A Dynamic Approach to Improve the Accuracy of Data Speculation

Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, Guri. S. Sohi
{moshovos, breach, vijay, sohi}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton St.
Madison, WI 53706
Phone: (608)-262-7985

March 13, 1996

A Dynamic Approach to Improve the Accuracy of Data Speculation

Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, Gurindar S. Sohi
{moshovos, breach, vijay, sohi}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton St.
Madison, WI 53706
Phone: (608)-262-7985
March 13, 1996

Abstract

Data speculation is used in instruction-level parallel (ILP) processors to allow early execution of an instruction before a logically preceding instruction on which it may be data dependent. If the instruction is independent, data speculation succeeds; if not, it fails, and the two instructions must be synchronized. This paper considers dynamic techniques to improve the accuracy with which data speculation is carried out. We propose dynamic techniques: (i) to predict if the execution of an instruction is likely to result in a data misspeculation, and (ii) to provide the synchronization needed to avoid a misspeculation. Experimental results evaluating the effectiveness of the proposed techniques are presented within the context of a Multiscalar processor.

1 Introduction

Speculative execution is an integral part of modern ILP processors, be they statically- or dynamically-scheduled designs. Speculation takes on two forms: *control speculation* and *data speculation*. Control speculation implies the execution of an instruction before its control dependences are resolved (i.e., before the execution of a preceding instruction on which it is control dependent). Data speculation implies the execution of an instruction before its data dependences are resolved (i.e., before the execution of a preceding instruction on which it *may* be data dependent).

The profitability of speculation depends upon two factors: (i) the overhead associated with performing the speculation, and (ii) the costs associated with misspeculation. The overhead associated with speculation includes the hardware and software means required to take corrective action in case of a misspeculation. The cost of misspeculation is a function of the probability of a misspeculation and the cost of recovering from a misspeculation. The cost of recovering from a misspeculation includes: (i) the inherent cost of restoring the correct machine state and (ii) the incidental cost of squashing the work of other (possibly correct) instructions.

To date, much attention has been focused on control speculation. This outlook is natural because control speculation is the first step. Control speculation (or some equivalent basic block enlargement technique such as if-conversion with predicated execution [3, 4]) is required if we want to consider instructions from more than one basic block for possible issue. Given the sizes of naturally-occurring basic blocks, the need to go beyond a basic block became apparent some time ago, and several techniques to permit control speculation have been developed, both in the context of statically- and dynamically-scheduled machine models [2,4,5,6,7,8,9,10,11].

To improve the accuracy of control speculation, branch prediction techniques are used. Improving the accuracy of control speculation (especially dynamic techniques) has been the subject of intensive research recently, and a plethora of papers on dynamic and static branch prediction techniques have been published.

The problem of data speculation has not received as much attention as the problem of control speculation. While the problem of ensuring correct execution while carrying out data speculation has received some attention [1,12,13,14], the issue of improving the accuracy of data speculation, especially dynamic techniques to do so, has not received any attention at all.

This paper is concerned with dynamic techniques for improving the accuracy of data speculation. In Section 2, we present the problem of data speculation and discuss how it affects different ILP execution models. Next, in Section 3, we discuss the components of a method for accurate and aggressive memory data speculation. In Section 4, we present a number of different implementations of this method. In Section 5, we provide an evaluation of these implementations within the context of a Multiscalar processor [15, 16, 17, 18]. Finally, we provide a summary of this work in Section 6 and offer concluding remarks.

2 Data Speculation

As a program executes, data values are produced and consumed by instructions of the program; such values are conveyed from the producer to the consumer by binding the value to a named storage (register or memory) location. Programs are written with an implied (sequential or logical or total) order. An ILP (or other parallel) machine, takes a suitable subset of the instructions (an instruction window) of a program and converts the total order (within this window) into a partial order, so that instructions may execute in parallel. The shape of the partial order (and the parallelism so obtained) is heavily influenced by the dependences that exist between the instructions in the total order. Dependences can be *unambiguous* (i.e., an instruction consumes a value that is known to be created by an instruction preceding it in the total order) or *ambiguous* (i.e., an instruction consumes a value that *may* be produced by an instruction preceding it in the total order). For example, an instruction may use a value bound to a register that may or may not be produced by a preceding instruction in the dynamic execution (the production of the value is governed by a control condition).

The problem of dealing with ambiguous data dependences is most acute in the case where the production and consumption of data is through memory. The primary reason for this distinction between registers and memory is that the existence of aliases is inherent in the memory name-space, while the register name-space is free of aliases. That is, registers are directly specified and usually may be analyzed statically. On the other hand, memory is indirectly specified and often may not be analyzed statically. Accordingly, we restrict our discussion in this paper to the speculation of dependences through memory, i.e., the speculation of load instructions, even though all the concepts presented in this paper could easily be applied to the data speculation of dependences through registers.

If a dependence is unambiguous, the producer and consumer must be *synchronized*, i.e., the consuming instruction must be delayed until the producing instruction has provided the value [19,20,21]. If the dependence is ambiguous, data speculation may be used. In other words, data speculation is the reordering of execution of producers and consumers with ambiguous data dependences such that an instruction is scheduled to execute before (logically preceding) instructions that may produce values it consumes. The speculation is erroneous (i.e., a *misspeculation*) if the resulting execution violates a true dependence inherent in the original program.

I1 : st $M(R_1)$
I2 : ld $M(R_2)$
I3 : ld $M(R_3)$
(a)

I2 : ld $M(R_2)$
I1 : st $M(R_1)$
I3 : ld $M(R_3)$
(b)

I3 : ld $M(R_3)$
I1 : st $M(R_1)$
I2 : ld $M(R_2)$
(c)

Figure 1. *Load/Store dependence example.*

The example of Figure 1 illustrates the concept of data speculation. As shown in part (a), the original program order specifies a store (I1) followed by two loads (I2 and I3). At the time (static or dynamic) instruction scheduling must be performed, the addresses of the store and the loads are unknown. Therefore, ambiguous memory dependences exist between the store and the first load, and between the store and the second load, as indicated by the arrows. The actual memory dependence in this example is between the store and the second load, as indicated by the dark arrow; no actual memory dependence exists between the store and the first load, as indicated by the light arrow. If the code sequence is executed as per the original program order (I1 I2 I3) no difficulty can arise. However, consider the case where data speculation has allowed the execution of one of the loads before the store. As shown in the new order (I2 I1 I3) of part (b), where the first load is scheduled before the store, this execution proceeds without mishap, because no true memory dependence exists between I1 and I2, and the true memory dependence between I1 and I3 is honored. In contrast, as shown in the new order (I3 I1 I2) of part (c), where the second load is scheduled before the store, the execution violates the program semantics. Precautions must be taken to ensure that I3 executes after I1 in any partially-ordered execution schedule.

The means for detecting erroneous data speculation and ensuring correct behavior depend upon the processing model. In a VLIW processor, software (run-time disambiguation, or RTD [12,14]) or a combination of software and hardware (the Memory Conflict Buffer [1]) is responsible for detecting the data misspeculation, and recovery software is responsible for recovering from the misspeculation.[1] In a superscalar processor, memory disambiguation hardware is responsible for detecting a mispeculation (if the processor carries out data speculation). The precise-state recovery hardware (used to recover from a control misspeculation) also serves the purpose of recovery from a data misspeculation. Likewise, in a Multiscalar processor, memory disambiguation hardware (the Address Resolution Buffer, or ARB [15,13]) is responsible for detecting a misspeculation, and the precise-state recovery hardware is used to recover from a data misspeculation.

To minimize the net cost of data misspeculation, we need to improve the accuracy of data speculation. A data speculation is erroneous if there is indeed a true dependence between the operations of interest. To improve the accuracy of speculation we therefore have to reduce the probability that we (incorrectly) speculate instructions that are truly dependent upon (not yet executed) preceding instructions. This goal is tantamount to improving the accuracy of memory disambiguation, i.e., classifying a dependence through memory as a true dependence.

Within the above context, VLIW processors have considered the problem of statically improving the accuracy of data speculation by attempting to reduce the probability that data speculation has to be carried out. Any available static memory dependence analysis technique may be able to unambiguously state that a true dependence exists, or that no dependence exists, in which case there is no need to resort to data speculation. However, such static techniques have not been very successful: many memory dependences are classified as ambiguous dependences, especially in non-numeric applications, necessitating the need for data speculation [1,12,14].

Most dynamically-scheduled superscalar processors have not exploited data speculation to date, mainly because the window sizes that modern superscalar processors can establish dynamically is limited to a few tens of instructions (mainly due to branch prediction limitations). Even if branch prediction allowed large dynamic windows to be built, hardware resources may not be able to support such a large instruction window. For example, the MIPS R10000 [22] is limited to at most 32 active instructions or four outstanding conditional branches (whichever comes first). Moreover, as instructions are entered into the dynamic window sequentially, stores are encountered before logically succeeding loads. Since address calculation typically requires simple arithmetic, addresses of stores can be computed very soon (assuming the base register is free). Thus, when it comes time to issue a load, the addresses of previous stores are known; data speculation is not needed in this case. However, as the instruction window size increases, the need for data speculation becomes more acute as exemplified by recent dynamically-scheduled superscalar processors which implement data speculation of memory references (albeit with no regard for the accuracy of this data speculation) [23,24].

In a dynamically-scheduled processing model with multiple (dynamic) program counters, such as the Multiscalar model [15,16,17,18], the problem of data speculation is especially important. In the Multiscalar model, multiple program counters are used to sequence through the static (sequential) program in parallel, with heavy use of control and data speculation. Here, a load may be issued before it is even known if any logically preceding stores exists, in which case the addresses of the previous store operations (if any) are irrelevant. In other words, every load is potentially a data speculative load whose actual dependences are unknown. In this situation, improving the accuracy of data speculation is crucial.

3 Components of a Solution

To improve the accuracy of data speculation, we have to dynamically detect that a data dependence is likely to be violated and convert speculation into on-the-fly synchronization. That is, when a load is ready to execute, predict whether it is likely to violate a true data dependence, and if so, delay it until a point when the load is likely (or certain) not to do so. For example, delay the load until the logically preceding store on which it depends has finished or is likely to have finished. There are three parts to this problem: (i) dynamically identify the store-load pairs that are likely to be data dependent, (ii) assign a synchronization mechanism to dynamic instances of these dependences, and (iii) use this mechanism to synchronize the store and the load instructions.

Dynamically tracking all possible ambiguous store-load pairs (analogous to what has to be done statically), is not an option that we consider desirable, or even practical. Fortunately, our experimental observations suggest that the following phenomena exists: *the static store-load instruction pairs that cause most of the dynamic data misspeculations are relatively few and exhibit temporal locality*[2]. That is, at any given time, different dynamic instances of a few

1. Some combination of the architecture, hardware, and software is also responsible for dealing with another issue: the correct handling of exceptions. Since this aspect is orthogonal to the subject of this paper and is handled as a matter of course in a dynamically-scheduled processors [1, 2], we do not discuss it any further.

static store-load pairs, either operating repeatedly on the same memory location (scalar variable) or operating on different memory locations, account for the majority of the misspeculations. This observation suggests that we may use past history to dynamically identify and track such store-load pairs, and cache this information (in a storage structure of reasonable size). The remaining issue is by what means to synchronize the store-load pair.

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 [25] 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 to achieve the necessary synchronization.

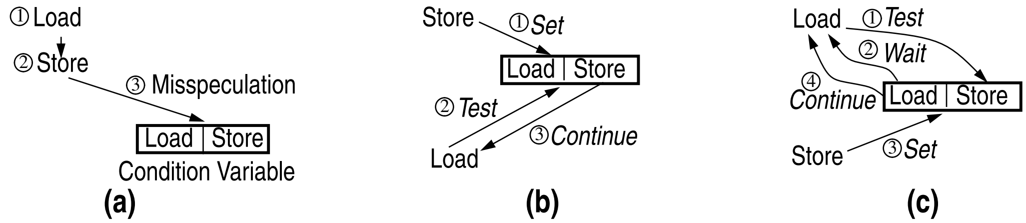

Figure 2. *Synchronization example*

The above concept is illustrated in the example of Figure 2. Assume, as shown in part (a), a misspeculation has led to the (dynamic) association of a condition variable with the offending load and store instructions. 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 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). In part (c), the order of execution is a load followed by a store. Before the load executes, it tests the condition variable; since the test of the condition variable fails, the load waits until the store sets the condition variable. After the stores executes, it sets the condition variable and signals the waiting load, which subsequently continues its execution as shown.

One approach to assigning a condition variable uses the data address of the memory location accessed by the misspeculated 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 storage location is accessed only by the corresponding store-load pair, the synchronization may not occur as planned. This subtle problem is evident because the misspeculation (resulting from data speculation) is not a product of the storage location (the site of the misspeculation); instead, it is a product of the edge between the producing store and the consuming load instructions (the source of the misspeculation).

Accordingly, an alternate approach is to use the dependence edge as a handle. The dependence edge may be specified using the instruction addresses (PCs) of the store-load pair in question. Unfortunately, as exemplified by the code sequence of Figure 3 part (b), just specifying the edge is not sufficient to capture the actual behavior of the dependence during execution. 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 data address of the memory location to be accessed, as shown in Figure 3 part (c). An alternate way of generating instance tags is shown in Figure 3 part (d) 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 ST_i and $LD_{i+distance}$ is tagged with the value,

2. As we show in the experimentation section by remembering up to 64 store-load pairs from those that conflicted in the past we are able to detect and remove the majority of the memory conflicts.

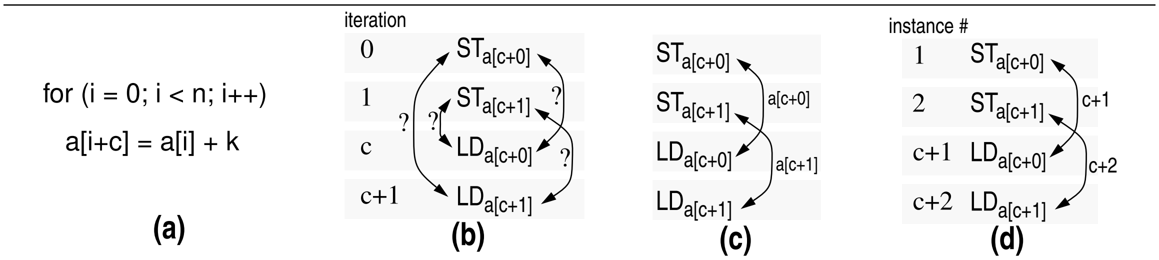

Figure 3. *Example code sequence that illustrates that multiple instances of the same static dependence can be active in the current instruction window.*

i+distance). Though both tagging schemes strive to provide unique tags, each may fall short of this goal under some circumstances.

4 Implementation Aspects

In this section, we describe possible implementations of the techniques to dynamically synchronize instruction pairs whose out-of-order execution is likely to result in a data misspeculation. We partition the support structures into two interdependent tables: a *memory dependence violation prediction table (MDVPT)* and a *memory dependence synchronization table (MDST)*. The MDVPT is used to identify 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. (We partition the support structures in this discussion, but there is no reason why a single structure could not be used if preferred.) As mentioned earlier, we restrict out discussion to memory dependences, though the structures described could easily be used for register dependences.

4.1 MDVPT

An entry of the MDVPT identifies a static dependence and provides a prediction as to whether or not subsequent dynamic instances of the corresponding static store-load pair result in a misspeculation (i.e., should the store and load instructions be synchronized). In particular, each entry of the MDVPT consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), and (4) 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 purpose of the prediction field is to capture in a reasonable way the past behavior of misspeculations for the instruction pair (in order to aid in avoiding future misspeculations). Though many options are possible for the prediction field, a discussion is postponed until later in this section.

4.2 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 MDVPT). 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) instance tag (INSTANCE), and (6) full/empty flag (F/E). The valid flag indicates if the entry is or is not in use. The load and store instruction address fields serve the same purpose as in the MDVPT. The load identifier uniquely identifies a dynamic instance of a load instruction. The instance tag field is used to distinguish between different dynamic instances of the same static dependence edge (using the data address of the storage location or the dependence distance between dynamic instances of the static store-load instruction pair as described in section 3). The full/empty flag provides the function of a condition variable.

4.3 Working Examples

The exact function and use of the fields in the MDVPT and the MDST are best understood by means of examples. Consider the following two working examples which explain the operation of the table structures. The first example (Figure 4) uses the data address of the storage location as an instance tag. The second example (Figure 5) uses the dependence distance between dynamic instances of the static store-load instruction pairs as an instance tag. For the working examples, assume that execution takes place on a processor which: (i) issues multiple memory accesses per cycle from a pool of load and store instructions and (ii) provides a mechanism to detect and correct misspeculations due to memory data speculation.

4.3.1 Using Data Address to Tag Dependence Edges

Consider the example in Figure 4 of a loop with the memory operations of three iterations active in the pool of load and store instructions. Each dynamic instance of the load and store instructions are shown numbered, and the true dependences are indicated as arrows connecting the corresponding instructions. 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 misspeculation (ST1-LD2 dependence) is detected, a MDVPT entry is allocated and the addresses of the load and the store instructions are recorded (action 1, part (b)). As a result of the misspeculation, instructions following the load are squashed and must be re-issued.

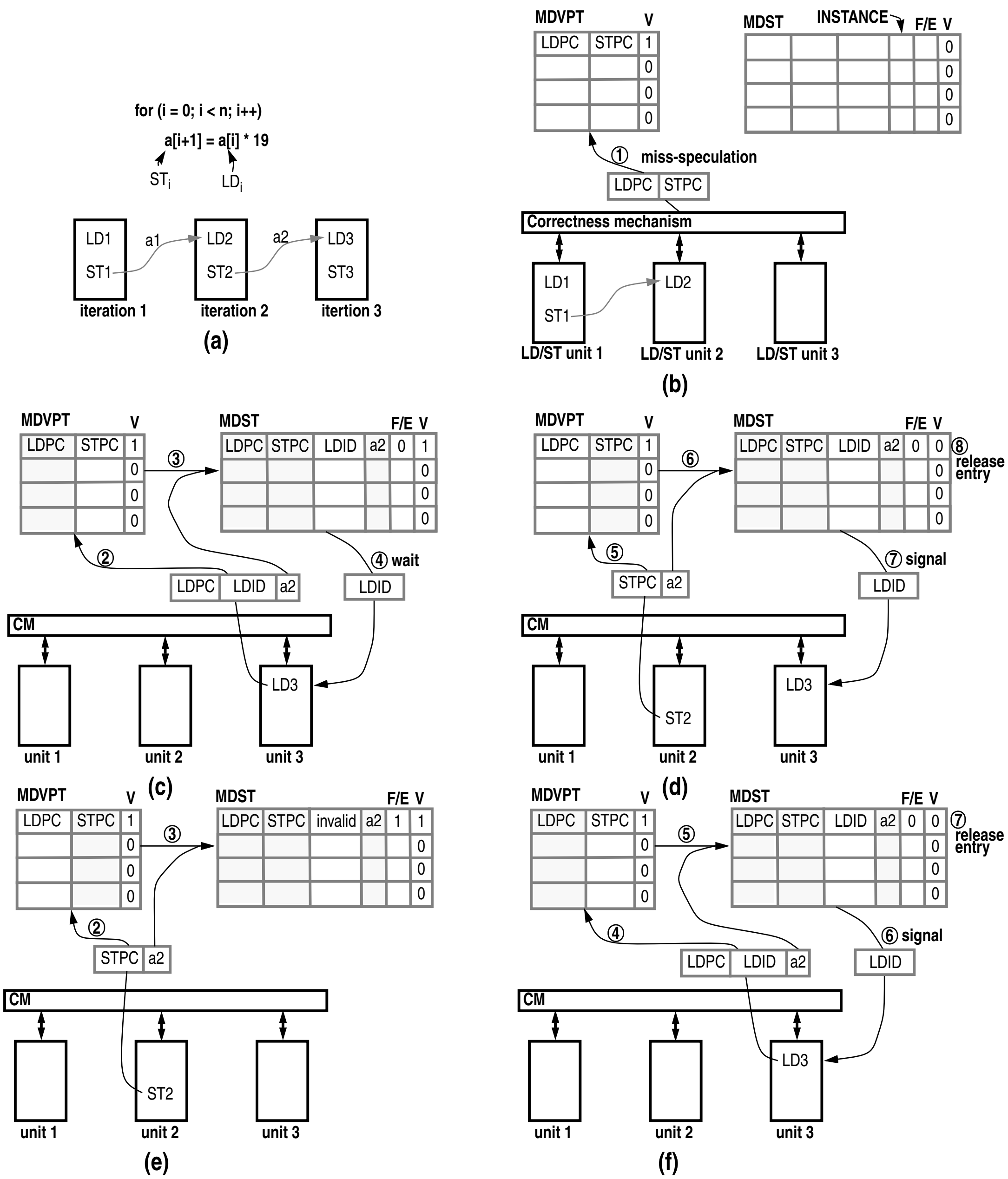

Figure 4. *Synchronization of memory dependences: using data addresses to distinguish instances of the same static dependence.*

As execution continues, assume that LD3 becomes ready to issue before ST2. When LD3 is ready to issue and execute, its instruction address, the data address of the access, and its assigned load identifier are sent to the MDVPT (action 2, part (c)). The instruction address of LD3 is matched against the contents of all load instruction address fields of the MDVPT (shown in grey). Since a match is found, the MDVPT inspects the entry predictor to determine if a synchronization is warranted. Assuming the predictor indicates a synchronization, the MDVPT allocates an entry in the MDST using the load instruction address, the store instruction address, and the data address of the access (action 3, part (c)) from the MDVPT. At the same time the full/empty flag of the allocated entry is set to empty, the MDST returns the load identifier to the load/store pool to indicate that the load must wait (action 4, part (c)).

When ST2 is ready to issue and execute, its instruction address and the data address of the access are sent to the MDVPT (action 5, part (d)). The instruction address of ST2 is matched against the contents of all store instruction address fields of the MDVPT (shown in grey). Since a match is found, the MDVPT inspects the contents of the entry and initiates a synchronization in the MDST. As a result, the MDVPT searches the MDST with a combination of the load instruction address, the store instruction address, and the data address (action 6, part (d)) to find the allocated synchronization entry. At the same time the full/empty field is set to full, 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 issues before LD3, it is unnecessary for LD3 to be delayed when it issues. 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 4. When ST2 is ready to issue and execute, it passes through the MDVPT as before with a match found (action 2, part (e)). Since a match is found, the MDVPT inspects the contents of the entry and initiates a synchronization in the MDST. However, no matching entry is found in the MDST since LD3 has yet to be seen. A new entry is allocated, and its full/empty flag is set to full (action 3, part (e)). When LD3 is ready to issue and execute, it passes through the MDVPT and determines a synchronization is warranted as before (action 4, part (f)). The MDVPT searches the MDST, where it 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)), and frees the MDST entry (action 7, part (f)).

4.3.2 Using Dependence Distance to Tag Dependence Edges

Consider the example in Figure 5 of a loop with the memory operations of three iterations active in the pool of load and store instructions. To use the dependence distance rather than the data address to tag dependence edges, an extra field is added to the MDVPT entry to record this value for store-load instruction pairs involved in a data misspeculation. Thus, as soon as a misspeculation is detected, the dependence distance between the offending store-load pair is recorded. In general, the synchronization occurs between store and load instructions, ST_i and $LD_{i+distance}$ respectively. The sequence of events and the steps in the synchronization process are nearly identical for using the dependence distance versus using the data address as an instance tag. The only notable difference, is that rather than search the MDST with the data address, the distance field of the MDVPT entry is used as indicated (action 6, part (d)) to search the MDST. For the sake of brevity, this repetitious description is omitted.

4.4 Other Issues

We now discuss a few other issues which relate to the implementations described above.

4.4.1 Intelligent Prediction

Upon matching a MDVPT entry, a determination must be made as to whether the instruction pair in question warrants synchronization. The simplest approach is to assume that any matching entry ought to be synchronized. However, this approach may lead to unnecessary delays in cases where store-load instruction pairs are usually active concurrently, but are misspeculated only some of the time. Instead, a more intelligent approach (perhaps borrowed from work on control dependence prediction) may be effective; any of the plethora of known methods (counters, voting schemes, adaptive predictors, etc.) to provide the intelligent prediction of control dependences may be applied to the prediction of data dependences, or entirely new prediction schemes developed. Regardless of the actual choice of mechanism, the prediction method ought to exhibit the quality that it strengthens the prediction when speculation succeeds and weakens the prediction when speculation fails.

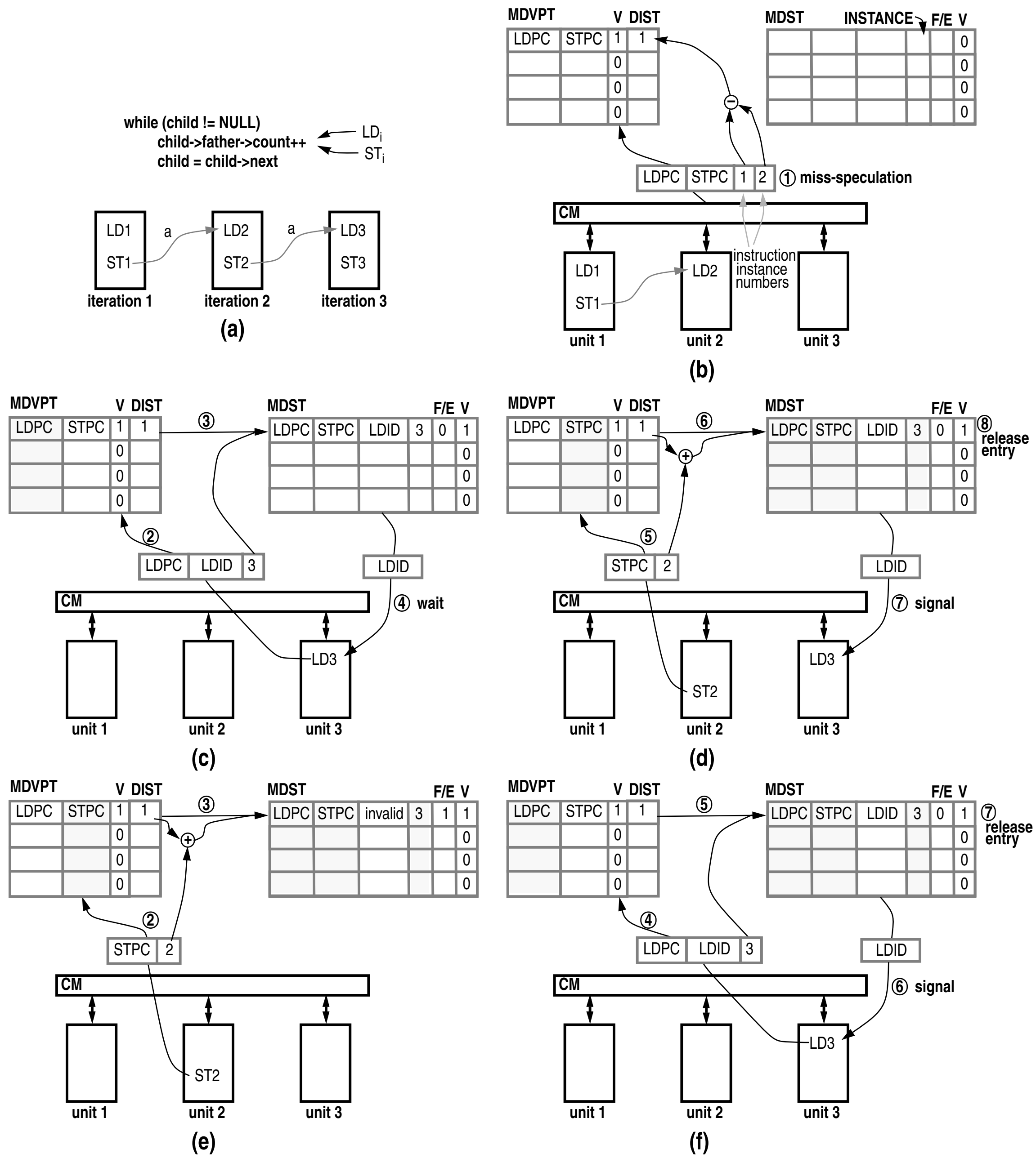

Figure 5. *Synchronization of memory dependences: using the distance of instruction instances to distinguish instances of the same static dependence.*

4.4.2 Incomplete Synchronization

So far, it has been assumed that any load which waits on the full/empty flag of an entry in the MDST eventually sees a matching store which signals to complete the synchronization. Since an MDVPT entry only provides a prediction, this expectation may not always be fulfilled. If this situation arises, the two main considerations are (i) to avoid deadlock and (ii) to free the MDST entry allocated for a synchronization that will never occur. The deadlock problem is easily solved, as it is reasonable to assume that a load is always free to execute once all prior stores are known to have executed. Likewise, in cases for which loads execute under the deadlock avoidance criteria described earlier, the load identifier may be used to free the MDST entry.

Under similar circumstances to those described above, a store may allocate an MDST entry for which no matching load is ever seen. Since stores never delay their execution, there is no deadlock problem in this case. However, it is still necessary to eventually free the MDST entry. Unfortunately, there is no execution condition[3], comparable to the deadlock avoidance criteria for the load that can be associated directly with the store. One possible solution is to free entries whose full/empty flag is set to full whenever an entry is needed and no table entries are not in use. Another possible solution is to allocate entries using random or LRU replacement, in which case entries are freed as needed.

4.4.3 Multiple Table Entry Matches

Although not illustrated in the examples, it is possible for a load or a store to match multiple entries of the MDVPT and/or of the MDST. This case represents multiple memory dependences involving the same static load and/or store instructions. A straightforward means to accommodate this case is to handle each entry individually, one after the other, as described above. Another viable approach is to ensure that a unique mapping with respect to loads, stores, or both loads and stores is maintained in the tables. If this situation is relatively uncommon, the adaptive nature of the prediction mechanism is likely to discard all but the most frequent misspeculations. If this situation is relatively common, a more aggressive approach that evaluates multiple entries simultaneously may be expedient.

4.4.4 Centralized Versus Distributed Structures

In our discussion, it has been assumed that the MDVPT and the MDST are centralized structures. However, as greater levels of instruction-level parallelism are exploited, greater numbers of concurrent memory accesses must be sustained. Under such conditions, the support structures are likely to play a key role in execution. Accordingly, it is important to assure that neither structure becomes a bottleneck. The most straightforward way to meet this demand is to multi-port the tables. While such an approach provides the needed bandwidth, its access latency and area grow quickly as the number of ports is increased. It is also possible to divide the table entries into banks indexed by the load and store instruction addresses. This solution is likely to be inadequate since temporal and spatial locality in instruction reference patterns may cause many conflicting bank accesses.

An alternative approach is to actually distribute the structures, with identical copies of the MDVPT and the MDST provided at each source of memory accesses (assuming multiple load/store queues, multiple load/store reservation stations, etc.). Each source of memory accesses need only use its local copy of the two tables most of the time. As soon as a misspeculation is detected, this fact is broadcast to all copies of the MDVPT, causing an entry to be allocated in each copy as needed. A load instruction uses both tables in the same manner as described earlier. A store instruction, on the other hand, behaves somewhat differently. In the event a match for a store is found in a local MDVPT, all identifying information for the entry is broadcast to all copies of the MDST. Each copy of the MDST searches its entries to find any allocated synchronization entry. The outcomes with respect to whether a match is or is not found are similar to those described earlier. In addition, any prediction update to an entry of a local MDVPT must be broadcast in order to maintain a similar view among all of the copies of this table.

5 Experimental Evaluation

In this section we evaluate the utility of the mechanisms proposed in the previous section. To do this, we require a processing model where dynamic data speculation is heavily used. As mentioned earlier, the superscalar model has not yet reached a point where data speculation is routine (or even considered worthwhile). Accordingly, we carry out our evaluation within the context of a Multiscalar processor.

A Multiscalar processor relies on a combination of hardware and software to extract parallelism from ordinary (sequential) programs. In this model of execution, the control flow graph (CFG) of a sequential program is partitioned into portions called tasks. These tasks may be neither control nor data independent. A Multiscalar processor sequences through the CFG speculatively, a task at a time, without pausing to inspect any of the instructions within a task. A task is assigned to one of a collection of processing units for execution by passing the initial program counter of the task. Multiple tasks execute in parallel on the processing units, resulting in an aggregate execution rate of multiple instructions per cycle. In this organization, the instruction window is bounded by the first instruction in the earliest executing task and the last instruction in the latest executing task. More details of the Multiscalar model can be found in [13, 15, 16, 17, 18].

In a Multiscalar processor, dependences may be characterized as *intra-task* (within a task) or *inter-task* (between individual tasks). The results herein are all simulated executions in which intra-task memory data dependences are not speculated, but inter-task memory data dependences are freely speculated. That is, misspeculations may only occur

3. Note that the MDVDT entry cannot be deallocated as soon as the store is retired since a later load may use it to do synchronization.

for store-load instruction pairs whose dependence edge crosses dynamic task boundaries. Furthermore, the results reflect execution with no compiler supported disambiguation of these memory dependences. This detail implies that even in cases where an unambiguous memory dependence exists, it is treated no differently than an ambiguous memory dependence during execution. At first glance, the reader may be tempted to conclude that the results of this section are not very useful since many dependences could be classified as unambiguous, even with a rudimentary compiler. However, this conclusion is not necessarily correct, and we elaborate on this next.

Like a superscalar processor, the goal of a Multiscalar processor is to execute a *sequential* program in *parallel*. In a sequential program, synchronization between operations is implicit: the specified order of the operations provides the synchronization. If the program were written with a partial order of execution in mind, synchronization between unambiguously-dependent operations would be provided by the software, using signal and wait operations on statically-named synchronization variables. However, this new program is not a sequential one any more, and all the problems involved in the static conversion of a totally-ordered (sequential) program into a partially-ordered (parallel) program persist. Moreover, the overhead associated with providing this explicit synchronization can be significant in terms of the extra named synchronization variables required, in terms of additional operations needed to perform the necessary synchronizations, as well as in terms of the unnecessary waiting time due to a conservative synchronization. If we start with a sequential program (with no explicit synchronization) any load operation which executes before a logically-preceding store (whose address is unknown) must be classified as an ambiguous, data speculative operation.

5.1 Methodology

The results presented in this paper have been collected on a simulator that faithfully represents a Multiscalar processor. The simulator accepts annotated big endian MIPS [26] instruction set binaries (without architected delay slots of any kind) produced by the Multiscalar compiler, a modified version of GCC 2.5.8. In order to provide results which reflect reality with as much accuracy as possible, the simulator performs all of the operations of a Multiscalar processor and executes all of the program code, except system calls, on a cycle-by-cycle basis. (System calls are handled by trapping to the OS of the simulation host.)

The programs studied in this work are taken from the SPECint92 benchmark suite (with inputs indicated in parentheses): *compress* (in), *espresso* (ti.in), *gcc* (integrate.i), *sc* (loada1), and *xlisp* (7 queens). Table I presents the dynamic (useful) instruction counts for the corresponding Multiscalar execution. (Only one version of a Multiscalar binary is created; the same Multiscalar binary is used for all the Multiscalar configurations in these experiments.) All benchmarks have been run to completion for the indicated input.

Benchmark Program	Useful Instructions
compress	73.38 M
espresso	595.88 M
gcc	72.99 M
sc	440.23 M
xlisp	247.56 M

Table I. *Dynamic Instruction Count per Benchmark*

Integer	Latency	Floating point	Latency
Add/Sub	1	*SP Add/Sub*	2
Shift/Logic	1	*SP Multiply*	4
Multiply	4	*SP Divide*	12
Divide	12	*DP Add/Sub*	2
Memory Store	1	*DP Multiply*	5
Memory Load	2	*DP Divide*	18
Branch	1		

Table II. *Latencies of functional units*

5.2 Configuration

This work evaluates Multiscalar processor configurations of 4 and 8 processing units with a global sequencer to orchestrate task assignment. The sequencer maintains a 1024 entry 2-way set associative cache of task descriptors. The control flow predictor of the sequencer uses a path based scheme which selects from 4 targets per prediction and maintains 7 path histories XOR-folded into a 15 bit path history register. The predictor storage is composed of both a task target table and a task address table, each with 32k entries indexed by the path history register. Each target table entry consists of a 2 bit counter and a 2 bit target. Each address table entry consists of a 2 bit counter and a 32 bit address. The control flow predictor includes a 64 entry return address stack.

The pipeline structure of a processing unit is a traditional 5 stage pipeline (IF/ID/EX/MEM/WB) which is configured with 2-way, out-of-order issue characteristics. (Thus the peak execution rate of a 4-unit configuration is 8 instructions per cycle). The instructions are executed by a collection of pipelined functional units (2 simple integer FU, 1 complex integer FU, 1 floating point FU, 1 branch FU, and 1 memory FU) according to the class of the particular instruction with the latencies indicated in Table II. A unidirectional, point-to-point ring connects the processing units to provide a communication path, with a 2 word width and 1 cycle latency between adjacent processing units. All memory requests are handled by a single 4-word split transaction memory bus. Each memory access requires a 10 cycle access latency for the first 4 words and 1 cycle for each additional 4 words, plus any bus contention.

Each processing unit is configured with 32 kilobytes of 2-way set associative instruction cache in 64 byte blocks. (An instruction cache access returns 4 words in a hit time of 1 cycle, with an additional penalty of 10+3 cycles, plus any bus contention, on a miss.) A crossbar interconnects the processing units to twice as many interleaved data banks. Each data bank is configured as 8 kilobytes of direct mapped data cache in 64 byte blocks with a 32 entry address resolution buffer, for a total of 64 kilobytes and 128 kilobytes of banked data storage as well as 256 and 512 address resolution entries for 4-unit and 8-unit Multiscalar processors respectively. (A data bank access returns 1 word in a hit time of 2 cycles, with an additional penalty of 10+3 cycles, plus any bus contention, on a miss.) Both loads and stores are non-blocking.

5.3 Results

For all results presented herein, we use the two table structures, MDVPT and MSDT, detailed in Section 4. Each table is fully associative and contains 64 entries.[4] It is assumed that each table is a centralized structure which provides as many ports as need for a particular Multiscalar processor configuration. For prediction purposes, an entry of the MDVPT contains a 3-bit up-down saturating counter which takes on values 0 through 7. The predictor uses a threshold value of 3 for prediction; values less than the threshold predict no misspeculation, and values greater than or equal predict misspeculation (and consequent synchronization). Each table maintains LRU information for purposes of replacement. An entry within a table may be allocated speculatively (without cleanup if bogus), but updates to the prediction mechanism within an entry only occur non-speculatively. All simulation runs are performed with the Multiscalar processor configurations described earlier.

Table III gives the values of useful instructions per cycle for simulation runs that use no prediction/synchronization (*NONE*) and perfect prediction/synchronization (*PERF*), as well as values for runs that use the two versions of the prediction/synchronization scheme described earlier, with the data address (*ADDR*) or the dependence distance (*DIST*) to provide instance tags. We see that there is always a difference between no and perfect prediction/synchronization, sometimes significant (as in the cases of espresso, sc, and xlisp). Furthermore, this difference becomes greater as we move from the 4 processing unit to the 8 processing unit configurations, since the instruction window which may be supported becomes greater. The two versions of the proposed scheme perform quite well, with the data address approach slightly superior to the dependence distance approach. Nevertheless, there is much work to be done to close the gap between such heuristics and perfect (more so for an aggressive processor configuration). (The poor showing for compress is likely attributable to loads upon which the prediction/synchronization mechanism imposes unnecessary delays. At this point, this behavior is still under investigation.)

Benchmark	**Useful Instructions / Cycles**							
	4-Unit				**8-Unit**			
	NONE	*ADDR*	*DIST*	*PERF*	*NONE*	*ADDR*	*DIST*	*PERF*
compress	1.42	1.44	1.44	1.47	1.73	1.61	1.61	1.90
espresso	2.15	2.67	2.68	2.69	2.61	3.68	3.67	3.77
gcc	1.67	1.69	1.69	1.71	1.81	1.87	1.86	1.99
sc	2.12	2.21	2.14	2.22	2.43	2.70	2.55	2.76
xlisp	1.70	1.90	1.90	1.92	1.93	2.36	2.36	2.40

Table III. *IPC with real control prediction*

4. The results of this section are intended to be in support of a new concept, and are not intended to be exhaustive. We make no attempt to vary the parameters of the MDVPT and MSDT; these will be the subject of future work.

Table IV gives the values of misspeculations per useful load for the same scenarios as Table III. We see that the relative differences between *NONE* and *PERF* are significantly greater with regard to this metric, even in cases where the relative differences of useful instructions per cycle are meager. Specifically, the relative differences in useful instructions per cycle are of the order of a few (tens) of percent, whereas the relative differences in misspeculations per useful load are in the range of one or two orders of magnitude. The proposed prediction/synchronization scheme reduces the number of misspeculations to less than 1% of useful loads in nearly all cases. However, a decrease in such misspeculations does not translate directly into a proportionate increase in performance. The main cause is twofold. First, the synchronized instructions may only represent a shift of cycles from loss time to stall time in the overall picture of execution. Second, as the synchronization is only a prediction, it is possible that unnecessary delays are imposed on instructions which otherwise have no memory data dependence and/or incur no misspeculations.

Benchmark	**Misspeculations / Useful Loads**					
	4-Unit			**8-Unit**		
	NONE	*ADDR*	*DIST*	*NONE*	*ADDR*	*DIST*
compress	.0712	.0091	.0091	.1317	.0085	.0090
espresso	.0226	.0005	.0003	.0268	.0041	.0031
gcc	.0194	.0079	.0074	.0307	.0174	.0168
sc	.0210	.0023	.0031	.0417	.0074	.0081
xlisp	.0359	.0008	.0004	.0437	.0007	.0008

Table IV. *Misspeculations with real control prediction*

In Tables V and VI, we present results for the same Multiscalar processor configurations, but we substitute perfect control flow prediction for the real control flow prediction. Though the results are similar to those presented above and may not be realizable in practice, the purpose of including them is to demonstrate that the problem of data misspeculation persists even in the presence of more accurate dynamic windows. In some cases, a more accurate dynamic window aggravates the problem, especially in going from the 4 processing unit to the 8 processing unit configuration. The trends in useful instructions per cycle and misspeculations per useful load are analogous to those with real control flow prediction. However, the gap between the versions of the prediction/synchronization scheme proposed in this work and perfect prediction/synchronization widens with perfect control flow prediction, indicating room for improvement in the configurations studied, as other factors contributing to performance loss are tackled.

Benchmark	**Useful Instructions / Cycles**							
	4-Unit				**8-Unit**			
	NONE	*ADDR*	*DIST*	*PERF*	*NONE*	*ADDR*	*DIST*	*PERF*
compress	1.79	1.84	1.84	1.87	2.30	2.37	2.37	2.78
espresso	2.24	2.88	2.90	2.90	2.79	4.13	4.15	4.23
gcc	2.03	2.05	2.05	2.06	2.55	2.63	2.63	2.70
sc	2.38	2.50	2.43	2.51	2.83	3.27	3.13	3.30
xlisp	1.94	2.12	2.12	2.12	2.48	2.84	2.85	2.86

Table V. *IPC with ideal control prediction*

Benchmark	**Misspeculations / Useful Loads**					
	4-Unit			**8-Unit**		
	NONE	*ADDR*	*DIST*	*NONE*	*ADDR*	*DIST*
compress	.0561	.0079	.0079	.1138	.0184	.0184
espresso	.0233	.0005	.0001	.0267	.0038	.0027
gcc	.0160	.0055	.0044	.0262	.0128	.0110
sc	.0178	.0013	.0011	.0392	.0044	.0036
xlisp	.0280	.0007	.0003	.0384	.0008	.0007

Table VI. *Misspeculations with ideal control prediction*

6 Summary and Concluding Remarks

This paper proposed and evaluated dynamic techniques to improve the accuracy of data speculation. While much research by academia and industry has been (and continues to be) focused on the accuracy of control speculation, none up to this point has considered the accuracy of data speculation. This lack of concern is likely due to the fact that establishing a window of instructions via control speculation logically precedes scheduling the instructions of the window via data speculation. As ILP processors continue to become more aggressive, we feel that the use of data speculation will become even more widespread, and techniques (especially dynamic ones) to improve the accuracy of speculation will become very important.

We proposed the concept of dynamic prediction/synchronization to improve the accuracy of data speculation and applied this approach to the problem of handling data dependences through memory. We proposed a scheme that monitors the past behavior of misspeculations, uses this information to predict if a future data speculation ought to take place, and adaptively synchronizes an instruction pair when out-of-order execution might result in a misspeculation. In our evaluation of versions of this scheme, we were able to eliminate considerable numbers of data misspeculations, in the range of one or two orders of magnitude. We found this reduction resulted in a significant performance boost in many cases.

In our opinion, this work represents only a first step towards improving the accuracy of data speculation. Though we have worked with the data speculation of memory dependences, these techniques are general and applicable (with minor modifications) to a range of other uses of data speculation (such as register dependences). We maintain that the accuracy of data speculation will become a very important issue in future processor designs. In the event this belief turns into a reality, we look forward to seeing future work in the area of data speculation accuracy carried out with as much rigor as has been the case in the area of control speculation accuracy.

Acknowledgments

We thank Todd Austin, Dionisios Pnevmatikatos and Jim Smith for their valuable comments and suggestions on improving this paper.

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. 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] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proc. ASPLOS VI*, pages 183–193, October 1994.

[2] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proc. ASPLOS V*, 1992.

[3] R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th Annual ACM Symposium on Principles of Programming Languages*, 1983.

[4] S. A. Mahlke, D. C. Liu, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. *MICRO-25*, pages 45–54, December 1992.

[5] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W. Hwu. Superblock formation using static program analysis. In *Proc. of MICRO-26*, pages 247–255, December 1993.

[6] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7), July 1981.

[7] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. 17th Annual Symposium on Computer Architecture*, pages 344–354, Seattle, WA, May 1990.

[8] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proc. ASPLOS V*, October 1992.

[9] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39-3:349–359, March 1990.

[10] W. W. Hwu and Y. N. Patt. Checkpoint Repair for High-Performance Out-of-Order Execution Machines. *IEEE Transactions on Computers*, C-36(12):1496–1514, December 1987.

[11] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[12] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678,

May 1989.

[13] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transcactions on Computers, forthcoming*.

[14] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proc. 21st Annual Symposium on Computer Architecture*, pages 200–210, May 1994.

[15] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, November 1993.

[16] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. 22nd Int. Symposium on Computer Architecture*, pages 414–425, 1995.

[17] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proc. MICRO-27*, pages 181–190, December 1994.

[18] B. Case. *What's next for Microprocessor Design*. Microprocessor Report, October 1995.

[19] H. Su and P. Yew. On data synchronizations for multiprocessors. In *Proc. 16th Annual Symposium on Computer Architecture*, pages 416–423, May 1989.

[20] S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[21] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[22] *R10000 Microprocessor, product overview*. MIPS Technologies incorporated, October 1994.

[23] *PowerPC 620 RISC Microprocessor Technical Summary*. IBM Order number MPR620TSU-01, Motorola Order Number MPC620/D, October 1994.

[24] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *1995 IEEE CompCon*, pages 123–128.

[25] J. L. Peterson and A. Silberschatz. *Operating System Concepts, second edition*. Addison-Wesley Publishing Company, 1985.

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