Appears in the Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-2000), June 12-14, 2000.

Understanding the Backward Slices of Performance Degrading Instructions

Craig B. Zilles and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706-1685, USA
{zilles, sohi}@cs.wisc.edu

Abstract

For many applications, branch mispredictions and cache misses limit a processor's performance to a level well below its peak instruction throughput. A small fraction of static instructions, whose behavior cannot be anticipated using current branch predictors and caches, contribute a large fraction of such performance degrading events. This paper analyzes the dynamic instruction stream leading up to these performance degrading instructions to identify the operations necessary to execute them early. The backward slice (the subset of the program that relates to the instruction) of these performance degrading instructions, if small compared to the whole dynamic instruction stream, can be pre-executed to hide the instruction's latency. To overcome conservative dependence assumptions that result in large slices, speculation can be used, resulting in speculative slices.

This paper provides an initial characterization of the backward slices of L2 data cache misses and branch mispredictions, and shows the effectiveness of techniques, including memory dependence prediction and control independence, for reducing the size of these slices. Through the use of these techniques, many slices can be reduced to less than one tenth of the full dynamic instruction stream when considering the 512 instructions before the performance degrading instruction.

1 Introduction

Program performance is difficult to characterize. Programs do not perform uniformly well or uniformly poorly. Rather they have stretches of good performance punctuated by performance degrading events. The overall observed performance of a given program depends on the frequency of these events and their relationship to one another and to the rest of the program.

Program performance is measured by retirement throughput. Since retirement is sequential, the presence of a long latency instruction blocks retirement and degrades performance. The events we speak of, therefore, are these long latency, or performance degrading, instructions. There are many ways an instruction can degrade performance, but branch mispredictions and instruction and data cache misses account for the majority. Not surprisingly, microarchitectural techniques have focused on reducing the frequency and observed latency of these performance degrading events. Although frequency reduction techniques have been effective, they do not come close to eliminating the events completely. At the same time, more aggressive microarchitectures make the opportunity cost (in terms of instruction retirement opportunities) associated with an event that much greater.

A promising solution to this problem is to complement the frequency reduction techniques with a generic latency tolerance technique, like *pre-execution* [22]. In general, pre-execution amounts to guessing the existence of a future performance degrading instruction and executing it (or what we think it will be) some time prior to its actual encounter in the machine, thereby at least partially hiding its latency. In this paper, we are not concerned with a particular pre-execution mechanism but rather with the properties of such instructions and their relationship to the program that determine whether any pre-execution mechanism will be effective.

To be effective with respect to a given instruction, a pre-execution technique needs three things. First, at an *initiation point* ahead of the instruction's execution, the pre-execution technique needs to know that the performance degrading instruction *will* be executed. Second, it has to know which other instructions contribute to the performance degrading instruction. Finally, these contributing instructions must not comprise the entire program up to that point; otherwise, pre-execution is tantamount to normal execution and no latency hiding will be achieved.

The key to answering all of these questions lies in the backward slice of the performance degrading instruction. The backward slice comprises all of the instructions in the program that contribute, either directly or indirectly, to its computation, either through values or control decisions. Cast in terms of this definition, the key to pre-execution is to minimize the size of the backward slice from the initiation point to the performance degrading instruction, with respect to the size of the entire program over that same period.

Due to the prevalence of ambiguous control and data dependences, conservative construction of slices leads to slices that are comparable in size to the full program. At the other extreme, a slice can be reduced to an arbitrarily small size, but the ability to predict the behavior of the original program will be lost. We explore the region between these extremes, using speculation techniques to minimize the slice's size while maximizing its ability to accurately pre-execute an event. By observing program behavior, speculation can be applied only where it is likely to succeed.

In this paper, we focus on two issues. First, we perform an empirical analysis to determine the statistical nature of slices of performance degrading instructions. Then, we explore techniques for exploiting program structure to speculatively reduce the size of slices that are too big to support pre-execution.

We perform this analysis by extracting the dynamic backward slices from instruction traces leading up to instructions which frequently cause branch mispredictions or data cache misses. We classify the instructions in the slice based on the role they play: value, address, existence, and control flow. This classification is described in detail in Section 3.

Because of the differing natures of each component, each sub-slice is optimized in isolation. First (Section 5.1), the value sub-slice is shown to be small and close to the event except in cases where recurrences (discussed in Section 6.1) occur.

The size of the address sub-slice (studied in Section 5.2) exceeds that of the value sub-slice when built conservatively, but it can be optimized effectively by identifying stable memory dependences and removing address calculations through speculative register allocation. Memory dependences which cannot be treated in this way often unnecessarily contribute significant overhead to the slice; in many cases this overhead can be avoided by careful selection of where to initiate the pre-computation.

Control dependences are found in slices for two purposes. Those that dictate whether the performance degrading instruction will execute (which make up the existence sub-slice discussed in Section 5.3) are infrequent when control independence is exploited and can be removed if the branch is highly biased. Control dependences which resolve the dataflow (the control flow sub-slice which is presented in Section 5.4) can be substantial and these branches tend to be less biased than existence branches. Full optimization of the control flow sub-slice requires analysis to detect equivalent paths.

In Section 6, we briefly touch on some issues related to constructing slices as a whole before concluding in Section 7.

2 Background and Related Work

Limit studies [8, 26] have shown that, in the presence of a perfect memory system and fully resolved control flow, the available instruction level parallelism, even in integer programs, is often many multiples of what is necessary to saturate modern processors. However, perfect memory systems and predictors cannot be realistically built, and processors therefore tend to retire instructions at only a fraction of their peak rate due to branch mispredictions and problems with instruction and data value availability. By identifying the backward slices of instructions which contribute significantly to the CPI and pre-executing these slices, the performance impact of these events can potentially be reduced.

a)
```
1.  int i = 0;
2.  int sum = 0;
3.  while (i < 8) {
4.      sum += i;
5.      i ++;
    }
6.  print sum;
```

b)
```
1.  int i = 0;
2.  int sum = 0;
3.  while (i < 8) {
4.      sum += i;
5.      i ++;
    }
6.  print sum;
```

Figure 1. *The backward (a) and forward (b) slices for an example program using the value of i at statement 5 as the criterion.*

2.1 All Instructions are not Created Equal

Performance degrading events are not distributed evenly across static instructions. Previous studies [1] have shown that a fraction of static instructions are responsible for the majority of cache misses. Branches demonstrate a similar behavior; particular static branches are harder to predict than others. Some recently proposed branch prediction mechanisms exploit this by partitioning branches based on predictability and allocate more resources to hard-to-predict branches [7, 9]. Predicting the particular dynamic instances of these instructions that degrade performance has been shown to be possible with moderate accuracy [11, 16].

2.2 Program Slicing

Program slicing is a technique that was proposed as an aid for understanding programs, specifically during debugging [27]. It allows the user to focus on the portion of the program responsible for a particular phenomenon. In this section, we briefly touch on some of the major issues in slicing that relate to this paper; more details can be found in program slicing surveys [3, 24].

A slice is expressed with respect to a *criterion*, typically consisting of a value (or a set of values) and a position in the program. A slice contains the set of all statements which are related to the criterion. In this paper, the criterion is always a performance degrading instruction, and the terms will be used as synonyms. The backward slice consists of all statements which could affect the computation of the criterion, and a forward slice contains all statements which could be affected by the criterion. Figure 1 shows a small example program and its backward (Figure 1a) and forward (Figure 1b) slices. Pre-execution is concerned only with backward slices.

A static program slice contains all statements in the program that could affect any dynamic instance of the criterion. Dynamic program slices (the focus of this paper) consider a particular execution of the program (i.e., for a given input) and contain only those statements that affect a particular dynamic instance of the criterion.

2.3 Pre-execution

An obvious approach to tolerating latency is to initiate long latency operations early. Software memory pre-fetching has been successfully practiced for decades, especially in scientific applications. Techniques for pre-computing branch outcomes (including prepare-to-branch, and hardware techniques [10, 20]) have likewise been proposed. The general term *pre-execution* applies to all of these techniques, although the exact manifestation depends upon the particular technique. The composition of the slice used by a pre-execution technique depends both on the technique in use as well as the event being pre-executed.

When using pre-execution to prefetch instructions, the associated slice consists of only the operations which resolve control flow to the extent that we know whether or not a block of instructions is going to be executed; this corresponds to the *existence sub-slice* described in Section 3. These pre-fetches are non-binding, in that mis-speculation only causes cache pollution.

Data memory values can be similarly pre-fetched, but in this case the slice consists of the operations necessary to generate the cache block's address. These pre-fetches can also be non-binding. The inclusion of the existence sub-slice can reduce the number of unused pre-fetches.

Pre-execution of branches is like data memory pre-fetching in that the slice needs to compute the input operands of the branch in order to evaluate the branch. Unlike the previous two cases, this pre-executed branch outcome (and perhaps target) needs to be bound to a particular dynamic branch instance to fully benefit from the pre-execution. This process of binding is non-trivial [5, 10, 20, 22]. In general, it may be necessary to have a very accurate existence sub-slice to correctly correlate pre-executed branch outcomes with branches as they are fetched. In addition, since pre-executed branch outcomes override predicted outcomes, mis-speculations in pre-execution can translate into mis-predicted branches requiring the slice to be at least as accurate as the hardware predictor.

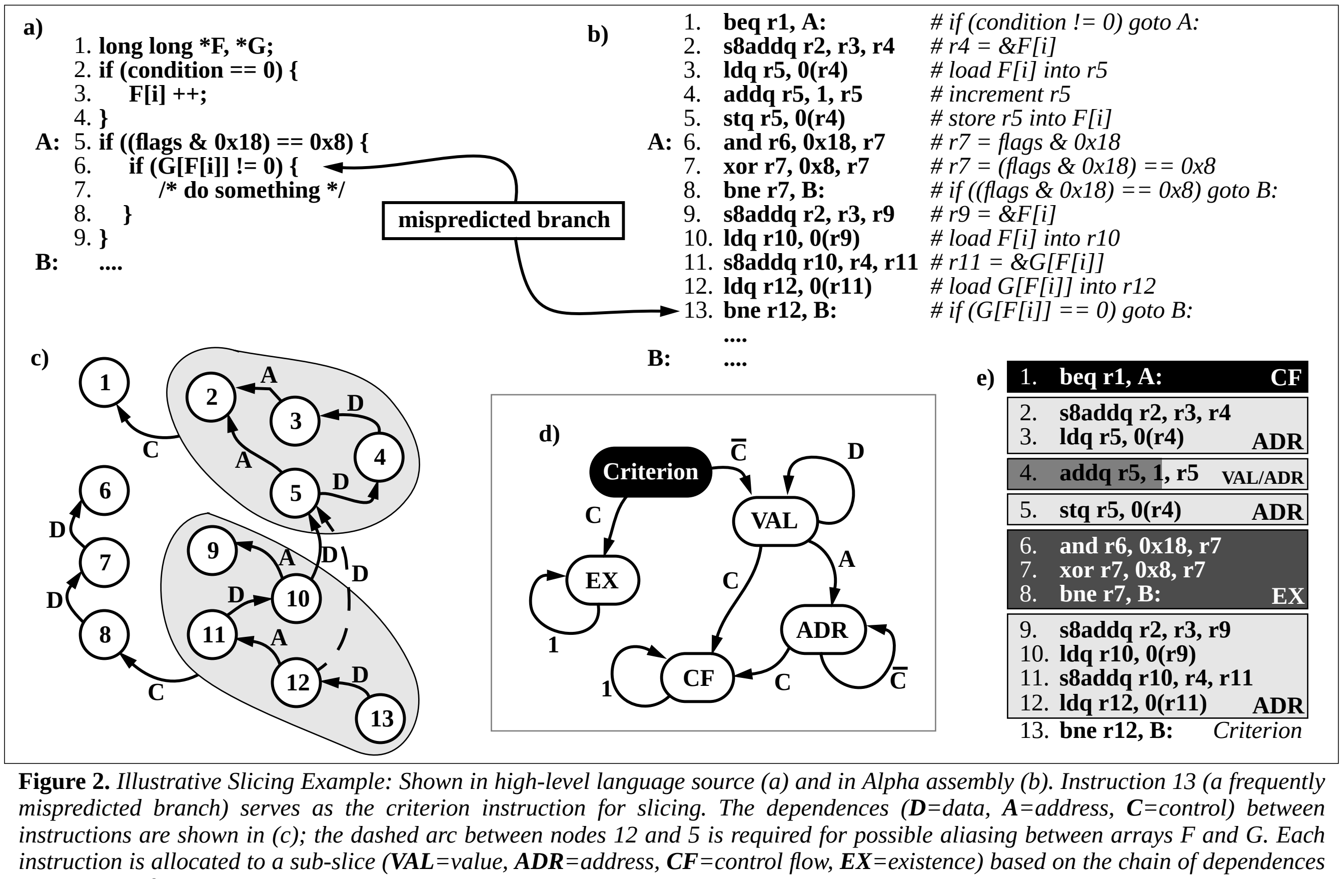
In this paper, we are concerned with identifying and optimizing these backward slices. Typically, the identification is performed by software, but hardware techniques for slice identification have been proposed for specific problem domains [19, 20]. Many embodiments of pre-execution are possible; most notably, the use of "subordinate" threads [5, 23, 28] seems to be a natural fit. However, rather than focus on a particular implementation, we instead study the characteristics of the slices and techniques for reducing slice size which can benefit many implementations.

3 Four Components of a Slice

We find it useful to logically break up a slice into multiple components, or sub-slices. This categorization is based on the role each instruction plays in the slice and suggests techniques that can be used to reduce the size of slices. Many such categorizations are possible. Our categorization recognizes four sub-slices:

- Value (VAL)
- Address (ADR)
- Existence (EX)
- Control flow (CF)

The *value sub-slice* consists of the arithmetic and logic operations which directly manipulate values that are ultimately used to compute the input operands of the criterion instruction. Given the branch outcomes (and hence the dynamic instruction stream) and the resolution of memory dependences, these are the instructions in the data dependence chain leading up to the criterion.



a)

```
    1. long long *F, *G;
    2. if (condition == 0) {
    3.     F[i] ++;
    4. }
A:  5. if ((flags & 0x18) == 0x8) {
    6.     if (G[F[i]] != 0) {
    7.         /* do something */
    8.     }
    9. }
B:     ....
```

b)

```
    1.  beq r1, A:             # if (condition != 0) goto A:
    2.  s8addq r2, r3, r4      # r4 = &F[i]
    3.  ldq r5, 0(r4)          # load F[i] into r5
    4.  addq r5, 1, r5         # increment r5
    5.  stq r5, 0(r4)          # store r5 into F[i]
A:  6.  and r6, 0x18, r7       # r7 = flags & 0x18
    7.  xor r7, 0x8, r7        # r7 = (flags & 0x18) == 0x8
    8.  bne r7, B:             # if ((flags & 0x18) == 0x8) goto B:
    9.  s8addq r2, r3, r9      # r9 = &F[i]
    10. ldq r10, 0(r9)         # load F[i] into r10
    11. s8addq r10, r4, r11    # r11 = &G[F[i]]
    12. ldq r12, 0(r11)        # load G[F[i]] into r12
    13. bne r12, B:            # if (G[F[i]] == 0) goto B:
        ....
B:      ....
```

e)

```
1.  beq r1, A:                CF

2.  s8addq r2, r3, r4
3.  ldq r5, 0(r4)             ADR

4.  addq r5, 1, r5            VAL/ADR

5.  stq r5, 0(r4)             ADR

6.  and r6, 0x18, r7
7.  xor r7, 0x8, r7
8.  bne r7, B:                EX

9.  s8addq r2, r3, r9
10. ldq r10, 0(r9)
11. s8addq r10, r4, r11
12. ldq r12, 0(r11)           ADR
13. bne r12, B:          Criterion
```

Figure 2. *Illustrative Slicing Example: Shown in high-level language source (a) and in Alpha assembly (b). Instruction 13 (a frequently mispredicted branch) serves as the criterion instruction for slicing. The dependences (**D**=data, **A**=address, **C**=control) between instructions are shown in (c); the dashed arc between nodes 12 and 5 is required for possible aliasing between arrays F and G. Each instruction is allocated to a sub-slice (**VAL**=value, **ADR**=address, **CF**=control flow, **EX**=existence) based on the chain of dependences which leads from the criterion instruction to it, using the state machine shown in (d). The assembly instructions are allocated to sub-slices (e); note that, because two different paths lead to the non-memory instruction 4, it is in both the VAL and ADR sub-slices.*

The *address sub-slice* is the set of instructions involved in calculating memory addresses for the value sub-slice. We include the loads and stores themselves with their address calculations because of their immediate offset. By computing these addresses, we identify memory dependences (load/store pairs) which communicate values.

Branch instructions in slices can play two roles: existence and control flow. An *existence* branch determines whether the criterion instruction is going to be executed (i.e., there exist paths starting from that branch which include the criterion and those which do not include it). More precisely, existence branches are not post-dominated [17] by the criterion.

A *control flow* branch has multiple paths which lead to the criterion instruction, but those paths contribute differently to the data dependence of the slice; hence the branch must be resolved to correctly generate the inputs to the criterion instruction.

This categorization of a branch is specific to a criterion; the same static branch may play different roles in different slices, or play no role whatsoever. If a branch does play one of these roles, it and the instructions in its backward slice are put in the existence or control flow sub-slices, accordingly. An indirect branch that has three or more targets could conceivably play both existence and control flow roles, simultaneously.

We allocate instructions to sub-slices based on the chain of dependences that connects them to the criterion instruction. Generally, two classes of dependences are recognized: data and control. *Data dependences* exist between an instruction that creates a value and an instruction that uses the value. A *control dependence* exists between an instruction and a branch if the outcome of the branch determines whether or not the instruction gets executed. Our classification further sub-divides the class of data dependences. If a data dependence to a memory instruction contributes to address generation, we classify it as an *address dependence*.

These dependences are demonstrated in Figure 2. Our example program is shown both in high-level language (Figure 2a) and Alpha assembly (Figure 2b) formats. In this example, the *if* statement in line 6 (assembly statement 13) is a hard to predict branch which we would like to pre-execute, and hence it becomes our criterion instruction. Figure 2c graphically shows the dependences between the assembly instructions. Note that the dashed arc between instructions 12 and 5 must be conservatively included if we cannot prove that arrays A and B do not overlap. Instructions 2 through 5 and 9 through 13 are control dependent on the outcomes of branch instructions 1 and 8, respectively. Dependences whose sources were executed before instruction 1 (i.e., sources of *r1*, *r2*, *r3*, *r6* and memory) are not shown.

A simple finite-state machine (FSM) (shown in Figure 2d) is used for sub-slice allocation. Starting at the criterion instruction (in the "criterion" state), the type of dependence edge traversed dictates a transition in the state machine. Each instruction is allocated to the sub-slice based on the state of the FSM. The one exception is that loads and stores are allocated to the same sub-slice as their address dependences due to the immediate offset. An instruction can be part of multiple sub-slices if multiple dependence paths exist between it and the criterion instruction.

The sub-slice allocation for our example is shown in Figure 2e. Instruction 4 is in both the VAL and ADR sub-slices. This occurs because there is a path consisting of only data dependences from instruction 4 to the criterion, as well as one including an address dependence.

4 Methodology

We study dynamic slices from the SPEC95 integer benchmarks, compiled for the Alpha AXP using Compaq's optimizing C compiler and peak flags (typically -arch ev6 -O4) with static linkage (which is necessary for our simulator infrastructure).

We focus on instructions that cause branch mispredictions and cache misses as the *criteria* for the slices generated. Using a large (64K bits of storage) YAGS predictor [9], a large (32K bits of storage) cascaded indirect branch predictor [7], and a 4-way set-associative 1MB L2 data cache, we identified the static instructions which caused the most branch mispredictions and cache misses in full runs of the benchmarks. For benchmarks which had negligible numbers of L2 cache misses, instructions which caused the most misses in a 2-way set-associative 64KB L1 data cache were selected.

To keep the study manageable, only the worst behaving branches and memory instructions (less than 10 each) were selected for each benchmark. These instructions account for between 7 and 60 percent of the events in the benchmarks. It should be noted that this selection, in many cases, biases our slices toward instructions which are in inner loops, but these are the instructions which represent the most opportunity. For each criterion instruction, we select a region of 100M instructions in length in which that instruction is active (some benchmarks required multiple regions to be selected to cover different phases of execution).

At the core of our infrastructure is a functional simulator built from the Alpha AXP version of the SimpleScalar Toolkit [4]. This simulator generates traces of the user level portions of the benchmarks. Our simulator makes two passes over the instructions: the first pass collects statistics about the dependences and constructs and analyzes control-flow and control-dependency graphs. The second pass gathers statistics about the slices.

We limit the scope of our dynamic slices to a window of 512 dynamic instructions leading up to the criterion. We reason that a pre-fetch distance of 512 instructions should enable on the order of a hundred cycles of latency to be hidden while retiring multiple instructions per cycle.

In the slices presented, artificial dependences on the stack pointer and global pointer are ignored. For fairness, all stack and global pointer computations are ignored when counting distances in the dynamic instruction stream. Likewise, all nops (inserted by the compiler for branch alignment and scheduling purposes) are completely ignored.

Due to space limitations, we cannot include all of our data. We have selected specific examples, which we include as figures, to demonstrate the important phenomena. We plot slices showing the cumulative number of instructions in the slice (the y axis) vs. distance from the criterion in the dynamic instruction stream (the x axis). For example, the point (512, 50) on one of these plots indicates that only 50 of the 512 instructions preceding the criterion (or about 10%) contribute to its execution. These plots include a "100% line" to allow comparisons to the full dynamic instruction stream up to that point. In addition to explaining these examples in the text, we summarize data not included in the figures to describe general trends we have observed.

5 Results

Our analysis begins in Section 5.1 with the VAL sub-slice because the operations in the data dependence chain dictate which instructions are found in the ADR and CF sub-slices. We focus our speculative techniques on the ADR, EX, and CF sub-slices in Sections 5.2, 5.3, and 5.4, respectively. These sub-slices account for the vast majority of the instructions in a conservatively constructed slice.

5.1 Value Sub-slices

In conservatively generated slices, the value sub-slice typically contributes the smallest component to a slice, and many of these instructions are clustered close to the event. Figure 3a shows a VAL sub-slice from a single static event that is representative of many of the slices observed in these benchmarks. Three curves mark the maximum, average, and minimum size of the sub-slice over all dynamic instances (of a particular event-causing static instruction) in the observed interval.

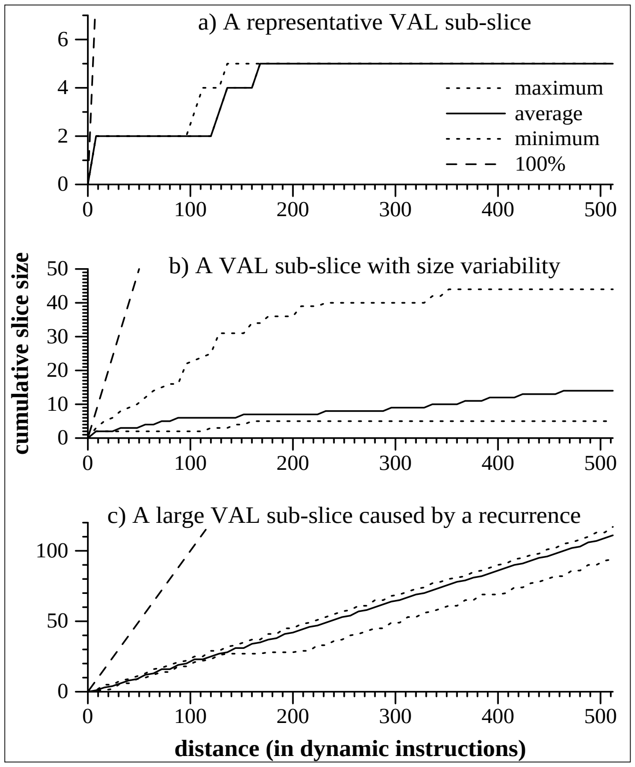


Figure 3. *Dynamic sizes of value (VAL) sub-slices.*

For the vast majority of slices observed, the VAL sub-slice is a very small fraction (less than 2% at a distance of 512 of the whole dynamic instruction stream). Typically, much, if not all, of the sub-slice is concentrated near the criterion instruction (within the first 10-20 instructions). In general, there is very little variability in the size of the VAL sub-slice.

Figure 3b shows a different static event where the maximum slice size is much larger than either the average or minimum sizes. When there is a lot of size variability in a VAL sub-slice, generally it is due to different control flow paths. In the above instance, the different behaviors are associated with different calling contexts.

In the rare cases when the VAL sub-slice is large, it is due to a recurrence. Figure 3c shows a case where one fifth of our window of 512 instructions is in the VAL sub-slice. Recurrences are discussed in Section 6.1.

The small size of VAL slices is somewhat an artifact of our definition of the VAL sub-slice, but the general trend is supported by previous parallelism studies [2, 12, 26]. Because of the small sizes, we are not directly concerned with further reducing the size of the VAL sub-slice. As will be seen in the next sections, much of the computation in these integer applications is present to identify the values on which to operate and which operations to perform.

5.2 Address Sub-slices

A load in a dynamic slice reads a value which was either created before the slice began or stored by a store in the slice. In the first case, the address must be generated to retrieve the value from memory; in the second, the address is generated to identify the store which supplies the value. In many cases, it is difficult to prove anything about the communication patterns between stores and loads, leading to ambiguous memory dependences. Before dealing with this ambiguity, we characterize the ADR sub-slice assuming an oracle that only includes the required loads and stores. We call this the unambiguous ADR sub-slice.

The average size of these sub-slices is quite a bit larger than VAL sub-slices. The slice shown in Figure 4a is representative, although there is a lot of variation between benchmarks as well as within a benchmark. On average, unambiguous ADR sub-slices consist of 4-10% of the 512 instructions before an event.

Two other characteristics are demonstrated by Figure 4a: a lot of variability between maximum and minimum slice sizes, and the appearance of a gradual ramp up (indicating an even distribution of instructions from the slice). Figure 4b shows a sampling of dynamic instances which make up the aggregate slice shown in Figure 4a. It can be seen that the spectrum between minimum and maximum is continuously populated rather than being concentrated at the extremes. Also, the individual slices are bursty (the gradual ramp shown in Figure 4a is merely an aggregate behavior), consisting of regions that affect the slice separated by flat regions which do not contribute. This is a reflection of the fact that a program is an interleaving of partially independent computations.

As previously mentioned, the unambiguous ADR sub-slices are optimistic. In general, we cannot identify which stores contribute to the VAL sub-slice without computing all addresses. In fact, without any information, there are two possible policies: conservative (assume dependence) and naive speculation (assume independence). The conservative policy assumes that any store could be part of the slice, forcing all store address computations into a conservatively large ADR sub-slice. Naive speculation predicts that no stores will affect the slice, and suffers from data dependence mis-speculation when stores should be included in the slice.

Figure 4c demonstrates the consequences of the conservative policy for the same slice considered in Figure 4a. On average, the ADR sub-slice has increased to 30% of the full program (around 60% in the worst case), compared to 5% for the unambiguous ADR sub-slice.

Figure 4d shows the mis-speculation rate for the naive speculation (speculate always) policy, as a function of distance in dynamic instructions from the criterion instruction (using the same slice as above). This rate shows the likelihood that at least one store that affects the VAL slice would be ignored. This data is somewhat pessimistic in that even if the store would not change the value in memory (i.e., a silent store) it is marked as a mis-speculation. Increasing the pre-execution distance rapidly increases the likelihood that a memory communication will be required to compute the criterion. Large windows are likely to include entire functions and therefore it is not uncommon for values in the VAL slice to be saved to and restored from the stack. There are some slices which contain no stores within 512 instructions, and there are some slices which are always misspeculated past a distance of 50.

5.2.1 Profiling Store Sets

Clearly, neither of these naive policies is sufficient. However with some information about past behavior, the unambiguous ADR sub-slice can be approximated by a speculative ADR sub-slice. Although memory allows any store to potentially communicate with any load, in practice the active dependences are only a small subset of all possible communication arcs. During any program execution, a majority of static loads are fed by a single static store [6, 13, 14]; the rest are fed by a small set of stores. This behavior seems to be inherent to the program's structure, because the same dependences are exercised across different data inputs. This suggests that profiling can be used to identify memory dependences with high accuracy, as proposed by Reinman, et al. [18].

Using these profiles, we can reduce the size of the ambiguous ADR sub-slice. Stores in the dynamic instruction stream that are not in the store sets of any of the loads already in the ADR sub-slice can be ignored. Only when the profile is inaccurate does a mis-speculation occur.

For most benchmarks, memory dependence profiles reduce the size of the ADR sub-slice to close to that of the unambiguous

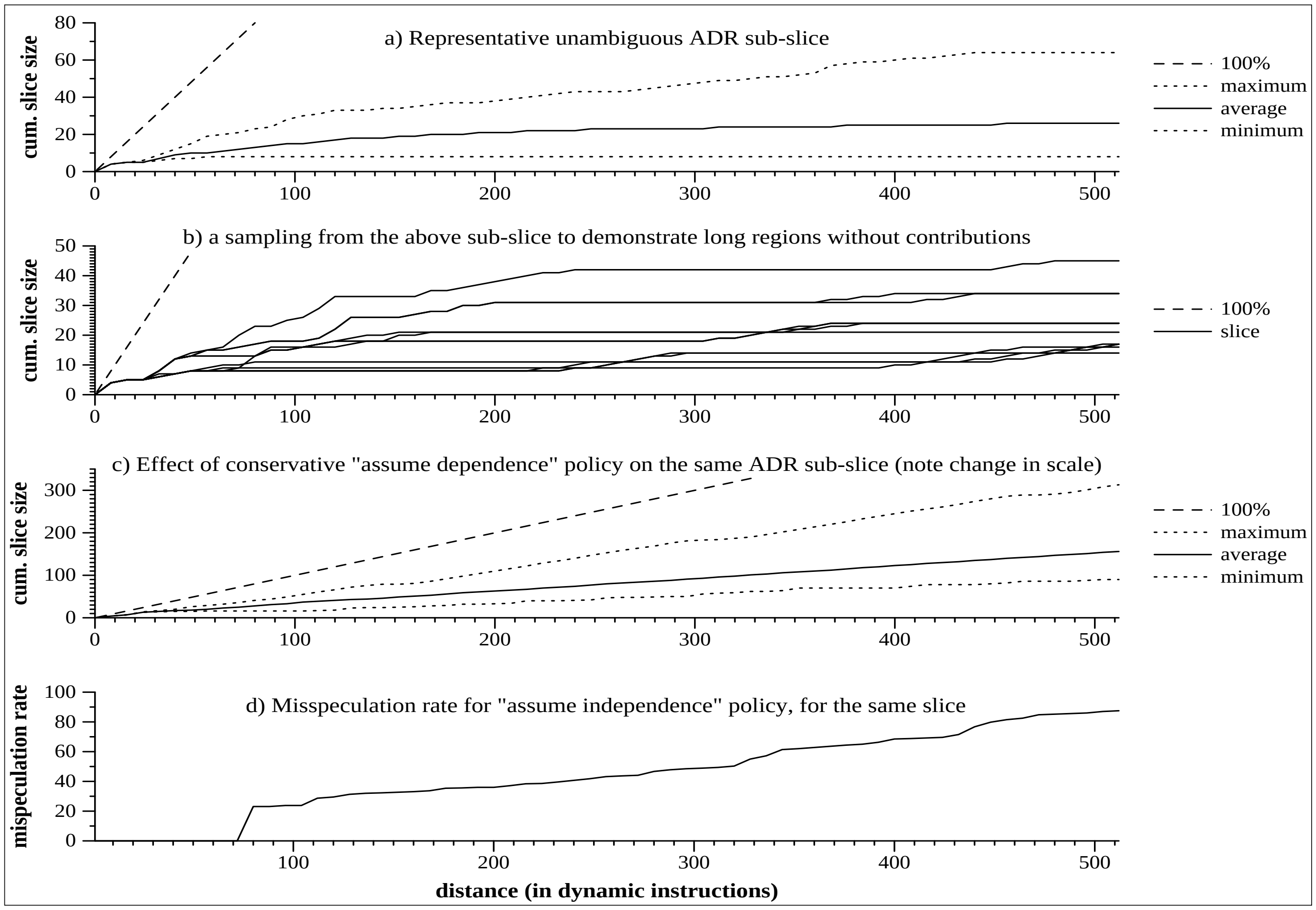


Figure 4. *Dynamic sizes of address (ADR) sub-slices.*

ADR sub-slice. On average, the ADR sub-slices are less than 10 instructions larger than the unambiguous version at a distance of 512. The fact that this disparity is so small indicates that often only one store from the store set exists in the window of 512 instructions.

Typically, when the disparity is large, there are many dynamic instances of the same static store in the window (because of a loop), and the store set mechanism must generate all of their addresses to select the correct producer. For this reason, using speculation to remove infrequently used stores from the store set provides only modest benefit. Usually, such stores do not contribute significantly to the store set because they are rarely executed. However, the induced mis-speculation rate is also modest (except in cases of recurrences discussed in Section 6.1), implying that, in most cases, it is sufficient to identify the dominant dependences; hence a sampling technique can be used.

Overall, store set profiling is successful at reducing the ADR sub-slice to the size of the unambiguous ADR sub-slice, but the size of the ADR sub-slice still dominates that of the VAL sub-slice. In the next section we exploit another common behavior of memory dependences to further reduce ADR sub-slice size.

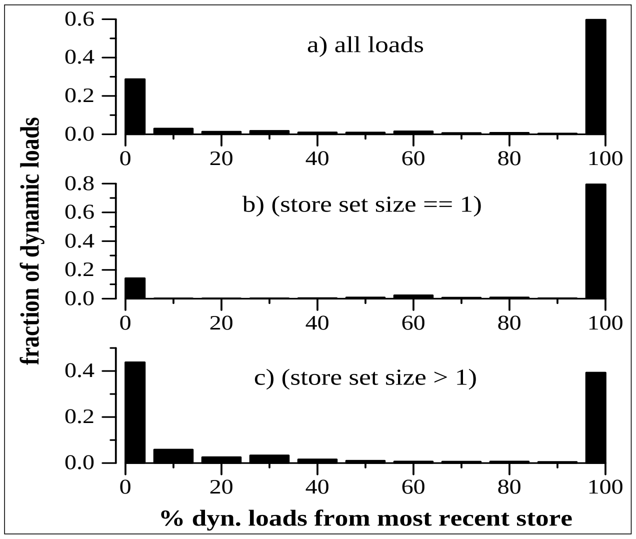


Figure 5. *Bimodal distribution of a static load's likelihood to use the most recently stored value from its store set.*

5.2.2 Speculative Register Allocation to Remove Unnecessary Address Calculations

Previous research has shown that many memory dependences are satisfied by the *most recent store* from its store set [13, 15, 18, 25]. In fact, static loads can be categorized into two groups: those that are very likely to be satisfied by the most recent store from their store set, and those that are very unlikely. Figure 5 shows the distribution of these likelihoods across all benchmarks, weighted by the execution frequency of the associated static load. The distribution is distinctly bimodal, in that highly-biased (at least a 95% bias) static instructions represent almost 90% of dynamic loads executed.

This behavior is not limited to loads whose store sets consist of a single static store (Figure 5b), but exists also for loads with multiple stores in their store sets (Figure 5c). Given these extreme tendencies, it should be easy to categorize loads into these two groups, even with incomplete data.

Once control flow has been resolved and the stores that could be in the ADR sub-slice have been identified, loads that exhibit this "most recent store" behavior can be accurately paired with stores without the need for address generation [13, 15, 18, 25]. If a register can be (speculatively) allocated for the communication in the slice, then both the load and store can be removed with all of the instructions in their address calculations. In most of the benchmarks, this can significantly reduce the size of the ADR sub-slice.

Some of the slices we consider consist exclusively of loads with "most recent store" behavior, causing the ADR sub-slice to disappear entirely. A majority of the remaining slices are significantly reduced, often cut in half. Figure 6 shows the average sizes of a representative ADR sub-slice when constructed with the techniques discussed.

When a memory dependence does not have a "most recent store" behavior, we have found that it is often inefficient to include its store set in the slice. Two such cases are prevalent. If only one dynamic instance of a store from the store set appears in the window, it is unlikely to cause a mis-speculation if we ignore it (by definition). The other common case is when multiple stores are in the window because they are in a tight loop (as shown in the illustrative example in Figure 7). In this case, address generation and loop control (which would need to be in the full slice as well) are a

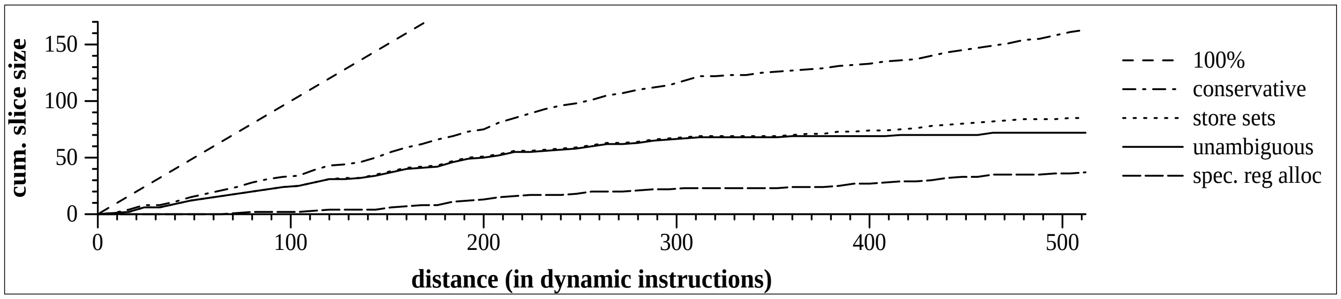


Figure 6. *Address (ADR) sub-slice average size by identification technique.* ***Conservative*** *includes all store address calculations,* ***store sets*** *includes only those from store sets of loads in the VAL sub-slice, and* ***unambiguous*** *includes only those stores which affect the criterion.* ***Speculative register allocation*** *removes the load and store address calculation of memory dependences which exhibit the "most recent store" behavior.*

significant portion of the loop. Inclusion of these instructions in the slice can significantly impact its size; a better solution in this second case may be to initiate the pre-execution immediately after the loop completes. This more efficient slice comes at the cost of decreased pre-execution distance.

5.3 Existence Slices

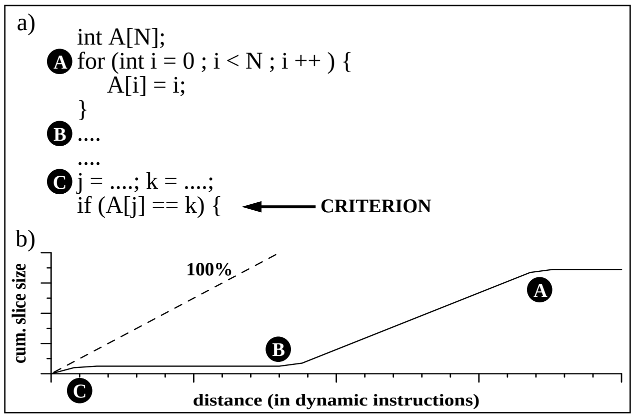
Without control independence analysis, all branch targets need to be resolved to determine whether the criterion will be executed. In many cases this requires executing 80% of the 512 instructions before a criterion. With control independence analysis, the EX sub-slice is much smaller but still can be substantial, even if we only consider control flow arcs which are exercised at least once. On average, the EX sub-slices are in the range of 10-12% (about 50-60 instructions at a pre-fetch distance of 512), but can be as high as 20-30% (100-150 instructions at a distance of 512) when the criterion is in a loop.

Like VAL and ADR sub-slices, the EX sub-slice's instructions tend to be clustered toward the criterion. This is not surprising given that the farther the criterion is from a branch, the more likely it is that there is a reconvergent point between the two.

Often more than half of the instructions in the EX sub-slice are memory dependences (in the same vein as the ADR sub-slice). By applying our memory dependence techniques, we can remove a significant portion of these. Frequently, the remaining EX memory dependences are already present in the ADR sub-slice. For this reason, the memory dependences from the EX sub-slice seldom contribute significantly to the total size of the slice.

The same cannot be said for the non-memory dependences in the EX sub-slice. In general, there is little overlap between these instructions and any other slice. They tend to directly contribute to the size of the total slice.

The impact of the EX sub-slice on the slice as a whole can be reduced by ignoring highly biased existence branches. We considered removing branches with biases greater than 98 percent and found that the benefit varied between benchmarks. By exploiting this statistical or speculative control independence, around half the slices we observed were reduced to one half to one third of their sizes. The other half were for the most part untouched.



```
a)
      int A[N];
   A  for (int i = 0 ; i < N ; i ++ ) {
          A[i] = i;
      }
   B  ....
      ....
   C  j = ....; k = ....;
      if (A[j] == k) {  ◄──── CRITERION
```

Figure 7. *Illustrative example of an inefficient slice due to a memory dependence (source code (a) and backward slice (b)). Assuming j is evenly distributed between 0 and (N-1), each store in the loop has a 1/N chance of contributing to the criterion. Including the loop in the slice impacts the slice's size, but removing it entirely causes mis-speculation if a pre-execution is initiated before (A). Initiating it at (B) avoids this trade-off, but reduces the latency that can be tolerated.*

One common existence branch is a null pointer test before a dereference. This branch can be removed by recognizing exception conditions as the end of a pre-execution. However, because the associated pointer value usually occurs in one of the other sub-slices, this optimization does not significantly reduce total slice size.

5.4 Control Flow Slices

In the conservative case, the CF sub-slice is the largest contributor to many of the backward slices and is also the one we have had the least success optimizing due to the limitations of our current infrastructure. There is significant variability; a number of criteria have non-existent CF sub-slices, but in some of the control-intensive benchmarks the conservative CF sub-slice can be as large as 50 to 60 percent of the 512 instruction window we considered.

Similar to the EX case above, the CF sub-slices include memory dependences, and their contributions can likewise be reduced using the techniques presented in Section 5.2. Unlike the EX sub-slice, however, speculating that highly biased branches (greater than 98% bias) will always follow their bias does not lead to a substantial reduction of the sub-slice size, in general. In many cases, these branches are less biased, so a larger mis-speculation rate must be tolerated for slice reduction. Also, unlike VAL, ADR, and EX sub-slices which tend to be more concentrated near the criterion, CF sub-slices are more evenly distributed throughout the dynamic instruction stream.

Upon closer inspection, we determined that some of the control dependences that we identified were false dependences. In these cases, instructions that are control dependent on the branch appear to be part of the slice, but all paths from the branch are symmetric with respect to the criterion. A common example of this is a conditional function call (shown in Figure 8) which saves and restores a register value in the slice. The function call performs a net null operation on the register, but our current infrastructure cannot detect this. We have identified other less trivial instances of this phenomena by inspecting slices by hand, but we have not been able to quantify the effect of these false dependences.

6 Discussion

In this section, we briefly discuss some characteristics of the slices and the process of constructing slices.

```
A = ....;                                   function() {
if (B) {                                        save A;
    function();                                 .....
}                                               restore A;
if (A) {  ◄──── CRITERION                   }
```

Figure 8. *Example of a false control dependence. Both paths through the example are equivalent with respect to **A**, because the path through **function()** has a net null effect on **A**.*

6.1 Overlapping Slices and Recurrences

It is not uncommon for the backward slices of more than one criteria to share instructions. In these cases, it may be beneficial to merge the slices to reduce the pre-execution overhead. A special case of this is when one criterion is in the slice of a second, in which case the second criterion's slice is often a superset of the first criterion's slice. An unpredictable branch based on the result of a cache-missing load is a repeating theme. Since the latency of these two events is serialized, the initiation of the pre-execution must be scheduled appropriately.

A special case of overlapping slices is when multiple slices from the same static criterion overlap. This occurs when a criterion instruction is in a tight loop. Each iteration will have one or more recurrences that appear in its backward slice. These recurrences can be address, data, or control. They can make the backward slice look deceptively large, but in such a slice there are many instances of the criterion evaluated, so the incremental slice size is small (less than or equal to the size of a loop iteration).

These incremental slices can only be exploited if we enable a pre-execution to evaluate the criterion multiple times. At this point, the decision must be made as to how many iterations should be executed. Typically, highly biased branches are speculatively removed from the existence sub-slice, but if the loop back-edge is removed, the pre-execution could iterate forever. To reduce the incremental slice size, complicated existence slices can be replaced with simple control which executes a fixed number of iterations (either using a loop or static unrolling), or feedback from the main computation can be used to throttle or terminate a pre-execution computation.

When the incremental backward slice makes up a significant portion of a loop iteration, the benefit of latency tolerance must be derived from the distance between the initiation of the pre-execution and the first iteration of the loop. If the long latency event itself is part of the recurrence (as in pointer chasing), then the initiation must be scheduled to tolerate the serial latencies. Such techniques, including root jumping, are discussed in [21].

Speculation has to be used very carefully on dependences in the recurrence. If an incorrect value is computed on the recurrence path, then it will be propagated to all future iterations.

6.2 Traditional Optimizations

Once a slice has been reduced to its essential elements, traditional compiler techniques can be used to further optimize it. In addition to the speculative register allocation discussed in Section 5.2.2, we have seen opportunities for loop invariant code motion, the removal of register moves, strength reduction, and the conversion of indirect branches into direct branches. In many instances, these techniques could not be applied to the original program due to ambiguous memory dependences, infrequently executed branches, or register pressure.

Similarly, dynamic compilation techniques could be used to generate slices which exploit invariant run-time values. Since these pre-execution computations are speculative, it is not necessary to verify that these run-time values are truly constant.

6.3 Identifying Slices

The effort required to identify slices depends significantly on many aspects of a program, not least of which is its representation. In this paper, we took a low-level approach, analyzing the program at the instruction level, as a processor might analyze it. Without the high-level information available from the source level, our infrastructure needed to rediscover some of the information that was known to the compiler.

This process is occasionally aggravated by the compiler; the most noteworthy example is code replication. Techniques like loop unrolling and trace scheduling require the slice construction algorithm to reconcile the replicated blocks with each other. Also, it is not uncommon for the criterion instructions to be replicated, potentially requiring multiple slices to pre-execute what is logically a single operation. Only when the different instances of a block have radically different behavior (with respect to the criterion) does the slice benefit from such replication. Programmers can likewise be a source of replication if they unroll loops by hand or otherwise replicate code.

The slice construction routine operates most efficiently on the smallest representation of the program. This requirement often conflicts with many performance optimizations performed by the compiler.

7 Conclusion

Instructions whose behavior cannot be anticipated by branch predictors or caches can significantly degrade processor performance. In the future, this will be further aggravated as processor microarchitecture continues the trend to higher clock speeds and deeper pipelines. This study finds that, in many cases, the behavior of these instructions can be represented by a reduced form of the program, specialized to compute the outcome of these instructions. If executed in parallel with the whole program, these reduced programs can initiate long latency events early so that they have completed by the time they are encountered by the whole program's execution.

These reduced programs are constructed by identifying the backward slice of the instruction to be pre-executed. In many integer benchmarks, the conservative backward slice consists of a large fraction of the program. The key to reducing slice size is speculation; by treating the result of a pre-execution as only a hint, infrequent and ambiguous dependences can be ignored. This speculation must be guided by profiling the application to identify dominant paths and dependencies and by analysis to coalesce paths which are equivalent with respect to the slice.

By exploiting control-independence, highly biased branches, and the stable nature of load-store dependences, we were able to reduce many slices down to less than 10% of the full program's dynamic instruction stream for the window of 512 instructions we considered. In almost all cases, mis-speculation rates below 5% were maintained. There were some slices for which the techniques we investigated were insufficient, often due to complex memory dependences or slices which necessitated much of the control flow to be resolved.

The techniques considered here are by no means an exhaustive list. Future work includes investigating if additional program behaviors and speculation techniques can be used to further reduce these slices. Path profiles might allow further refinement of the control flow sub-slice by enabling different instances of the same static branch to be treated differently. Also, value prediction has the potential to break data dependences, possibly removing full computation chains from the slice.

8 Acknowledgements

Many of the ideas in this paper, as well as the writing, have benefitted from discussions with Amir Roth. In addition, we would like to thank Manoj Plakal, Dan Sorin, and the anonymous reviewers for their comments on earlier drafts of this paper. This work was supported in part by National Science Foundation grants MIP-9505853 and CCR-9900584 and an equipment grant from Intel Corp. Craig Zilles was supported by NSF and Intel fellowships during the academic years 1998 and 1999, respectively.

9 References

[1] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proc. 26th International Symposium on Microarchitecture*, pages 139–152, Dec. 1993.

[2] T. Austin and G. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proc. 19th International Symposium on Computer Architecture*, May 1992.

[3] D. Binkley and K. Gallagher. *Advances in Computers*, chapter 34: Program Slicing. Academic Press, San Diego, CA, 1996.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th International Symposium on Computer Architecture*, May 1999.

[6] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[7] K. Driesen and U. Hoelzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Proc. 31st International Symposium on Microarchitecture*, pages 249–258, Dec. 1998.

[8] K. Ebcioglu, E. Altman, S. Sathaye, and M. Gschwind. Optimizations and Oracle Parallelism with Dynamic Translation. In *Proc. 32nd International Symposium on Microarchitecture*, pages 284–295, Nov. 1999.

[9] A. Eden and T. Mudge. The YAGS Branch Prediction Scheme. In *Proc. 31nd International Symposium on Microarchitecture*, pages 69–77, Nov. 1998.

[10] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st International Symposium on Microarchitecture*, pages 59–68, Dec. 1998.

[11] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *Proc. 29th International Symposium on Microarchitecture*, Dec. 1996.

[12] M. Lam and R. Wilson. Limits of control flow on parallelism. In *Proc. 19th International Symposium on Computer Architecture*, May 1992.

[13] A. Moshovos. *Memory Dependence Prediction*. Ph.D. thesis, University of Wisconsin-Madison, 1998.

[14] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proc. 24th International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.

[15] A. Moshovos and G. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[16] T. Mowry and C.-K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997.

[17] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, CA, 1997.

[18] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying Loads and Store Instructions for Memory Renaming. In *Proc. 1999 International Conference on Supercomputing*, Jun. 1999.

[19] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[20] A. Roth, A. Moshovos, and G. Sohi. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proc. 1999 International Conference on Supercomputing*, pages 356–364, Jun. 1999.

[21] A. Roth and G. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proc. 26th International Symposium on Computer Architecture*, pages 111–121, May 1999.

[22] A. Roth and G. Sohi. Speculative Data Driven Sequencing for Imperative Programs. Technical Report CS-TR-2000-1411, University of Wisconsin, Madison, Feb. 2000.

[23] Y. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, V(3):121–181, 1995.

[25] G. Tyson and T. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[26] D. Wall. Limits of Instruction Level Parallelism. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[27] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[28] C. Zilles, J. Emer, and G. Sohi. The Use of Multithreading for Exception Handling. In *Proc. 32nd International Symposium on Microarchitecture*, pages 219–229, Nov. 1999.