution of LD2.

As execution continues, assume that the address of LD3 is calculated before the address of ST2. At this
point, LD3 may speculatively access the memory hierarchy. Before LD3 is allowed to do so, its instruction
address, its instance number (which is three), and its assigned load identifier (the exact value of LDID is
immaterial) are sent to the MDPT (action 2, part (c)). The instruction address of LD3 is matched against the
contents of all load instruction address fields of the MDPT (shown in grey). Since a match is found, the
MDPT inspects the entry predictor to determine if a synchronization is warranted. Assuming the predictor indicates a synchronization, the MDPT allocates an entry in the MDST using the load instruction address, the store instruction address, the instance number of LD3, and the LDID assigned to LD3 by the OOO core (action 3, part (c)). At the same time, the full/empty flag of the newly allocated entry is set to empty. Finally, the MDST returns the load identifier to the load/store pool indicating that the load must wait (action 4, part (c)).

When ST2 is ready to access the memory hierarchy, its instruction address and its instance number (which is 2) are sent to the MDPT (action 5, part (d)). (We do not show the STID since, as we later explain, it is only needed to support control speculation.) The instruction address of ST2 is matched against the contents of all store instruction address fields of the MDPT (shown in grey). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. As a result, the MDPT adds the contents of the DIST field to the instance number of the store (that is, 2 + 1) to determine the instance number of the load that should be synchronized. It then uses this result, in combination with the load instruction address and the store instruction address, to search through the MDST (action 6, part (d)), where it finds the allocated synchronization entry. Consequently, the full/empty field is set to full, and the MDST returns the load identifier to the load/store pool to signal the waiting load (action 7, part (d)). At this point, LD3 is free to continue execution. Furthermore, since the synchronization is complete, the entry in the MDST is not needed and may be freed (action 8, part (d)).

If ST2 accesses the memory hierarchy before LD3, it is unnecessary for LD3 to be delayed. Accordingly, the synchronization scheme allows LD3 to issue and execute without any delays. Consider the sequence of relevant events shown in parts (e) and (f) of Figure 15.9. When ST2 is ready to access the memory hierarchy, it passes through the MDPT as before with a match found (action 2, part (e)). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. However, no matching entry is found there since LD3 has yet to be seen. Consequently, a new entry is allocated, and its full/empty flag is set to full (action 3, part (e)). Later, when LD3 is ready to access the memory hierarchy, it passes through the MDPT and determines that a synchronization is warranted as before (action 4, part (f)). The MDPT searches the MDST, where it now finds an allocated entry with the full/empty flag set to full (action 5, part (f)). At this point, the MDST returns the load identifier to the load/store pool so the load may continue execution immediately (action 6, part (f)). It also frees the MDST entry (action 7, part (f)).

15.5.2 Multiple Dependences Per Static Load or Store

Although not illustrated in the example, it is possible for a load or a store to match multiple entries of the MDPT and/or of the MDST. This case represents multiple memory dependences involving the same static load and/or store instructions (for example in the code “if (cond) store1 M else store2 M; load M,” there are two dependences (store1, load) and (store2, load) which may alternate in the dynamic execution stream. There are three challenges from a practical perspective: (1) how to predict multiple dependences per load, (2) how to allocate multiple MDST entries when multiple dependences are predicted on a single load, and (3) how to wake-up a load forced to wait on multiple MDST entries. One solution would be to define the problem away by tracking only a single dependence per store or load. However, support for multiple dependences per static instruction is very important.

The solution is to use level of indirection to represent the set of all dependences that have a common store or load (for example in the code “if (cond) store1 M; else store2 M; load M;” both the (store1, load) and the (store2, load) dependences will be represented using a common tag). This approach was suggested in [62,17]. In this scheme, separate entries for loads and stores are allocated in the MDPT. The format of these entries is shown in part (b) of Figure 15.10. (Note that a split MDPT and MDST is illustrated in the figure. As shown, in these entries, we do not record the dependences the corresponding instructions have. Instead
The image contains a diagram illustrating the synchronization of memory dependences in a computing context. The diagram includes several iterations and instructions, each labeled with iterations 1, 2, and 3. The code snippet shows a while loop with conditions and operations such as incrementing a count and checking pointers.

**Figure 15.9: Synchronization of memory dependences.**
we use a tag, to which we will refer to as a **synonym**. Synonyms are assigned using a global counter when misspeculations occur. If no synonym has been assigned to either the static load or the static store, a new synonym is generated using the global counter and is assigned to both instructions. If a synonym has been already assigned to only one of the instructions (as the result of misspeculating a different static dependence involving that instruction), the same synonym is assigned to the other instruction. If both instructions already have synonyms assigned to them which are different, the smallest one is assigned to both instructions [17].

15.6 Related Work

Naive memory dependence speculation was proposed for the Multiscalar architecture [26]. Support for dependence mispeculation detection and recovery was proposed in the form of the **Address Resolution Buffer (ARB)** [27] which also implements memory renaming. Other recently proposed techniques to support speculation of memory dependences, memory renaming and memory dependence mispeculation detection are presented in [31, 32]. Naive memory dependence speculation was used in the PA8000 processor [38] and in the Power 620 processor [1,50]. Knight also proposed using memory dependence speculation along with a hardware-based mispeculation detection mechanism in the context of speculative, parallel execution of otherwise sequential Lisp programs [47].

Finally, several hardware-based techniques have been proposed that aim at improving accuracy over naive memory dependence speculation. There are two closely related proposals. In the first proposal by Steely, Sager and Fite [84], mispeculated loads and stores are given tags derived from the addresses via which the mispeculation occur. These tags are used by the out-of-order scheduler to restrict load execution. Hesson, LeBlanc and Ciavaglia [33] describe the **store barrier cache** and the **store barrier** approach. An implementation of the store barrier cache was also presented [3]. The techniques we describe in this chapter were also reported [63,61].

Selective speculation is implemented in the Alpha 21264 processor [45] where an **independence predictor** is used to predict whether a load can execute freely. Finally, Chrysos and Emer proposed using a level of indirection for the purposes of memory dependence speculation/synchronization. In their **store set** approach a tag is used to represent the set of all stores that a load has had a memory dependence mispeculation. They proposed the incremental approach we also utilize to build memory dependence sets. Synchronization takes place through a separate table, and moreover, to preclude ordering problems on dependences with non-unit distances and to attain a simple synchronization table design, stores that have been assigned to the same store set (i.e., synonym) are executed in-order.
15.7 Experimental Results

In this section, we study various methods of extracting load/store parallelism and their interaction with memory dependence speculation under a centralized, continuous instruction window execution model. Specifically: (1) we demonstrate that higher performance is possible if we could extract load/store parallelism and that the performance improvements are higher when the instruction window is larger (Section 15.7.2). (2) We demonstrate that naive memory dependence speculation can be used to attain some of the performance benefits possible. However, we also demonstrate that the net penalty of mispeculation is significant (Section 15.7.3). (3) We consider using an address-based scheduler to extract this parallelism and show that higher performance is possible than when naive memory dependence speculation is used (Section 15.7.4). In an address-based scheduler, loads and stores post their addresses as soon as possible, and loads are allowed to inspect the addresses of preceding stores before obtaining a memory value. (4) We show that performance drops rapidly when inspecting preceding store addresses increases load execution latency (i.e., when going through the address-based scheduler increases load latency — Section 15.7.4). (5) We demonstrate that an organization where memory dependence prediction is used to schedule load/store execution — instead of using an address-based scheduler— offers performance similar to that possible had we had perfect in-advance knowledge of all memory dependences (Section 15.7.5).

We note that the various load/store execution models we consider in this section are derived from meaningful combinations of the following parameters: (1) whether loads are allowed to inspect preceding store addresses before obtaining a value from memory, (2) whether stores wait for both data and base registers to become available before posting their addresses for loads to inspect, (3) whether loads with ambiguous dependences can issue (i.e., whether memory dependence speculation is used).

15.7.1 Methodology

In all experiments reported in this chapter we used the SPEC95 benchmark suite [86]. We the benchmarks and input data sets shown in Table 15.1. All programs were compiled using the GNU gcc compiler version 2.7.2. The base instruction set architecture is the MIPS-I [42] but with no architectural delay slots of any kind. Fortran sources were compiled by first converting them to C using AT&T’s f2c compiler. All programs were compiled using the -O2 optimization level and with loop unrolling and function inlining enabled. We employ two simulation techniques: (1) trace-driven simulation and (2) detailed, execution-driven timing simulation. Traces were generated using a functional simulator for the MIPS-I ISA. The functional simulator executes all user-level instructions. System calls are serviced by the OS of the host machine. We also make use of detailed, execution-driven timing simulation. For this purpose we utilized a simulator of a dynamically scheduled superscalar processor. The simulator executes all user-level instructions including those on control speculative paths. Systems calls are redirected to the OS of the host machine.

The default superscalar configuration we used is detailed in Table 15.2. We used a 32K data cache to compensate for the relatively small memory working sets of the SPEC95 programs. For some experiments we use a 64-entry reorder buffer model. That model, has 4 copies of all functional units, a 2-port load/store queue and memory system, and can fetch up to 4 instructions per cycle.

Finally, to attain reasonable simulation times we utilized sampling for the timing simulations. In this technique which was also employed, for example, in [96,67,13], the simulator switches between functional and timing simulation. The mode of simulation is changed once a predefined number of instructions have been simulated. In all sampling simulations the observation size is 50,000 instructions. We chose sampling ratios that resulted in roughly 100M instructions being simulated in timing mode (i.e., sample size). We did not use sampling for 099.go, 107.mgrid, 132.ijpeg and 141.apsi. We used a 1:1 timing to functional simulation ratio (i.e., once 50000 instructions are simulated in timing mode, we switch to functional mode and simulate
18

50000 instructions before switching back to timing mode, and so on) for: 110.applu, 124.m88ksim, 130.li, 134.perl and 145.fpppp. We used a 1:2 timing to functional simulation ratio (i.e., once 5000 instructions are simulated in timing mode, we switch to functional mode and execute 100000 instructions before switching back to timing mode, and so on) for: 101.tomcatv, 102.swim, 126.gcc, 129.compress, 146.wave5 and 147.vortex. We used a 1:3 timing to functional simulation ratio for 103.su2cor. And finally, we used a 1:10 timing to functional simulation ratio for 104.hydro2d and 125.turb3d. During the functional portion of the simulation the following structures were simulated: I-cache, D-cache, and branch prediction.

15.7.2 Performance Potential of Load/Store Parallelism

An initial consideration with the techniques we proposed is whether exploiting load-store parallelism can yield significant performance improvements. The reason is that in order to determine or predict memory dependences we need additional functionality: (1) To determine memory dependences we need a mechanism where loads and stores can post their addresses and execute accordingly to the dependences detected, that is, we need an address-based scheduler. (2) To predict memory dependences we need a memory dependence

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECint’95</td>
<td></td>
</tr>
<tr>
<td>099.go</td>
<td>play level = 9, board size = 9</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>modified test input: 370 iterations of Dhrystone Benchmark</td>
</tr>
<tr>
<td>126.gcc</td>
<td>reference input file recog.i</td>
</tr>
<tr>
<td>129.compress</td>
<td>modified train input: maximum file size increased to 50,000</td>
</tr>
<tr>
<td>130.li</td>
<td>modified test input: (queens 7)</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>test input</td>
</tr>
<tr>
<td>134.perl</td>
<td>modified train input: jumple.pl with dictionary reduced by retaining every other 15th word</td>
</tr>
<tr>
<td>147.vortex</td>
<td>modified train input: persons.258k database, PART_COUNT 250, LOOKUPS 20, DELETES 20, STUFF_PARTS 100, PCT_NEWPARTS 10, PCT_LOOKUPS 10, PCT_DELETES 10, PCT_STUFFPARTS 10</td>
</tr>
<tr>
<td>SPECfp’95</td>
<td></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>modified train input: N = 41</td>
</tr>
<tr>
<td>102.swim</td>
<td>modified train input: X = 256, Y = 256</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>modified test input: LSIZE = 4 4 4 8 16</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>modified test input: MPROW = 200</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>modified test input: LMI = 4</td>
</tr>
<tr>
<td>110.applu</td>
<td>modified train input: itmax = 25, nx = 10, ny = 10, nz = 10</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>modified train input: nsteps = 1, itest = 0</td>
</tr>
<tr>
<td>141.apsi</td>
<td>modified train input: grid points x = 32, grid points y = 8, time steps = 130</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>modified reference input: natoms = 4</td>
</tr>
<tr>
<td>146.wave5</td>
<td>modified train input: particle distribution 1000 20, grid size 625x20</td>
</tr>
</tbody>
</table>

Table 15.1: Benchmark input parameters.
predictor and also a synchronization mechanism. For this reason an important consideration is whether this additional functionality is justified. Accordingly, we motivate the importance of exploiting load-store parallelism by comparing a model of a typical dynamically-scheduled ILP processor that does not attempt to determine and exploit load-store parallelism with one that is identical except in that it includes an oracle load-store disambiguation mechanism. Under this model, execution proceeds as follows: After an instruction is fetched, it is decoded and entered into the instruction window where its register dependences and availability are determined. If the instruction is a store, an entry is also allocated in a store buffer to support memory renaming and speculative execution. All instructions except loads can execute (i.e., issue) as soon as their register inputs become available. Stores wait for both data and address calculation operands before issuing. As a result, loads wait in addition for all preceding stores to calculate their addresses or write their data.

Table 15.2: Default configuration for superscalar timing simulations

Figure 15.1 reports the performance improvements possible when perfect memory dependence information is available. We consider two base configurations, one with a 64-entry instruction window and one with a 128-entry window. For all programs, exploiting load/store parallelism has the potential for significant performance improvements. Furthermore, we can observe that when loads wait for all preceding store ("NO")
bars) increasing the window size from 64 to 128 results in very small improvements. However, when the oracle disambiguator is used, performance increases sharply. This observation suggests that the ability to extract load/store parallelism becomes increasingly important performance wise as the instruction window increases.

When loads are forced to wait for all preceding stores to execute it is false dependences that limit performance. The fraction of loads that are delayed as the result of false dependences along with the average false dependence resolution latency are given in Table 15.3. We report false dependences as a fraction over all committed loads. We account for false dependences once per executed load and the time the load has calculated its address and could otherwise access memory. If the load is forced to wait because a preceding store has yet to access memory, we check to see if a true dependence with a preceding yet un-executed store exists. If no true dependence exists, we include this load in our false dependence ratio (this is done only for loads on the correct control path). We define, false dependence resolution latency to be the time, in cycles, a load that could otherwise access memory is stalled, waiting for all its ambiguous memory dependences to get resolved (i.e., all preceding stores have executed). We can observe that the execution of many loads and in some cases of most loads, is delayed due to false dependences and often for many cycles.

15.7.3 Performance with Naive Memory Dependence Speculation

As we have seen, extracting load/store parallelism can result in significant performance improvements. In this section we measure what fraction of these performance improvements naive memory dependence speculation can offer. For this purpose, we assume the same processor model assumed in the previous section but we allow loads to speculatively access memory as soon as their address operands become available. All
speculative load accesses are recorded in a separate structure, so that preceding in the program order stores can detect whether a true memory dependence was violated by a speculatively issued load. Figure 15.1, part (a) reports performance (as IPC) for the 128-entry processor model when, from left to right, no speculation is used, when oracle dependence information is available and when naive memory dependence speculation is used. We can observe, that for all programs naive memory dependence speculation results in higher performance compared to no speculation. However, the performance difference between naive memory dependence speculation and the oracle mechanism is significant, supporting our claim that the net penalty of mispeculation can become significant. Memory dependence misspeculations are at fault. The frequency of memory dependence misspeculations is shown in part (b) of Figure 15.1. We measure mispeculation frequency as a percentage over all committed loads. A breakdown in terms of the address-space through which the mispeculation occurs is also shown. We can observe, that though loads cannot inspect preceding store addresses, mispeculations are rare. Nevertheless, the impact misspeculations have on performance is large.

In this context, the memory dependence speculation/synchronization methods we proposed could be used to reduce the net performance penalty due to memory dependence misspeculations. However, before we consider this possibility (which we do in Section 15.7.5) we first investigate using an address-based scheduler to extract load/store parallelism and its interaction with memory dependence speculation.

### 15.7.4 Using Address-Based Scheduling to Extract Load/Store Parallelism

We have seen that using oracle memory dependence information to schedule load/store execution has the potential for significant performance improvements. In this section we consider using address-based dependence information to exploit load/store parallelism. In particular we consider an organization where an address-based scheduler is used to compare the addresses of loads and stores and to guide load execution. We confirm that even in this context, memory dependence speculation offers superior performance compared to not speculating loads. However, we also demonstrate that when having to inspect addresses

<table>
<thead>
<tr>
<th></th>
<th>False Dependences%</th>
<th>Resolution Latency</th>
<th></th>
<th>False Dependences%</th>
<th>Resolution Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>26.4%</td>
<td>13.7</td>
<td>tomcatv</td>
<td>61.2%</td>
<td>36.3</td>
</tr>
<tr>
<td>m88ksim</td>
<td>59.9%</td>
<td>14.8</td>
<td>swim</td>
<td>91.0%</td>
<td>5.4</td>
</tr>
<tr>
<td>gcc</td>
<td>39.0%</td>
<td>47.3</td>
<td>su3cor</td>
<td>79.6%</td>
<td>91.2</td>
</tr>
<tr>
<td>compress</td>
<td>70.3%</td>
<td>18.5</td>
<td>hydro2d</td>
<td>85.2%</td>
<td>9.7</td>
</tr>
<tr>
<td>li</td>
<td>44.2%</td>
<td>39.1</td>
<td>mgrid</td>
<td>45.4%</td>
<td>26.6</td>
</tr>
<tr>
<td>ijpeg</td>
<td>70.3%</td>
<td>22.9</td>
<td>applu</td>
<td>45.4%</td>
<td>26.6</td>
</tr>
<tr>
<td>perl</td>
<td>59.8%</td>
<td>39.1</td>
<td>turb3d</td>
<td>77.0%</td>
<td>55.6</td>
</tr>
<tr>
<td>vortex</td>
<td>67.2%</td>
<td>54.5</td>
<td>apsi</td>
<td>77.5%</td>
<td>78.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fpppp</td>
<td>88.7%</td>
<td>51.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>wave5</td>
<td>83.6%</td>
<td>9.7</td>
</tr>
</tbody>
</table>

Table 15.3: Fraction of loads with false dependences and average false dependence resolution latency (cycles) for the 128-entry instruction window processor.
increases load execution latency, performance drops compared to the organization where oracle dependence information is available in advance (which we evaluated in the preceding section).

In the processor models we assume, stores and loads are allowed to post their addresses for disambiguation purposes as soon as possible. That is, stores do not wait for their data before calculating an address. Furthermore, loads are allowed to inspect preceding store addresses before accessing memory. If a true dependence is found the load always wait. When naive memory dependence speculation is used, a mispecu-

---

**Figure 15.1:** Naive memory dependence speculation. (a) Performance results (IPC). (b) Memory dependence mispeculation frequency.
lation is signaled only when: (1) a load has read a value from memory, (2) the value has been propagated to other instructions, and (3) the value is different than the one written by the preceding store that signals the mispeculation. As we noted earlier, under this processor model mispeculations are virtually non-existent. There are three reasons why this is so: (1) loads get either delayed because they can detect that a true dependence exists with a preceding store, (2) loads with unresolved dependences that correspond to true dependences are allowed to access memory but before they have a chance of propagating the value read from memory they receive a value from a preceding store, or (3) loads are delayed because preceding stores consume resources to have their addresses calculated and posted for disambiguation purposes.

Figure 15.2 reports how performance varies when naive memory dependence speculation is used compared to the same configuration that performs no speculation of memory dependences. For these experiments we use an 128-entry window processor model. We also measure how performance varies in terms of the time it takes for loads and stores to go through the address-based scheduler. We vary this delay from 0 to up to 2 cycles. In the calculation of the relative performance with naive speculation of part (a) of the figure, we should note that the base configuration is different for each bar. The absolute performance (i.e., the IPC) of the corresponding base configuration is reported in part (b). It can be seen that, for most programs, naive memory dependence speculation is still a win. Performance differences are not as drastic as they were for the model where loads could not inspect preceding store addresses, yet they are still significant. More importantly the performance difference between no speculation and memory dependence speculation increases as the latency through the load/store scheduler also increases. For some programs, naive memory dependence speculation results in performance degradation. These programs are 147.vortex for all scheduling latencies and 145.fpppp when the scheduling latency is 0. It is not misspeculations that cause this degradation. This phenomenon can be attributed to loads with ambiguous dependences that get to access memory speculatively only to receive a new value from a preceding store before they had a chance to propagate the value they read from memory. These loads consume memory resources that could otherwise be used more productively. This phenomenon supports our earlier claim that there is an opportunity cost associated with erroneous speculation.

While including an address-based scheduler does help in exploiting some of the load/store parallelism, a load may still be delayed even when naive memory dependence speculation is used. The reason is that in a real implementation for preceding stores to calculate their addresses and post them for scheduling purposes, they must consume resources. These resources include issue bandwidth, address calculation adders and load/store scheduler ports. The same applies to loads that should wait for preceding stores. If perfect knowledge of dependences was available in advance, stores would consume resources only when both their data and address calculation operands become available. For this reason, we next compare the absolute performance of the processor models that use an address-based scheduler to that of the processor model that utilizes oracle dependence information to schedule load execution (Section 15.7.2).

Figure 15.3 reports relative performance compared to the configuration that uses no speculation but utilizes an address-based scheduler with 0 cycle latency (the IPC of this configuration was reported in Figure 15.2, part (b), 0-CYCLE configuration). From left to right, the four bars per program report performance with: (1) oracle disambiguation and no address-based scheduler (Section 15.7.2), (2) through (4) naive memory dependence speculation and address-based scheduler with a latency of 0, 1 and 2 cycles respectively (which we evaluated earlier in this section). We can observe that, with few exceptions, the 0-cycle address-based scheduler that uses naive speculation and the oracle mechanism perform equally well. Interestingly, the oracle configuration performance significantly better for 147.vortex and 145.fpppp supporting our earlier claim about resource contention and the opportunity cost associated with erroneous speculation. We can also observe that once it takes 1 or more cycles to go through address-based disambiguation the oracle configuration has a clear advantage. The only exception is 104.hydro2d where the oracle configuration does significantly worse. This result may come as a surprise, however it is an artifact of our euphemistic use of the term “oracle”. In the oracle model we assume, a store is allowed to issue only after both its data and address oper-
ands become available. As a result, dependent loads always observe the latency associated with store address calculation, which in this case is 1 cycle to fetch register operands and 1 cycle to do the addition. Under these conditions, dependent loads can access the store value only after 3 cycles the store has issued.

Figure 15.2: (a) Relative performance of naive memory dependence speculation as a function of the address-based scheduler latency. Performance variation is reported with respect to the same processor model that does not use memory dependence speculation. Base performance (IPC) is shown in part (b).
When the address-based scheduler is in place, a store may calculate its address long before its data is available and dependent loads can access the store’s value immediately. In an actual implementation, it may be possible to overlap store address calculation and store data reception without using an address-based scheduler (e.g., [34, 84]).

15.7.5 Speculation/Synchronization

In this section we consider using an implementation of our speculation/synchronization approach to improve accuracy over naive memory dependence speculation. As we have observed in the previous chapter, in a continuous window processor that utilizes an address-based load/store scheduler along with naive memory dependence speculation, misspeculations are virtually non-existent. In such an environment there is no need for our speculation/synchronization method. However, as we observed in Section 15.7.3, if an address-based scheduler is not present then the net penalty of misspeculation resulting from naive memory dependence speculation is significant. Moreover, and as we have seen in the previous section, the performance potential of a method such a speculation/synchronization (oracle configuration of Figure 15.3) is often close or exceeds the performance possible with an address-based scheduler with 0 cycle latency. Accordingly, in this section we restrict our attention to a configuration that does not use an address-based load/store sched-
uler. In the rest of this section we first provide the details of the speculation/synchronization mechanism we simulated and then proceed to evaluate its performance.

The speculation/synchronization mechanism we used in these experiments comprises a 4K, 2-way set associative MDPT in which separate entries are allocated for stores and loads. Dependencies are represented using synonyms, i.e., a level of indirection. The algorithm used to generate synonyms is the one described in Section 15.5.2 with the only difference being that no unit distance is associated with each synonym (units distances were used in the Multiscalar execution model). It is implied that a load is synchronized with the last preceding store instance that has been assigned the same synonym. MDPT entries are allocated —if they don’t exist already— upon the detection of a memory dependence violation for both offending instructions. No confidence mechanism is associated with each MDPT entry; once an entry is allocated, synchronization is always enforced. However, we flush the MDPT every million cycles to reduce the frequency of false dependences (this method was proposed in [17]). The functionality of the MDST is incorporated into the register-scheduler, which we assume to follow the RUU model [81]. This is done as follows: an additional register identifier is introduced per RUU entry to allow the introduction of speculative dependences for the purposes of speculation/synchronization. Stores that have dependences predicted use that register identifier to mark themselves as producers of the MDPT supplied synonym. Loads that have dependences predicted by the MDPT, use that register identifier to mark themselves as consumers of the MDPT supplied synonym. Synchronization is achieved by: (1) making loads wait for the immediately preceding store (if there is any) that is marked as the producer of the same synonym, and (2) having stores broadcast their synonym once they issue, releasing any waiting loads. A waiting load is free to issue one cycle after the store it speculatively depends upon issues.

Figure 15.4, part (a) reports performance results relative to naive memory dependence speculation (Section 15.7.3). As it can be seen our speculation/synchronization mechanism offers most of the performance improvements that are possible had we had perfect in advance knowledge of all memory dependences (oracle). This is more clearly shown in part (b) of the same figure, where we report the relative over our speculation/synchronization mechanism performance of oracle speculation. On the average, the performance obtained by the use of our mechanism is within 1.001% of that possible with the oracle mechanism. For four programs (126.gcc, 101.tomcatv, 102.swim and 107.mgrid) our mechanism results in performance that is virtually identical to that possible with oracle speculation. For the rest of the programs the differences are relatively minor (3% in the worst case) when compared to the performance improvements obtained over naive speculation. To help in interpreting these differences we also present the mispeculation rates exhibited when our mechanism is in place. These results are shown in Table 15.4 (reported is the number of mispeculations over all committed loads). As it can be seen, mispeculations are virtually non-existent. This observation suggests that for the most part the performance differences compared to oracle speculation are the result of either (1) false dependences, or (2) of failing to identify the appropriate store instance with which a load has to synchronize with. False dependences are experienced when our mechanism incorrectly predicts that a load should wait although no store is actually going to write to the same memory location. In some cases and even when prediction correctly indicates that a dependence exists, our mechanism may fail to properly identify the appropriate store instance. This is the case for memory dependences that exhibit non-unit instance distances (e.g., a[i] = a[i - 2]). In such cases, a load is delayed unnecessarily as it is forced to wait for the very last preceding store instance that has been assigned the same synonym.

The results of this section suggest that our speculation/synchronization method can offer performance that is very close to that possible had we had perfect, in advance knowledge of all memory dependences. However, further investigation is required to determine how selective speculation and the store barrier policy would perform under this processor model.
Techniques to exploit instruction-level parallelism (ILP) are an integral part of virtually all modern high-performance processors. With these techniques, instructions do not necessarily execute one after the other.

Figure 15.4: Performance of an implementation of speculation/synchronization. (a) Performance improvements over naive speculation. (b) Relative performance of oracle speculation over our speculation/synchronization.

15.8 Summary

Techniques to exploit instruction-level parallelism (ILP) are an integral part of virtually all modern high-performance processors. With these techniques, instructions do not necessarily execute one after the other.
and in the order they appear in the program. Rather, instructions are executed in any order convenient provided however, that program semantics are maintained (i.e., the same results are produced). This ability is useful in reducing execution time by executing instructions in-parallel (many at the same time) and by avoiding stalling execution while an instruction takes its time to execute (e.g., it performs a relatively time consuming calculation or accesses a relatively slow storage device for its operands). An arena where ILP techniques are particularly useful is that of tolerating memory latency where these techniques are used to send loads requests as early as possible, overlapping memory processing time with other useful computation.

One approach to exploiting ILP is to first make sure that executing an instruction will not violate program semantics before the instruction is allowed to execute. In the case of a load, this action amounts to waiting to determine if a preceding, yet to be executed store writes to the same memory location, that is whether a true dependence with a preceding store exists. However, and as we explain in detail in chapter 3, waiting to determine whether a preceding store writes to the same memory location is not the best option. Higher performance is possible if memory dependence speculation is used, that is, if a load is allowed to execute speculatively before a preceding store on which it may be data dependent. Later on, and after the preceding store has calculated its address, we can check whether program semantics were violated. If no true memory dependence is violated in the resulting execution order, speculation was successful. Otherwise, speculation was erroneous and corrective action is necessary to undo the effects of erroneous execution. A penalty is typically incurred in the latter case.

In this chapter we focused on dynamic memory dependence speculation techniques and studied how existing methods of applying memory dependence speculation will scale for future generation processors. We demonstrated that as processors attempt to extract higher-levels of ILP by establishing larger instruction windows: (1) memory dependence speculation becomes increasingly important, and (2) the net penalty of memory dependence mispeculation can become significant. The latter observation suggests that further performance improvements are possible if misspeculations could be avoided. In the conventional centralized,

<table>
<thead>
<tr>
<th>Mispeculation Rate%</th>
<th>NAIVE</th>
<th>SYNC</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>2.5%</td>
<td>0.0301%</td>
</tr>
<tr>
<td>m88ksim</td>
<td>1.0%</td>
<td>0.0030%</td>
</tr>
<tr>
<td>gcc</td>
<td>1.3%</td>
<td>0.0028%</td>
</tr>
<tr>
<td>compress</td>
<td>7.8%</td>
<td>0.0034%</td>
</tr>
<tr>
<td>li</td>
<td>3.2%</td>
<td>0.0035%</td>
</tr>
<tr>
<td>jpeg</td>
<td>0.8%</td>
<td>0.0090%</td>
</tr>
<tr>
<td>perl</td>
<td>2.9%</td>
<td>0.0029%</td>
</tr>
<tr>
<td>vortex</td>
<td>3.2%</td>
<td>0.0286%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>1.0%</td>
<td>0.0001%</td>
</tr>
<tr>
<td>swim</td>
<td>0.9%</td>
<td>0.0017%</td>
</tr>
<tr>
<td>su3cor</td>
<td>2.4%</td>
<td>0.0741%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>5.5%</td>
<td>0.0740%</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.1%</td>
<td>0.0019%</td>
</tr>
<tr>
<td>applu</td>
<td>1.4%</td>
<td>0.0039%</td>
</tr>
<tr>
<td>turb3d</td>
<td>0.7%</td>
<td>0.0009%</td>
</tr>
<tr>
<td>apsi</td>
<td>2.1%</td>
<td>0.0148%</td>
</tr>
<tr>
<td>fpppp</td>
<td>1.4%</td>
<td>0.0096%</td>
</tr>
<tr>
<td>wave5</td>
<td>2.0%</td>
<td>0.0034%</td>
</tr>
</tbody>
</table>

Table 15.4: Memory dependence mispeculation rate with our speculation/synchronization mechanism ("SYNC" columns) and with naive speculation ("NAIVE" columns).
Motivated by the aforementioned observations we studied the trade-offs involved in memory dependence speculations and presented techniques to improve the accuracy of memory dependence speculation. Specifically, we presented techniques to: (1) identify via memory dependence prediction those loads and stores that would otherwise be mis Speculated, and (2) delay load execution only as long as it is necessary to avoid a memory dependence mispeculation. The best performing technique we propose is memory dependence speculation and synchronization, or speculation/synchronization. With this technique, initially loads are speculated whenever the opportunity exists (as it is common today). However, when mispeculations are encountered, information about the violated dependence is recorded in a memory dependence prediction structure. This information is subsequently used to predict whether the immediate execution of a load will result in a memory dependence violation, and (2) if so, which is the store this load should wait for.

We studied memory dependence speculation under a conventional centralized, continuous window processor (typical current superscalar) that utilizes fetch and execution units of equal bandwidth, and a program order priority scheduler (i.e., when there are many instructions ready to execute, the older ones in program order are given precedence). For this processor model we made two observations. The first is that using an address-based load/store scheduler (i.e., a structure where loads can inspect preceding store addresses to decide whether memory dependences exist) coupled with naive memory dependence speculation offers performance very close to that possible with perfect, in advance knowledge of all memory dependences, provided that going through the address-based scheduler does not increase load latency. The second is that if building an address-based load/store scheduler is not an option (clock cycle) or not a desirable option (complexity), naive memory dependence speculation can still offer most of the performance benefits possible by exploiting load/store parallelism. However, under this set of constraints the net penalty of mispeculation is significant suggesting that our memory dependence speculation and synchronization technique might be useful in improving performance. Specifically, timing simulations show that an implementation of our techniques results in performance improvements of 19.7% (integer) and 19.1% (floating-point) which are very close to those ideally possible: 20.9% (integer) and 20.4% (floating-point).

While most modern processors utilize a centralized, instruction window it is not unlikely that future processors will have to resort to distributed, split-window organizations. As many argue, the reason is that the existing, centralized methods may not scale well for future technologies and larger instruction windows (e.g., [66]). An example were techniques similar to those we presented have been put to use already exists: the Alpha 21264 processor utilizes both a split-instruction window approach and selective memory dependence speculation to reduce the net penalty of dependence mis speculations [45]. As our results indicate, techniques to predict and synchronize memory dependences can greatly improve the performance of future split-window processors, allowing them to tolerate slower memory devices by aggressively moving loads up in time while avoiding the negative effects of memory dependence mis-speculations. Moreover, speculation/synchronization may be used as a potentially lower complexity/faster clock cycle alternative to using an address-based load/store scheduler for exploiting load/store parallelism.

Bibliography


