Appears in the Proceedings of the 24th Annual International Symposium on Computer Architecture

Dynamic Speculation and Synchronization of Data Dependences

Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton St.
Madison, WI 53706
{moshovos, breach, vijay, sohi}@cs.wisc.edu

Abstract

Data dependence 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 dependence speculation succeeds; if not, it fails, and the two instructions must be synchronized. The modern dynamically scheduled processors that use data dependence speculation do so blindly (i.e., every load instruction with unresolved dependences is speculated). In this paper, we demonstrate that as dynamic instruction windows get larger, significant performance benefits can result when intelligent decisions about data dependence speculation are made. We propose dynamic data dependence speculation techniques: (i) to predict if the execution of an instruction is likely to result in a data dependence mis-speculation, and (ii) to provide the synchronization needed to avoid a mis-speculation. 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 may take two forms: control speculation and data speculation. Control speculation implies the execution of an instruction before the execution of a preceding instruction on which it is control dependent. Data speculation implies the execution of an instruction before the execution of a preceding instruction on which it *may be* or *is* data dependent.

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) is required if we want to consider instructions from more than one basic block for possible issue. Given the sizes of basic blocks, the need to go beyond a basic block became apparent some time ago, and several techniques to permit control speculation were developed, both in the context of statically- and dynamically-scheduled machine models. Improving the accuracy of control speculation (especially dynamic techniques) via the use of better branch prediction has been the subject of intensive research recently; a plethora of papers on dynamic and static branch prediction techniques have been published.

Data speculation has not received as much attention as control speculation. The two forms of data speculation that have received some attention are data value speculation and data dependence speculation. In data value speculation an attempt is made to predict the data value that an instruction is going to produce [15,19]. In data dependence speculation, no explicit attempt is made to predict data values. Instead, a prediction is made on whether the input data value of an instruction has been generated and stored in the corresponding named location (memory or register).

Most of the research on data dependence speculation has focused on ensuring correct execution while carrying out this form of speculation [8,9,10,18] and on static dependence analysis techniques [1,2,5,6,21]. So far, no attention has been given to dynamic techniques to improve the accuracy of data dependence speculation. This is because in the small instruction window sizes of modern dynamically scheduled processors [12,11,14], the probability of a mis-speculation is small, and furthermore, the net performance loss that is due to erroneous data dependence speculation is small.

In this paper, we argue that as dynamically-scheduled ILP processors are able to establish wider instruction windows, the net performance loss due to erroneous speculation can become significant. Accordingly, we are concerned with dynamic techniques for improving the accuracy of data dependence speculation while maintaining the performance benefits of aggressive speculation. We propose techniques that attempt: (i) to predict those instructions whose immediate execution is going to violate a true data dependence, and (ii) to delay the execution of those instructions only as long as is necessary to avoid the mis-speculation. A preliminary evaluation of the ideas presented in this paper was first reported in [17].

The rest of this paper is organized as follows: First, in section 2 we review data dependence speculation and discuss how it affects ILP execution. Then in section 3, we discuss the components of a method for accurate and aggressive memory data dependence speculation, while in section 4, we present an implementation framework for this method. In section 5, we provide experimental data on the dynamic behavior of memory dependences and present an evaluation of an implementation of the method we propose within the context of a Multiscalar processor [3,4,7,20]. Finally, in section 6 we list what, in our opinion, are the contributions of this work and offer concluding remarks. In the discussion that follows we are concerned with data dependence speculation; accordingly, we use the terms data dependence speculation, data speculation, and speculation interchangeably.

2 Data Dependence Speculation

Programs are written with an implied, total order. As a program executes, data values are produced and consumed by its instructions. These values are conveyed from the producer to the consumer by binding the value to a named storage location, namely registers and memory.

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 subset into a partial order. This is done so that instructions may execute in parallel and/or in an execution order that might be different from the total order. The shape of the

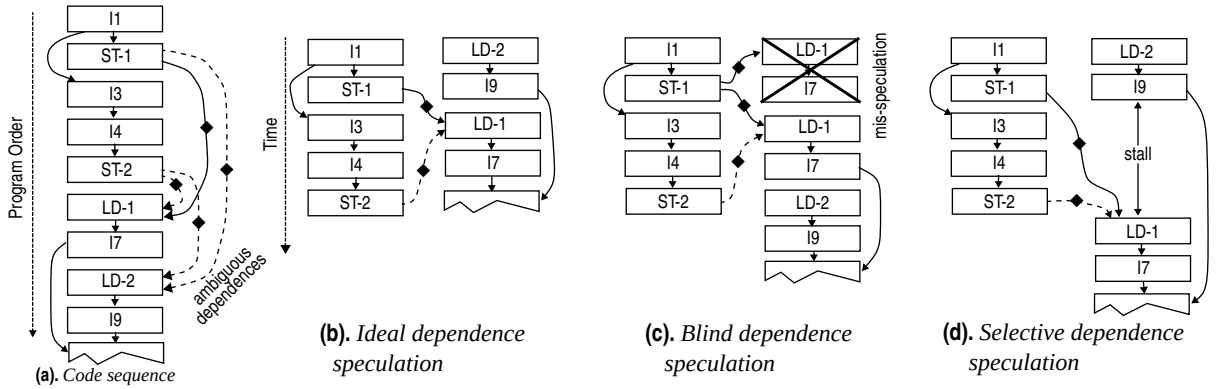

Figure 1. *Data dependence speculation examples. Arrows indicate dependences. Dependences through memory are marked with diamonds. Dotted arrows indicate ambiguous dependences that are resolved to no dependence during execution.*

partial order and the parallelism so obtained are heavily influenced by the dependences that exist between the instructions in the total order. Dependences may 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). During execution, an ambiguous dependence gets resolved to either a true dependence, or to no dependence.

To maintain program semantics, a producer/consumer instruction pair that is linked via a true dependence has to be executed in the order implied by the program. However, any execution order is permissible if the two instructions are linked via an ambiguous dependence that gets resolved to no dependence. This latter case represents an opportunity for parallelism and hence for higher performance. Unfortunately, the mere classification of a dependence as ambiguous implies the inability to determine whether a true dependence exists without actually executing the program. It is for this reason that ambiguous dependences may obscure some of the parallelism that is available. This problem is most acute in the case where the production and consumption of data is through memory. Thus, in this paper, we restrict our discussion to memory dependences even though all the concepts we present could easily be applied to the speculation of register dependences.

To expose the parallelism that is hindered by ambiguous dependences, data dependence speculation may be used. In data dependence speculation, a load is allowed to execute before a store on which it is ambiguously dependent. If no true dependence is violated in the resulting execution, the speculation is successful. If, however, a true dependence is violated, the speculation is erroneous (i.e., a mis-speculation). 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 [7,8,9,10,16,18].

Though data dependence speculation may improve performance when it is successful, it may as well lead to performance degradation because a penalty is typically incurred on mis-speculation. Consequently, to gain the most out of data dependence speculation we would like to use it as aggressively as possible while keeping the net cost of mis-speculation as low as possible.

The modern dynamically-scheduled processors that use data dependence speculation [11,12,14] do so blindly (i.e., a load is speculated whenever possible). No explicit attempt is made to reduce the net cost of mis-speculation. The reasons are simply that, in this environment, mis-speculations are extremely infrequent, and the cost incurred on mis-speculation is low. Both phenomena are directly attributable to the window sizes that these processors can establish (these are limited to a few tens of instructions in the best case). As window sizes grow larger, however, we argue that minimizing the net cost of mis-speculation becomes important. Under these new conditions, the mis-speculations become more frequent, and the cost of mis-speculations becomes relatively high.

To minimize the net cost of mis-speculation, while maintaining the performance benefits of speculation, we may attempt: (i) to minimize the amount of work that is lost on mis-speculation, (ii) to reduce the time required to redo the work that is lost on mis-speculation[1], or (iii) to reduce the probability of mis-speculation (or, in other words, to reduce the absolute number of mis-speculations). In this work we pursue the third alternative. We elaborate on this in the next section.

3 Components of a Solution

The ideal data dependence speculation mechanism not only avoids mis-speculations completely, but also allows loads to execute as early as possible. That is, loads with no true dependences (within the instruction window) execute without delay, whereas loads that have true dependences are allowed to execute only after the store (or the stores) that produces the necessary data has executed. Equivalently, loads with true dependences are synchronized with the store (or the stores) they depend upon. It is implied that the ideal data dependence speculation mechanism has perfect knowledge of all the relevant data dependences.

An example of how the ideal dependence speculation mechanism affects execution is shown in figure 1. In part (b), we show how the code sequence of part (a) may execute under ideal data dependence speculation as compared to when speculation is used blindly, part (c). 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 indicated by the diamond marked 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, whereas *LD-1* is forced to synchronize with *ST-1*.

In contrast to what is ideally possible, in a real implementation, the relevant data dependences are often unknown. Therefore, if we are to mimic the ideal data dependence speculation mechanism, we have to attempt: (i) to predict whether the immediate execution

1. One such technique is Dynamic Instruction Reuse [19].

of a load is likely to violate a true data dependence, and if so, (ii) to predict the store (or stores) the load depends upon, and, (iii) to enforce synchronization between the dependent instructions. However, since this scheme seems elaborate, it is only natural to attempt to simplify it. One possible simplification is to use *selective* data dependence speculation, i.e., carry out only the first part of the (ideal) 3-part operation. In this scheme the loads that are likely to cause mis-speculation are not speculated. Instead, they wait until the data addresses of all preceding stores, that have not yet executed, are known to be different; explicit synchronization is not performed. (We use the term *selective data dependence speculation* to signify that we make a decision on whether a load should be speculated or not. Loads with dependences are not speculated at all, whereas loads with no dependences can execute freely. In contrast, in ideal dependence speculation, we make a decision on when is the right time to speculate a load.) An example of how selective speculation may affect execution is shown in part (d) of figure 1. In this example, *LD-2* is speculated, whereas *LD-1* is not, since the prediction correctly indicates that *LD-2* has no true dependences while *LD-1* does. However, with this scheme, and due to the lack of explicit synchronization, a load may be delayed more than necessary (*LD-1* waits for *ST-2* also). In practice, and as we demonstrate in the evaluation section, selective data dependence speculation can lead to inferior performance when compared to blind speculation (part (c) of figure 1) even when perfect prediction of dependences is assumed. Even though other simplifications to the 3-part ideal operation may be possible, in this paper we restrict our attention to dependence speculation schemes that attempt to mimic the ideal data dependence speculation system. We do so because our primary goal is to demonstrate the potential of dynamic dependence speculation and synchronization mechanisms, rather than to perform a thorough evaluation of a variety of mechanisms.

To mimic the ideal data dependence speculation system, we need to implement all the 3 components of the ideal system as described before. That is, we must: (i) dynamically identify the store-load pairs that are likely to be data dependent (i.e., the dependences that are likely to cause mis-speculation), (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 is not an option that we consider desirable, or even practical. Fortunately, our empirical observations suggest that the following phenomena exists: *the static store-load instruction pairs that cause most of the dynamic data mis-speculations are relatively few and exhibit temporal locality* (we present empirical evidence in section 5). That is, at any given time, different dynamic instances of a few 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 mis-speculations. 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 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.

The above concept is illustrated in the example of figure 2

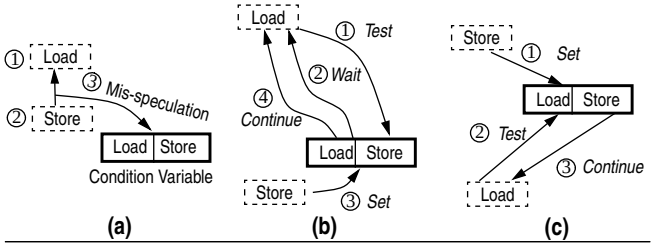

Figure 2. *Synchronization example*

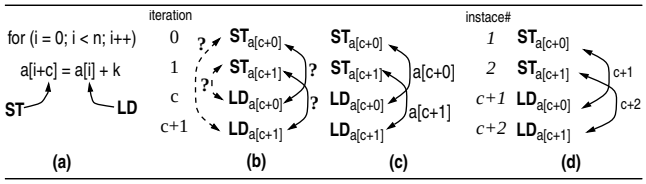

Figure 3. *Example code sequence that illustrates that multiple instances of the same static dependence can be active in the current instruction window. In parts (b), (c), and (d), the relevant store and load instructions from four iterations of the loop of part (a) are shown.*

where we assume that some means exist 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 mis-speculation 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 mis-speculation 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 the dependence prediction correctly indicates that a dependence exists but fails to detect that the order of execution has changed (most likely in response to external events whose behavior is not easy or desirable to track and predict, such as cache misses or resource conflicts). Synchronization is desirable even in this case since, otherwise, the 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 dependence edge may be specified using the (full or part of)

instruction addresses (PCs) of the store-load pair in question. Unfortunately, as exemplified by the code sequence of figure 3 part (b), using this information may not be sufficient to capture the actual behavior of the dependence during execution; the pair (PC_{ST}, PC_{LD}) matches against all four edges shown even though the dotted ones 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 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[2]. 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[3] (as may be seen for the example code, a dependence edge between ST_i and $LD_{i+distance}$ is tagged - in addition to the instruction PCs - with the value i+distance). Though both 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).

Since, our primary goal in this paper, is to introduce and evaluate novel mechanisms (and not to carry out a thorough analysis of a variety of options), we restrict our attention to the second scheme where the dependence distance is used to tag dependences.

4 Implementation Aspects

As we discussed in the previous section, in order to improve the accuracy of data dependence speculation, we attempt: (i) to predict dynamically, based on the history of mis-speculations, whether a store-load pair is likely to be mis-speculated and if so, (ii) to synchronize the two instructions. In this section, we describe an implementation framework for this technique. 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. We do so, since we believe that the crux of the proposed technique is better described when the support structures are considered in this fashion. However, it is possible and probably desirable in an actual implementation, to combine the prediction and the synchronization structures and/or to integrate them with other components of the processor. For example, a simple extension is to provide the synchronization functionality in the data cache or some other similar storage structure, so that both the data and the necessary synchronization are provided at the same point. Later in this paper, we describe the implementation of a single structure that provides both dependence prediction and synchronization and discuss its advantages and its limitations. However, since our goal is to demonstrate the utility of the proposed technique, we do not consider further integration or any other implementations.

4.1 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 mis-speculation (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 mis-speculation caused the allocation of the entry (if we were to use the data 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 mis-speculations for the instruction pair in order to aid in avoiding future mis-speculations or unnecessary delays. Many options are possible for the prediction field (for example an up-down counter or dependence history based schemes); a discussion is postponed until later in this section. The prediction field is optional since, if omitted, we can always predict that synchronization should take place.

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 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, a dynamic instance of a load or a store instruction respectively. The exact encoding of this field 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 instruction as its LDID or STID). The instance tag field is used to distinguish between different dynamic instances of the same static dependence edge (in the working example that follows we show how to derive the value for this field). The full/empty flag provides the function of a condition variable.

2. At this point we are not concerned with mechanisms that implement this functionality. However, note that only the difference between the instance numbers is relevant and not the absolute values. As we explain in the evaluation section, in Multiscalar we can approximate the instance numbers by using statically assigned stage identifiers. In a superscalar environment we may use a small associative pool of counters. Load and store instructions can then be numbered based on their PC as they are issued. To support invalidations due to mis-speculation, these counters will have to be treated as registers. Alternatively, a load (store) that has to synchronize, may perform a backward (forward) scan through the instruction window attempting to locate the appropriate store (load) instruction.

3. To aid understanding, this scheme can be viewed as a dynamic, run-time implementation of the linear recurrence dependence analysis done by compilers.

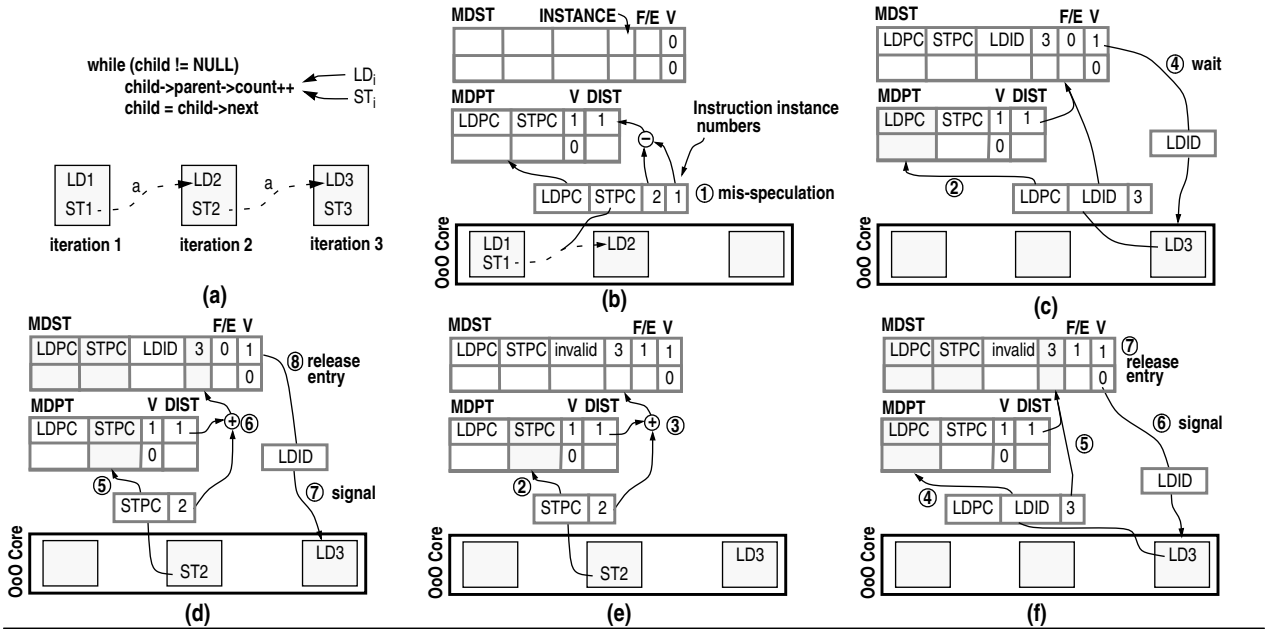

Figure 4. *Synchronization of memory dependences.*

4.3 Working Example

The exact function and use of the fields in the MDPT and the MDST are best understood by means of an example. In the discussion that follows we are using the working example of figure 4. For the working example, 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 mis-speculations 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 mis-speculation (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 a result of the mis-speculation, 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 3), 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 4. 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)).

4.4 Issues

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

4.4.1 Intelligent Prediction

Upon matching a MDPT 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 (i.e., the predictor field is optional). However, this approach may lead to unnecessary delays in cases where the store-load instruction pairs are mis-speculated only some of the time. Instead, a more intelligent approach may be effective. Any of the plethora of known methods (counters, voting schemes, adaptive predictors, etc.) used to provide the intelligent prediction of control dependences may be applied, with appropriate modifications, to the prediction of data dependences. 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.

4.4.2 Incomplete Synchronization

So far, it has been assumed that any load, that waits on the full/empty flag of an entry in the MDST, eventually sees a matching store that signals to complete the synchronization. Since an MDPT 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. At that point, the load identifier has to be send to the MDST where it is used to free the entry that was allocated for the particular load.The information recorded in the MDST entry can then be used to locate update the corresponding prediction entry in the MDPT

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, we cannot de-allocate this entry when the store retires (recall that in section 3 we explained that we would like to synchronize a store-load pair when the prediction indicates that we should, even if the execution order does not violate the dependence). A 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 Mis-speculations

In the event of control or data mis-speculation, it is desirable, although not necessary, to invalidate any MDST entries that were allocated to the instructions that are squashed. The LDID and the STID fields can be used to identify the entries that have to be invalidated.

Typically, many instructions continuous in the program order, are invalidated when a mis-speculation occurs. Thus, we may have to invalidate multiple MDST entries on mis-speculation. Fortunately, the MDST has to be notified only of those instructions that have entries allocated to them, which are typically going to be few. To support multiple invalidations per cycle, several options exist such as (i) providing multiple ports to the STID and LDID tags, or (ii) using a suitable encoding of the STID and LDID tags that would allow for the invalidation of a range of instructions. For example, we can use as many bits as the maximum number of simultaneously, unresolved control transfer instructions allowed. This encoding allows us to invalidate at a basic block granularity with an associative lookup.

4.4.4 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 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) store$_1$ M else store$_2$ M; load M*, there are two dependences *(store$_1$, load)* and *(store$_2$, load)*). There are several ways of addressing this issue.

A straightforward approach is to ensure, by means of the replacement and allocation policies, that a unique mapping with respect to both loads and stores is maintained in the tables. For example, in the MDPT we may allow a new entry to be created only after any pre-existing entries for the same static load or store are de-allocated. To maintain a unique mapping in the MDST, we may force a load or a store to stall and retry if there is another entry for either of the instructions that have to synchronize (alternatively we may de-allocate the pre-existing entry). This approach is acceptable when: (i) multiple dependences per static load and store are relatively uncommon or (ii) when the dynamic dependence pattern consists of long series during which only one of the many dependences is active for the most part. In both cases, the adaptive nature of the prediction mechanism is likely to discard all but the most frequent mis-speculations. If multiple dependences are relatively common, a more aggressive approach that evaluates multiple entries simultaneously is expedient. One approach is to support multiple stores per load or vice versa. This can be achieved by modifying the entries MDPT and MDST to include multiple fields for store PCs per load (or the other way around).

If multiple dependences are to be fully supported within the implementation framework we presented in this section, the following considerations must be addressed: (i) when multiple dependences are predicted from the MDPT, how to allocate multiple entries, one per predicted dependence, in the MDST, (ii) when synchronization happens on an MDST entry, how to determine whether the particular load has other entries it has to wait for, and (iii) when a store synchronizes simultaneously with many loads in the MDST, how to go about sending all the LDIDs. Again several options exist. For the purposes of this paper, we address all three considerations by combining the two tables into a single structure where each prediction entry carries with it a predefined number of synchronization entries (note that in this organization, the PCs of the instructions need not be recorded in each synchronization entry). We next explain how this organization addresses the aforementioned issues. Allocating multiple synchronization entries, each for a different prediction entry, is straightforward since the prediction and synchronization entries are now physically adjacent. To determine whether a load has other synchronization entries when a synchronization occurs, we do a second associative lookup using the load's LDID. If no other entries are found the load is allowed to continue execution. Finally, when multiple loads are simultaneously synchronized, we allow only up to a predefined number of them to do so at any given cycle (selecting the loads to wake up among those that have been signalled is no different than selecting the instructions to execute from those that are ready in an OoO processor).

4.4.5 Centralized Versus Distributed Structures

So far it has been assumed that the MDPT 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. As a consequence, 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 MDPT 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 mis-speculation is detected, this fact is broadcast to all copies of the MDPT, 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 MDPT, 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 MDPT 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 present experimental evidence in support of our observations on the dynamic behavior of memory dependences, and we evaluate the utility of the mechanism we proposed in the previous section. To do so, we require a processing model where dynamic data dependence speculation is heavily used and where the dynamic window size is relatively large. One processing model that satisfies both requirements is the Multiscalar processing model [7, 20]. Accordingly, we use various configurations of Multiscalar processors for most of the experiments we perform. However, for some of our experiments we use an unrealistic OoO execution model. We do so in order to demonstrate that our observations on the dynamic behavior of memory dependences are not specific to the Multiscalar processing model.

The unrealistic OoO execution model we use corresponds to a processor that is capable of establishing a perfect, continuous window of a given size. Under this model and for a window size of *n*, a load is always mis-speculated if a preceding store, on which it is data dependent, appears within less than *n* instructions apart in the sequential execution order. This model represents the worst case scenario with respect to the number of mis-speculations that can be observed at run-time since it assumes that every dependence that is visible from within the given instruction window is mis-speculated. We use this model not only to show that our observations about the dynamic behavior of memory dependences hold even under these extreme conditions, but also, to provide some insight on how the number of possible mis-speculations and dependences varies as a function of the dynamic window size.

To demonstrate the utility of the proposed mechanisms, we simulate various configurations of a Multiscalar processors. 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 control and data dependent. 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 [3,4,7,8,20].

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, mis-speculations may only occur 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.

The goal of any OoO execution processor, be it superscalar or Multiscalar, is to execute a sequential program in parallel. In doing so, any processor of this kind, dynamically converts the sequential program order into a parallel execution order. In this environment, the only condition that prevents the OoO execution of two instructions is the existence of a dependence that the OoO execution engine can detect without executing the instructions. This implies that even if the compiler knows that a particular memory dependence exists, nothing prevents the dynamic speculation of the corresponding load instruction. Consequently, to prevent the speculation of a dependence, the compiler has to identify by some means (for example through ISA extensions) that a load should not be speculated immediately and to enforce synchronization between unambiguously-dependent instructions (perhaps by using signal and wait operations on compiler generated synchronization variables or via full/empty bits). This is not a trivial task and futhermore, a program in which synchronization has been inserted is not a sequential program any more.

5.1 Methodology

The results we present have been collected on a simulator that faithfully represents a Multiscalar processor. The simulator accepts annotated big endian MIPS instruction set binaries (without architected delay slots of any kind) produced by the Multiscalar compiler, a modified version of GNU GCC 2.5.8 compiler (the SPECint95 benchmarks were compiled with the newest Multiscalar compiler which was built on top of GCC 2.7.2). 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.)

We performed the bulk of our experimentation with programs 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). However, to demonstrate the utility of the proposed data dependence speculation mechanism, we also report performance results (for one Multiscalar configuration) for

the SPECint95 and SPECfp95 suite. However, in order to keep the simulation time of the SPEC95 programs reasonable, we used either the train or the test input data sets (which sometimes are in the order of a few billion instructions). We used the train data set for the following programs: *099.go*, *129.compress*, *132.ijpeg*, *134.perl* (jumble), *147.vortex*, *101.tomcatv*, *110.applu*, *141.apsi*, *145.fpppp*, and *146.wave5*. For *124.m88ksim*, *126.gcc*, *130.li*, *102.swim*, *103.su2cor*, *104.hydro2d*, *107.mgrid*, and *125.turb3d*, we used the test data set. All programs, except 101.tomcatv, 125.turb3d, and 146.wave5, were ran to completion for the input used. Table 1 reports the dynamic, useful (i.e., committed), instruction counts for the corresponding Multiscalar execution. Only one version of a Multiscalar binary is created per benchmark; the same Multiscalar binary is used for all the Multiscalar configurations in these experiments. The Multiscalar binaries are also used by the unrealistic OoO execution model, however in this case, the Multiscalar specific annotations are ignored.

SPECint92		***SPECint95***		***SPECfp95***	
compress	73 M	*099.go*	595 M	*101.tomcatv*	2.8 G
espresso	596 M	*124.m88ksim*	496 M	*102.swim*	776 M
gcc	73 M	*126.gcc*	1.6 G	*103.su2cor*	1.4 G
sc	440 M	*129.compress*	39 M	*104.hydro2d*	1.2 G
xlisp	247. M	*130.li*	1.08 G	*107.mgrid*	5.9 G
		132.ijpeg	1.58 G	*110.applu*	675 M
		134.perl	2.37 G	*125.turb3d*	2.8 G
		147.vortex	3.04 G	*141.apsi*	2.9 G
				145.fpppp	511 M
				146.wave5	2.7 G

Table 1. *Dynamic instruction count per benchmark (committed instructions).*

5.2 Configuration

In this section we give the details of the Multiscalar processor configurations we used in our experimentation. We simulate Multiscalar processor configurations of 4 and 8 processing units (or stages) 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 the path based scheme described in [13]. The control flow predictor also 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 and with the latencies indicated in table 2. 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-stage and 8-stage 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.

Scalar	***Cycles***	***Scalar***	***Cycles***	***Floating-Point***	***Cycles***
Add/Sub	1	*Store*	1	*Add/Sub SP/DP*	2/2
Shift/Logic	1	*Load*	1	*Multiply SP/DP*	4/5
Multiply	4	*Branch*	1	*Divide SP/DP*	12/18
Divide	12				

Table 2. *Functional Unit Latencies ("SP/DP" stands for "Single/Double precision").*

5.3 Dynamic behavior of memory dependences

As we noted in section 3, the number of mis-speculations increases with the window size. Furthermore, the vast majority of the mis-speculations observed dynamically can be attributed to relatively few static dependences (store-load pairs) that exhibit temporal locality. In this section, we present experimental evidence in support of these observations. To do so, we simulate *data dependence caches*, or *DDCs*, of various sizes. A DDC of size *n*, records the data dependences that caused the *n* most recent mis-speculations. We count two events, hits and misses. These we define as follows: whenever a mis-speculation occurs we search through the DDC using the instruction PCs of the offending store and load instructions. If a matching entry is found, we count a hit, otherwise, we count a miss. A low data dependence cache miss rate implies that the relevant data dependences exhibit temporal locality.

In table 3, we report the number of mis-speculations observed under the unrealistic OoO model for various window sizes (WS column). As it can be seen, moving from a window of 8 instructions to a window of 32 instructions results in a dramatic increase in the number of mis-speculations. It is implied that most of the dynamic dependences are spread across several instructions (which may include many unrelated stores). This observation provides a hint to why selective data dependence speculation (i.e., not speculating the loads with dependences within the current window) may cause performance degradation when compared to blind speculation; when a dependence is spread across several, unrelated stores, it is often the case that it takes more time to wait until all the unrelated stores are resolved than to incur a mis-speculation and re-execute the load and the instructions that follow it.

In table 4, we show the number of static dependences that are responsible for 99.9% of all dynamic mis-speculations. Note that as the window size increases more static dependences are exposed. These newly exposed dependences may be far more frequent than the dependences observed when the window is smaller. This explains, for example, why in *compress* fewer dependences are responsible for the vast majority of mis-speculations when the window increases from 8 to 16 or 8 to 32. Finally, in table 5 we show the miss-rate of DDCs of 32, 128, and 512 entries. As it can be seen, even when all the dynamic dependences (that are visible from within the given instruction window) are mis-speculated, only a few static dependences cause most of the mis-speculations. Furthermore, DDCs of moderate size capture most of these dependences.

For the Multiscalar model we use two configurations, one with four stages and one with eight stages. The number of mis-speculations observed for these configurations are shown in table 6. As it

WS	compress	espresso	gcc	sc	xlisp
8	33	47.45 K	276.08 K	2.99 M	2.03 M
16	857.05 K	1.01 M	602.73 K	6.09 M	5.35 M
32	2.46 M	1.41 M	1.57 M	11.38 M	8.67 M
64	3.74 M	7.25 M	2.51 M	13.73 M	13.85 M
128	4.31 M	10.87 M	3.33 M	26.91 M	19.79 M
256	5.42 M	14.15 M	4.26 M	32.18 M	23.93 M
512	6.05 M	17.19 M	5.02 M	35.59 M	26.66 M

Table 3. *Unrealistic OoO model: Number of dynamic memory dependences observed as a function of window size (WS).*

WS	compress	espresso	gcc	sc	xlisp
8	6	33	255	46	31
16	2	104	704	127	73
32	5	291	1753	250	105
64	9	429	2967	380	167
128	18	848	4446	589	266
256	25	1500	6083	722	333
512	26	2021	8001	851	367

Table 4. *Unrealistic OoO model: number of static dependences responsible for 99.9% of all mis-speculations observed ("WS" stands for "window size").*

WS	CS	compress	espresso	gcc	sc	xlisp
8	32	18.1818	0.1317	2.2430	0.8489	0.0275
8	128	18.1818	0.1194	0.8645	0.0034	0.0027
8	512	18.1818	0.1194	0.1252	0.0034	0.0027
16	32	0.0028	0.3331	18.3372	5.5203	1.5507
16	128	0.0028	0.0190	1.5781	0.0445	0.0026
16	512	0.0028	0.0178	0.7249	0.0043	0.0026
32	32	0.0020	1.9247	33.0022	6.6796	4.8927
32	128	0.0020	0.1128	4.3829	1.7359	0.0033
32	512	0.0020	0.0343	1.0176	0.0045	0.0028
64	32	0.0034	2.3540	44.4893	11.1012	19.7762
64	128	0.0023	0.2453	10.5860	3.3251	0.1762
64	512	0.0023	0.0142	1.2652	0.0062	0.0028
128	32	0.0269	17.3775	52.1914	21.5784	37.8038
128	128	0.0027	0.6911	20.5244	3.7346	2.6479
128	512	0.0027	0.0389	2.2264	0.1043	0.0029
256	32	0.0274	20.8417	56.0067	29.7778	55.9566
256	128	0.0026	3.5985	27.4677	4.4675	5.2390
256	512	0.0026	0.1261	3.6096	0.3614	0.0034

Table 5. *Unrealistic OoO model: Miss-Rate (percentage) of DDC as a function of window size and DDC size. WS stands for "window size", and CS stands for "DDC size".*

Stages	compress	espresso	gcc	sc	xlisp
4	1.04 M	2.38 M	285 K	2.57 M	2.18 M
8	1.99 M	2.86 M	464 K	4.81 M	2.76 M

Table 6. *Multiscalar model: number of mis-speculations observed.*

can be observed, mis-speculations are more frequent when the window size is larger (8 stages as opposed to 4 stages). In table 7, we report the miss-rates of DDCs of various sizes for the 8-stage

CS	compress	espresso	gcc	sc	xlisp
1	50.970	16.68	85.20	73.74	68.29
2	31.470	11.18	75.59	28.16	39.82
4	11.240	9.74	62.96	16.22	18.77
8	0.660	4.28	45.84	14.45	4.40
16	0.020	1.10	31.93	6.46	1.59
32	0.002	0.36	18.05	2.96	0.31
64	0.002	0.10	8.92	0.88	0.01
128	0.002	0.05	4.55	0.17	.0004
256	0.002	0.03	3.16	0.02	.0004
512	0.002	0.02	2.93	0.01	.0004
1024	0.002	0.02	2.40	0.01	.0004

Table 7. *8-stage Multiscalar: DDC miss-rates (percentage) as a function of the DDC size ("CS" stands for DDC size).*

configuration only (i.e., for this experiment we use the configuration with the higher number of mis-speculations). As it can be seen, even a DDC of 64 entries exhibits a miss rate of less than 10% for all benchmarks. Furthermore, a DDC with 1024 entries captures virtually all static dependences for all benchmarks except gcc. For the Multiscalar model, we do not show measurements on the number of static dependences that cause most of the mis-speculations. That these dependences are few is implied by the results of the next section.

5.4 Comparison of dependence speculation policies

In this section we: (i) demonstrate that selective speculation may lead to inferior performance when compared to blind speculation and (ii) obtain an upper bound on the performance improvement that is possible through the use of the data dependence prediction and synchronization approach we described in section 3.

To do so, we compare the following four data dependence speculation policies: (i) NEVER, (ii) ALWAYS, (ii) WAIT, and (vi) PSYNC (for perfect synchronization). Under NEVER, no data dependence speculation is performed. Under ALWAYS, dependence speculation is used blindly (this is the policy used in the modern ILP processors that implement dependence speculation). Under policy WAIT, data dependence speculation is used selectively, that is loads with true dependences are *not* synchronized; instead they are forced to wait until the addresses of all previous stores are known to be different. Finally, under PSYNC, loads with no dependences execute as early as possible, whereas loads with true dependences synchronize with the corresponding stores. Policy PSYNC provides an upper limit on the performance improvement that is possible through the use of the mechanisms we presented in section 3. For PSYNC and WAIT we make use of perfect dependence prediction.

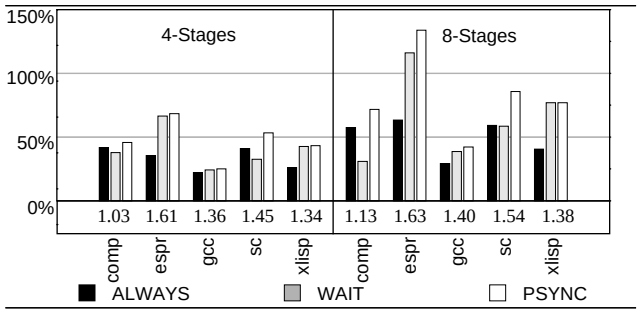

Figure 5. *Comparison of three data dependence speculation policies. Speedups (%) are relative to policy NEVER.*

In figure 5, we report (along the X-axis) the IPC of Multiscalar processor configurations that do not use data dependence speculation (policy NEVER) and the speedups obtained when policies ALWAYS, WAIT, or PSYNC are used instead. Since the dynamic window size is an important consideration we simulate Multiscalar configurations of four and eight stages. It can be observed that even blind data dependence speculation (policy ALWAYS) significantly improves performance in all cases. Furthermore, in contrast to when dependence speculation is not used, increasing the window size results in sizeable performance benefits.

Focusing on policy PSYNC, we can observe that it constantly improves performance over policy ALWAYS, sometimes significantly and furthermore, that the difference between PSYNC and ALWAYS becomes greater as the window size increases (8 stages compared to 4 stages). In addition, under policy PSYNC, increasing the window size typically results in higher performance. The results about policy PSYNC demonstrate that the technique we described in section 3 has the potential for performance improvements that are often significant (even when compared to blind speculation). Finally, selective data dependence speculation (policy WAIT) generally outperforms blind speculation (policy ALWAYS) and performs comparably to policy PSYNC in the 4-stage configuration for three of the benchmarks (espresso, gcc, and xlisp). However, for compress and sc, it performs worse than both PSYNC and ALWAYS (the cause of this phenomenon we explained in section 3, figure 1). As we move to larger windows (8 stages) the difference between PSYNC and WAIT becomes more significant for all benchmarks except xlisp.

5.5 Evaluation of the proposed mechanism

In the previous section we demonstrated the performance potential of our data dependence speculation technique. In this section we evaluate an implementation of this technique. The implementation we simulate is based on the mechanism we detailed in section 4. In this implementation, the MDPT and MDST are combined into a single structure where each MDPT entry carries as many MDST entries as there are stages. This implementation allows us to support multiple dependences per static store or static load as we explained in section 4. However, with this organization, only a single synchronization entry is allowed per static dependence and per stage. The simulated structure is a centralized, fully associative resource that provides as many ports as need for a particular Multiscalar processor configuration. For prediction purposes, each entry 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 dependence, and values greater than or equal predict dependence and consequent synchronization. We also maintain LRU information for purposes of replacement. An entry within the table may be allocated speculatively without cleanup if bogus, but updates to the prediction mechanism within an entry only occur non-speculatively when a stage commits. To distinguish between instances of the same static dependence we use a variation of the instance distance scheme which we discussed in section 3. In this scheme we approximate the instance numbers via the use of stage identifiers which are statically assigned to each stage. A load that is forced to synchronize on multiple dependences is allowed to execute only after all of them are satisfied. All simulation runs are performed for the Multiscalar processor configurations described earlier, and unless otherwise noted, the MDPT/MDST structure we simulate has 64 entries.

The results presented in this section are in support of a new concept. Consequently, our primary goal is to demonstrate the utility of the proposed mechanism. Though a thorough evaluation of the design space is highly desirable, it is not possible to include in this paper since the design space is vast, and the simulation method that is necessary (instruction driven, timing simulation) is extremely time consuming.

Even though we do not attempt an exhaustive evaluation of the design space, we do simulate two different dependence predictors which we refer to as (i) SYNC and (ii) ESYNC (the “E” is for enhanced). SYNC is our baseline predictor that uses an up/down saturating counter (as described in the beginning of this section). ESYNC, in addition to the up/down counter, also records for each dependence the PC of the task that issued the corresponding store instruction. Synchronization is enforced on a load that matches a MDPT entry only if the task PC of the stage at distance DIST (as recorded in the MDST entry) matches the task PC recorded in the predictor. This enhancement targets loads that have multiple static dependences which occur via different execution paths. In this case, the load does not have to wait for all the dependences, only for the dependence that lies on the proper execution path. However, since the task PC represents only minimal control path information, this predictor may fall short of its goal under some circumstances.

In the rest of this section, we first present and discuss results on the SPECint92 programs. We report the accuracy of the dependence prediction mechanism, the mis-speculation rate, and the performance improvement obtained. The speedups reported are relative to blind speculation (policy ALWAYS of section 5.4), which is the policy currently implemented in several modern processors. We later present and discuss results on the SPEC95 programs. For the latter programs, we report only performance numbers (due to space limitations).

In table 8, we report the breakdown of the dynamic dependence predictions for the SPECint92 programs. Since a load on which a dependence prediction is made may not necessarily have a dependence, a single number cannot be used to describe the accuracy of dependence prediction (in contrast to what is possible in control prediction). Instead, a dependence prediction has to be classified into one of four possible categories depending on whether a dependence is predicted and on whether a dependence actually exists. In the results shown, we count the dependence predictions done on loads that were either committed or have been issued from tasks that were squashed as a result of a dependence mis-speculation (we do not count predictions on loads that were squashed for other reasons). Predictions are recorded only once per dynamic load and at the time the load is ready to access the memory hierarchy. Furthermore, for those loads on which a dependence is predicted, the prediction is recorded after we have checked the synchronization entries for the first time. That is, in the case when a dependence is predicted, we count a “no dependence” outcome if a pre-existing, matching, synchronization entry allows the load to continue execution without delay, otherwise we count a “dependence” outcome. A dependence prediction is correct when the predicted and the actual outcomes are the same (rows *“N/N”* and *“Y/Y”*), otherwise the prediction is incorrect (rows *“N/Y”* and *“Y/N”*). An incorrect dependence prediction may result in mis-speculation (*“N/Y”*), or it may unnecessary delay the load (*“Y/N”*). We will refer to the latter predictions as *false dependence predictions*.

In table 9, we report how the mis-speculation frequency (defined as the number of mis-speculations over committed loads) improves when the proposed mechanism is in place for 4 and 8 stage configurations. In nearly all cases, the proposed prediction/synchronization mechanism reduces the number of mis-speculations by an order of magnitude. Furthermore, mis-speculations are typically reduced to less than 1% of useful loads. However, note that a decrease in the number of mis-speculations does not necessarily translate directly into a proportionate increase in performance (after all, if we did not use speculation, the mis-specula-

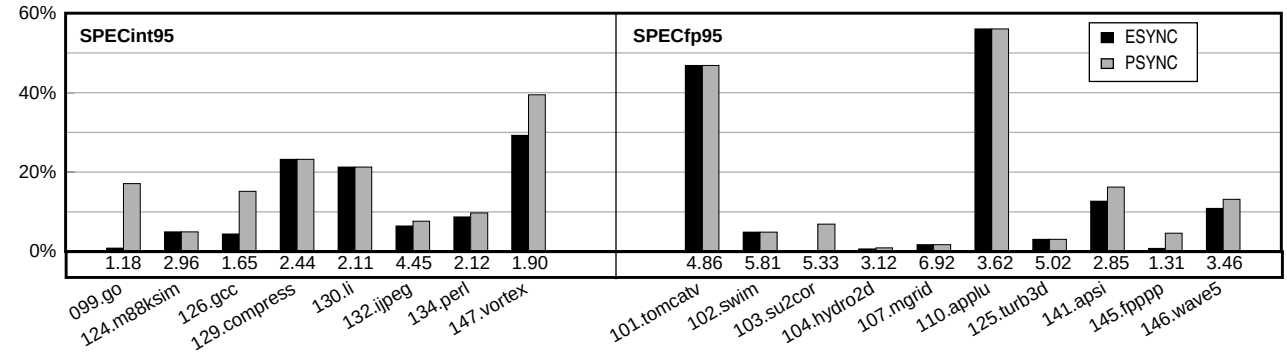

Figure 7. *Performance of our data dependence speculation mechanism for the SPEC95 programs. We simulate an 8-stage Multiscalar processor and we report speedups relative to blind speculation (policy ALWAYS) for the ESYNC predictor and for perfect dependence speculation (policy PSYNC). Along the X-axis we report the IPC obtained when the ESYNC predictor is in use.*

		P/A	compress	espresso	gcc	sc	xlisp
4-Stages	SYNC	N/N	81.62	98.62	95.56	97.19	95.99
		N/Y	0.18	0.02	1.38	0.26	0.08
		Y/N	4.17	0.03	0.80	0.14	0.06
		Y/Y	14.03	1.33	2.27	2.41	3.86
	ESYNC	N/N	85.79	98.63	96.06	97.40	96.07
		N/Y	0.09	0.01	1.20	0.24	0.00
		Y/N	0.30	0.03	0.07	0.08	0.00
		Y/Y	13.81	1.33	2.67	2.28	3.92
8-Stages	SYN	N/N	73.60	95.52	93.60	95.00	94.99
		N/Y	0.15	0.20	1.65	0.62	0.08
		Y/N	4.95	0.18	1.61	0.26	0.14
		Y/Y	21.31	4.09	3.15	4.11	4.79
	ESYNC	N/N	79.57	95.54	94.85	95.35	95.12
		N/Y	0.07	0.05	1.48	0.66	0.00
		Y/N	0.00	0.07	0.09	0.04	0.01
		Y/Y	20.37	4.34	3.58	3.95	4.87

Table 8. *Dependence prediction breakdown (%).“N” and “Y” stand for “No dependence” and “Dependence” respectively, whereas “P/A” stands for “Predicted/Actual”.*

	Policy	compress	espresso	gcc	sc	xlisp
4-Stages	ALWAYS	0.07312	0.02178	0.02007	0.02089	0.03556
	SYNC	0.00083	0.00016	0.00650	0.00271	0.00055
	ESYNC	0.00001	0.00013	0.00434	0.00250	0.00001
8-Stages	ALWAYS	0.13567	0.02613	0.03162	0.03891	0.04436
	SYNC	0.00205	0.00137	0.01428	0.00747	0.00069
	ESYNC	0.00004	0.00046	0.01159	0.00698	0.00002

Table 9. *Mis-speculations per committed load.*

tion rate would be zero). The main cause is twofold. First, the synchronized instructions may only represent a shift of cycles from time lost due to mis-speculations, to stall time in the overall picture of execution. That is, even though a load is not mis-speculated, there may be little other work to do while the load is waiting to synchronize. Second, false dependence predictions may impose unnecessary delays.

In figure 6, we show how the performance varies when our mechanism is in place, as compared to the base case Multiscalar processor that speculates all loads as early as possible (policy ALWAYS of figure 5). For almost all cases, the proposed mecha-

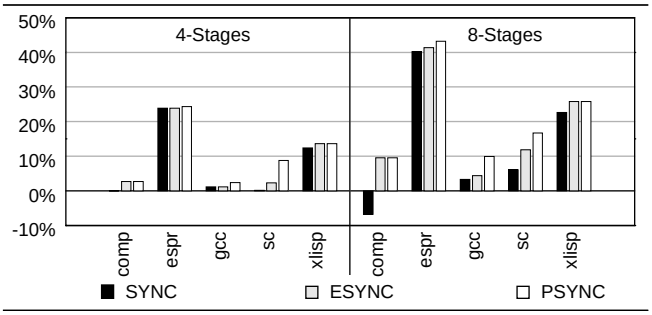

Figure 6. *Performance of our data dependence speculation mechanism on the SPECint92 programs. Speedups (%) are relative to policy ALWAYS.*

nism with the ESYNC predictor, not only improves performance, but also performs close to what is ideally possible (PSYNC columns). The SYNC predictor also improves performance for most of the programs. However the SYNC predictor never outperforms the ESYNC predictor. The SYNC predictor also offers virtually no performance improvement (over blind speculation) for compress and sc in the 4-stage configuration. Furthermore, performance degradation is observed for compress on the 8-stage configuration. False dependence predictions (“YN” marked rows in table 8) are responsible for this poor behavior. The counter based SYNC predictor fails to capture the data dependence patterns exhibited by this program. The ESYNC predictor, however, is able to successfully capture these patterns, since the dependences occur only via specific execution paths. There are two causes for the marked improvement demonstrated for espresso: (i) the average, dynamic task size is about 100 instructions, and (ii) most of the mis-speculations are the result of simple recurrences that occur most of the time (note however that the memory locations involved are often accessed via pointers). Consequently, for this program, the cost of mis-speculations is relatively high, whereas, even a simple up/down counter based predictor can capture the dynamic behavior of the most important dependences.

In figure 7, we report the performance results for the SPEC95 programs on an 8-stage Multiscalar processor. Along the X-axis we report the IPC obtained when our data dependence speculation/synchronization is used. The ESYNC bars represent the speedup obtained relatively to blind speculation (policy ALWAYS of section 5.4), whereas the PSYNC marked bars represent the speedup possible when ideal speculation and synchronization is used (policy PSYNC of section 5.4). Overall, our dependence speculation/synchronization mechanism improves performance, often significantly, for almost all the programs studied. Further-

more, our mechanism quite often performs close to what is ideally possible for the given configuration.

For the SPECint95 programs, the potential performance improvement is appreciable, ranging from 5% to almost 40%. For 124.m88ksim, 129.compress, and 130.li, our mechanism performs comparably to the ideal mechanism. Though the mechanism does not perform as well for 132.ijpeg, 134.perl, and 147.vortex, it does capture a significant amount of the gain that is possible. Nevertheless, both 099.go and 126.gcc fall short of this potential as compared to the ideal dependence speculation. The dependence patterns of these programs are quite irregular and exhibit relatively poor temporal locality as compared to the other programs. In particular, the performance of 099.go is limited by poor control prediction (even with the fairly sophisticated control prediction scheme used) and instruction supply.

For the SPECfp95 programs, most of the dependences we capture are loop recurrences. However, for 145.fpppp and 103.su2cor our mechanism is unable to synchronize some of the dependences. For these two programs, the size of the working set of dynamic dependences exceeds the capacity of our dependence prediction structures. Closer examination reveals that the instruction window established by 145.fpppp can grow to be as large as a few thousand instructions. (Most of the time is spent in a loop whose iterations execute each around 1000 instructions. With the greedy task partitioning policy currently used by the Multiscalar compiler, each iteration executes as a single task.) Tasks of similar size are also experienced in part of 103.su2cor. With the given instruction window size, it is no surprise that the working set of dependences is quite large. Increasing the size of the dependence prediction structures or breaking up each iteration to several tasks are two possible solutions. For 101.tomcatv and 110.applu, our mechanism performs very close to what is ideally possible. Our mechanism is also able to synchronize dependences that would otherwise cause performance degradation for 141.apsi and 146.wave5, but to a lesser extent. It should be noted that we simulated the first 2.8 billion instructions for 101.tomcatv and 146.wave5. Performance improves when these programs are simulated to completion. For example, at 10 billion instructions, the IPC for 101.tomcatv with the ESYNC mechanism is 5.68, whereas the IPC for 146.wave5 at completion (6.4 billion instructions) is 3.79. For 102.swim, 104.hydro2d, 107.mgrid, and 125.turb3d, there is little to be gained from dependence speculation and synchronization for the given configuration. For those programs, some other part of the processor (for example the functional units or the memory system) is saturated.

6 Implications and Conclusions

We make the following contributions in this paper:

- We demonstrate that, as the dynamic window sizes get larger, the net performance loss due to data dependence mis-speculations becomes significant.
- We identify three possible directions that can be followed to minimize this performance loss: (1) minimizing the work lost on mis-speculation, (2) minimizing the time required to redo this work, and (3) improving the accuracy of speculation.
- We observe that the static data dependences that are responsible for the majority of mis-speculations are few and dynamically exhibit temporal locality. The latter observation applies even when all dependences visible from within the dynamic instruction window are considered.
- We propose the concept of dynamic dependence prediction and synchronization and use it to reduce the net performance loss due to data dependence mis-speculation. We also identify the key issues involved in designing such data dependence speculation structures.
- We describe a microarchitectural technique that can be used to implement dynamic data dependence prediction and synchronization. Further, we demonstrate that for a specific OoO processor this technique can provide significant performance improvements. We finally identify most of and address some of the key design issues.

Our experimental results confirm the efficacy of the technique we propose. However, since this work introduces a new concept, we were not able to do a thorough evaluation of the design space and to explore many alternatives and other possible applications of the proposed technique. We believe that this fact does not diminish the importance of our results and observations. In our opinion, this work represents only a first step towards improving the accuracy of data dependence speculation and towards using dynamic dependence speculation and synchronization. Several directions for future research exist in improving the mechanisms we presented, in using the proposed technique in other processing models, and in using data dependence speculation in ways different than those we have discussed.

Though we have worked with memory dependences, the proposed techniques are general and applicable to a range of other uses of data speculation. Such uses include register dependences (this is mostly relevant to multiple program counter execution models such as Multiscalar) and value prediction (for example in a data speculation approach that uses value prediction only when dependences are likely to exist). We also believe that exposing the dependence prediction (MDPT) and/or the synchronization (MDST) structures to the compiler (perhaps via ISA extensions) opens new possibilities for statically orchestrated dependence speculation. (For example the synchronization variables can be allocated by the compiler to enforce synchronization of unambiguous dependences, whereas the prediction can be probed by the program during run-time to make on-the-fly decisions on when and which dependences to speculate.)

Even though in this work we considered fairly simple dependence predictors, any of the plethora of predictors used for branch prediction may be used, with appropriate modifications, to improve the accuracy of dependence prediction. Further improvement of our mechanisms may be possible by considering alternative dependence tagging schemes and synchronization primitives. Furthermore, it would be interesting to consider integrating the dependence prediction and synchronization structures with other components of the processor (for example, we may implement the synchronization functionality in the data cache or in a similar structure so that both the data and the necessary synchronization are provided from the same structure).

The techniques we proposed are applicable to processing models other than Multiscalar. However, further study is necessary, since differences in the instruction window size and in the granularity of checkpointing may influence the relative performance of various dependence speculation and synchronization schemes. We maintain that as ILP processors continue to become more aggressive, the use of data speculation will become even more widespread, and techniques (especially dynamic ones) to improve the accuracy of data dependence speculation, such as those proposed in this paper, will become important.

Acknowledgments

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.

We thank Jim Smith, Todd Austin, Andy Glew, Sridhar Gopal, Stefanos Kaxiras, and Dionisios Pnevmatikatos for their valuable comments and suggestions on this work.

References

[1] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–452, Oct. 1987.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Boston, MA: Kluwer Academic Publishers, 1988.

[3] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 181–190, Dec. 1994.

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

[5] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. Ph.D. thesis, Yale University, Feb. 1985.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 242–256, June 1994.

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

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

[9] 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, Oct. 1994.

[10] 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.

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

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

[13] Q. Jacobson, S. Bennett, N. Sharma, and J. Smith. Control Flow Speculation in Multiscalar Processors. In *Proc. 3rd Annual International Symposium on High-Performance Computer Architecture*, Feb. 1997.

[14] J. Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. *Digital Equipment Corp., Hudson, MA*, Oct. 1996.

[15] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, Dec. 1996.

[16] 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.

[17] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. A dynamic approach to improve the accuracy of data speculation. Technical Report 1316, Computer Sciences Dept., University of Wisconsin-Madison, March 1996.

[18] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[19] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proc. 24th Int. Symposium on Computer Architecture*, June 1997.

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

[21] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–12, June 1995.