Slice-Processors: An Implementation of Operation-Based Prediction

Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

Dionisios N. Pnevmatikatos
Electronic and Computer Engineering
Technical University of Crete
pnevmati@ece.tuc.gr

Amirali Baniasadi
Electrical and Computer Engineering
Northwestern University
amirali@ece.northwestern.edu

ABSTRACT

We describe the Slice Processor micro-architecture that implements a generalized operation-based prefetching mechanism. Operation-based prefetchers predict the series of operations, or the computation slice that can be used to calculate forthcoming memory references. This is in contrast to outcome-based predictors that exploit regularities in the (address) outcome stream. Slice processors are a generalization of existing operation-based prefetching mechanisms such as stream buffers where the operation itself is fixed in the design (e.g., address + stride). A slice processor dynamically identifies frequently missing loads and extracts on-the-fly the relevant address computation slices. Such slices are then executed in-parallel with the main sequential thread prefetching memory data. We describe the various support structures and emphasize the design of the slice detection mechanism. We demonstrate that a relatively simple organization can significantly improve performance over an aggressive, dynamically-scheduled processor and for a set of pointer-intensive programs and for some integer applications from the SPEC'95 suite. In particular, a slice processor that can detect slices of up to 8 instructions extracted over of a region of up to 32 instructions improves performance by 11% on the average (even if slice detection requires up to 32 cycles). Allowing slices of up to 16 instructions results in an average performance improvement of 15%. Finally, we study how our operation-based predictor interacts with an outcome-based one and find them mutually beneficial.

1. INTRODUCTION

Prediction-based methods have been instrumental in sustaining the exponential performance growth we have enjoyed over the past decades. Prediction allows processors to guess forthcoming program demands and take appropriate action on the program's behalf. A particular area where prediction-based methods have been very successful is that of prefetching memory data. Prefetching guesses what memory data the program will need and attempts to read them in advance of the actual program references. If successful, prefetching reduces the negative impact of long memory latencies. While existing prefetching methods have been effective, there is a continuous need for improvements main because memory latencies do not improve as fast as processor clock speeds.

Both static and dynamic prefetching methods for the data and the instruction memory streams have been proposed. In this work we are concerned with dynamic, hardware-based memory data prefetching. Existing prefetchers of this kind fall into two broad categories: *outcome-based* and *operation-based*. *Outcome-based* methods work by detecting repeating patterns in the program's data address stream (a subset of the program's data outcome stream). For example, an outcome-based prefetcher may observe that every time the program accesses address 100 soon thereafter it also accesses address 200. Consequently, such a prefetcher may then initiate an access to address 200 every time an access to address 100 is observed. *Operation-based* prefetchers also observe the program's address stream. However, rather than trying to identify repeating address patterns these prefetchers look for evidence of repeating operations (or computations). In general, these prefetchers predict that once the program accesses address *A* then it will soon access address *f(A)* where *f()* is some operation or sequence of operations. *Stream buffers* or *stride-based* prefetchers are examples of existing operation-based prefetchers. There, *f()* is simply a linear progression of the form *"f(A)=A + stride"*. In this case, the sought after operation is fixed in the design. Other operation-based prefetching methods have been proposed, however, they also target specific classes of memory accesses such as those produced when accessing recursive data structures (see section 5 for a summary).

In this work we build on the success of existing operation-based prefetching methods and propose the *Slice Processor* micro-architecture. *Slice Processors* are a generalized implementation of operation-based prefetching. A slice processor *dynamically* identifies problematic loads and automatically extracts the computation slices that were used to calculate their target addresses. These slices include only those instructions that are necessary for executing the offending load. The slice is then used to calculate subsequent memory accesses and to prefetch them. Slices execute speculatively in the form of *scout-threads* which run in-parallel with the main sequential program (or thread). Scout threads affect program execution only indirectly by prefetching memory data.

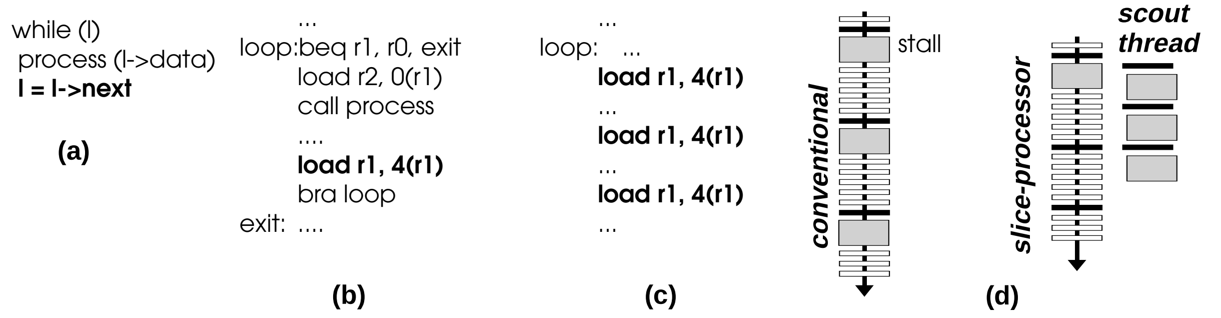


Figure 1: (a) A code-fragment with a frequently missing load shown in boldface. (b) Assembly pseudo-code for the code of part (a). (c) Dynamic execution trace showing the operations used to calculate the references that miss. (d) Comparing conventional execution with execution in a slice processor (ideal case).

In this paper, much of our focus is in discussing the details of the slice processor microarchitecture. We made an effort to minimize complexity while facilitating integration with a conventional high-performance processor core. We also made an effort to minimize as much as possible the interaction with the conventional core. We demonstrate that a slice processor can improve performance by 11% on the average by prefetching memory data for a set of pointer-intensive integer applications. This slice processor is capable of detecting slices of up to 8 instructions in length detected over a region of up to 32 instructions and requires 32 cycles to detect such slices. Moreover, it uses in-order execution for scout threads.

Other proposals aimed at extracting and pre-executing program slices exist. We review them in section 5. To the best of our knowledge, however, this is the first work that proposes mechanisms for automatic, dynamic identification of arbitrary program slices and of pre-executing them for prefetching memory data. While it may be possible to extend our methods to also predict other events such as branches or values, in this work we restrict our attention to prefetching.

The rest of this paper is organized as follows. In section 2 we explain the rationale of our approach by means of an example. We also discuss the advantages and disadvantages of operation-based prediction. In section 3, we present an implementation of the slice-processor. Here we explain how we can leverage the experience with outcome-based predictors to also perform operation-based prediction. In section 4, we demonstrate that our slice-processor architecture can be used to tolerate miss latencies for a set of integer applications. We also show that in most cases, it interacts favorably with a stride prefetcher. In section 5, we review related work. Finally, in section 6 we offer concluding remarks.

2. MOTIVATION

We motivate operation-prediction using the code fragment of figure 1(a). A linked list is traversed while performing a computation on its data elements (shown as a call to routine "process()"). Let us assume that the subroutine "process()" is well-behaved and that the linked list is not cache-resident. Moreover, let us assume that performance is limited by how quickly we can traverse the linked list. Under these assumptions, performance will be limited by how quickly we can service the various instances of the "l = l->next" statement. As shown in part (b), in machine code, this statement translates into a single self-recurring load (shown in bold). Dynamically, the execution of this loop unravels as shown in part (c) with multiple instances of the aforementioned load appearing in sequence.

An outcome-based prefetcher will be able to predict the list references only if some form of regularity exists in the resulting address stream. This may be incidental or the result of carefully allocating the list's elements. In general, however, this may not be the case[1]. While there might be no regularity in the memory stream, we can observe that regularity exists in the *operation* that produces the memory stream. In fact, this operation remains constant throughout the loop. As shown in part (c), the next address is calculated starting from the current address in register r1, by simply accessing memory location "r1+4". This observation suggests that it might be possible to predict the operation (as opposed to its outcomes) and use it to compute the addresses (the outcomes) of the linked list's elements. To do so, we may use techniques similar in spirit to those used in outcome-based predictors, where rather than using *outcomes* as our prediction unit we instead use *operations* or *computation slices*.

Our slice processor builds on this observation. The goal of a slice processor is to dynamically predict the operations, or the *slices* that are used to calculate the memory addresses accessed by frequently missing loads. This is done by observing the slices of offending loads as they are committed. These slices are then used to compute subsequent memory references. In the example of figure 1, the slice processor will extract the slice containing the self-recurring load references shown in part (c). This slice will be used to prefetch subsequent list elements every time the frequently missing load is encountered. An abstract view of how execution progresses as shown in figure 1(d). On the left, we show how execution may proceed in a conventional processor. Upon encountering a load that miss (dark box), execution stalls waiting for memory, shown as the gray box (of course, in a dynamically-scheduled processor, other instructions may execute in the meantime, however, we omit them for clarity). In a slice processor, as soon as a slice is extracted, it is used to prefetch memory data in the form of *scout thread*. The scout

1. Regularity will exist when the same list is traversed repeatedly. In this case, the space required to store the repeating pattern will be proportional to the size of the linked list itself. Of course, careful encoding may be able to reduce these space requirements.

thread runs in parallel with the main sequential thread affecting its performance indirectly.

While we have used a simple linked list as our motivating example, the slice processor is, in principle, capable of extracting arbitrary slices. In the general case, detected slices do not consist of self-recurring instances of the same load. Rather they include an arbitrary mix of instructions that end with a load that misses often. Moreover, while we have shown slices that use the previous address of a load reference to compute the next one, in general this may not be the case. A slice may be rooted on other computations. In section 4.8, we discuss few of the most frequent slices exploited in a slice processor implementation and we show, that the slice processor can capture fairly elaborate address calculations in addition to array and simple linked lists.

2.1. Running Ahead of the Main Thread

One may wonder how scout threads can run ahead of the main thread and prefetch memory data. There are several reasons why this may be the case:

1. A conventional processor has to fetch, decode and execute not only the slice, but also, all other instructions. In our example, these include the intervening function call and the loop overhead. The scout thread comprises only those instructions that are necessary for calculating an address.
2. The processor's scope is also limited by the size of its instruction window.
3. Every time a branch mis-prediction occurs, a conventional processor discards all subsequent instructions and starts over again. However, the computations that lead to a frequently missing load may be control independent of those intermediate mis-speculated branches. (In the example of figure 1, the list traversal loads are control independent of any control-flow in function "process()".) Consequently, the more instructions the conventional processor has to execute and the more spurious branch mispredictions it encounters, the higher the probability that a scout thread will manage to prefetch data.
4. Finally, conventional processors have no mechanisms for recognizing that a particular slice should be given priority and executed as soon as possible even when it is present in the instruction window A scout thread implements a "poor man's" prioritization of slices that lead to frequently missing loads. Intuitively, for applications with low parallelism, this is less of a concern for future wide superscalars.

2.2. Operation- and Outcome-based Prediction

We have explained that a slice processor has the potential of prefetching memory references that may foil outcome-based predictors. However, when outcome-based prediction is possible, it has an inherent advantage. Outcome-based prediction collapses a series of computations into a lookup reducing the inherent latency required to calculate memory addresses. Operation-based prediction cannot reduce this latency beyond what may be possible by optimizing the slice. Moreover, while the outcome stream of a computation may exhibit strong repetition, the computation itself may be complex or may change unpredictably due to intervening control flow. For this reason, we view slice processors as complementary to existing outcome-based methods.

3. THE SLICE-PROCESSOR MICRO-ARCHITECTURE

In designing the slice processor microarchitecture we built on the experience with outcome-based predictors. The simplest form of dynamic outcome-based predictors is that of *"last outcome"*. For example, a "last outcome" branch predictor simply predicts that a branch will follow the same direction it did last time it was encountered. Accordingly, we designed the slice processor to implement a *"last operation"* prediction scheme. That is, once a frequently missing load is found and its slice is extracted, our slice processor predicts that next time around the same slice can be used to calculate the desired target address.

To explain the operation of a slice processor it is interesting to review how outcome-based predictors operate. In an outcome-based predictor, we start by selecting candidate instructions for prediction (e.g., loads for address prediction and prefetching, or calls and returns for return address stack prediction). Then, we observe their outcome stream identifying regularities. In a "last outcome" predictor we simply record the last observed outcome. Next time around the same instruction is encountered, we simply predict that it will produce the recorded outcome. Optionally, we may also implement a confidence mechanism to selectively turn on and off prediction based on past successes or failures. Prediction in the slice-processor architecture parallels that of an outcome-based predictor. In particular, the slice processor operates as follows:

- We start by identifying candidate instructions for operation prediction. Since we limit our attention to prefetching, our candidate selector inspects retiring instructions looking for frequently missing loads. More elaborate candidate selection heuristics may be possible (for example, only selecting loads that an outcome-based predictor fails to predict), however, in this work we are interested in demonstrating the potential of our approach hence a simple, first-cut candidate selector suffices.
- .Outcome-based predictors simply record the outcomes of candidate instructions. In an operation-based predictor however, our unit of prediction is an operation, or a computation slice. Before we can record this slice, we have to detect it. Accordingly, once a candidate load is identified, the dataflow graph is traversed in reverse to identify the computation slice that lead to it. Detecting these slices is the responsibility of the *Slicer.* Detected slices are stored in the *slice cache* where they are identified by the PC of the *lead* instruction. The lead instruction is the earliest in program order instruction of the slice (this is *not* the frequently missing load which appears at the end of the slice)
- In outcome-based predictor, prediction is performed by simply looking up previously stored outcomes (while we may be using elaborate hash functions and leveraging history information, nevertheless, we eventually reach to a previously recorded outcome). In operation-based prediction, we have to execute a computation slice to obtain the desired outcome. This is done by spawning the corresponding slice upon encountering a dynamic instance of a lead instruction. The

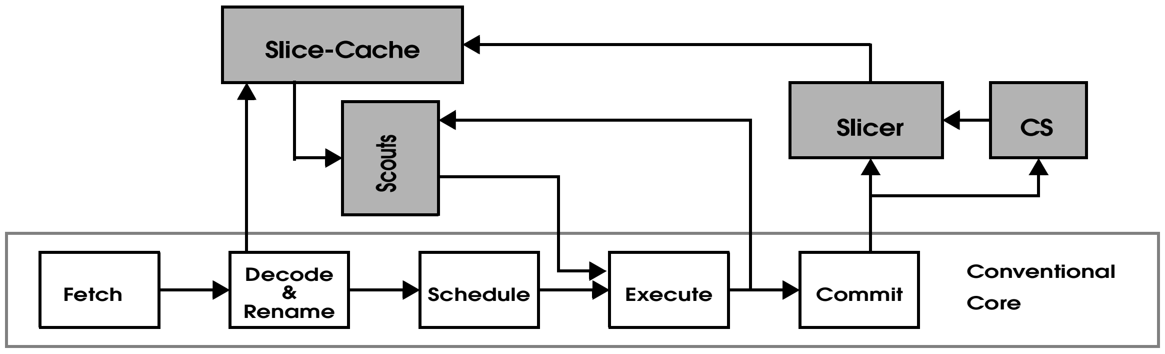


Figure 2: The Slice-processor micro-architecture. Shown in grey are the structures necessary for slice processing and how they interact with an existing processor core.

slice cache is probed as instructions in the main thread are decoded and renamed. Once a lead instruction is encountered, the slice returned from the slice cache is used to spawn a *scout thread*. The scout thread begins execution immediately and proceeds in-parallel with the main thread. Scout threads execute in one of the available scout execution units, or *scouts*. Since in this work we focus on loads that miss, no further action is necessary. Scout-thread execution may benefit performance only indirectly by prefetching data that the main thread will need. Scout thread execution is speculative and can be readily interrupted and discarded. Since scout threads may need results produced by the main thread, a communication path is provided from the complete stage of the conventional processor.

- Finally, in an outcome-based prediction we may opt for dynamic feedback mechanisms that strengthen or weaken the likelihood of reapplying a specific prediction based on past successes or failures. To simplify our design, we did not implement such feedback mechanisms in our slice processor.

Figure 2 shows the general organization of a slice processor. Shown is a conventional out-of-order core augmented with the structures necessary for slice processing. As instructions commit, they enter the slicer in program order. At the same time they probe the candidate selector CS. If a candidate instruction is encountered, we initiate slice detection. After few cycles, a full slice is produced and written into the slice cache. The slice cache is also accessed by loads as they enter the decode stage. If a matching entry is found, a copy of the slice entry and of the rename table is sent to the next available scout and the corresponding scout thread is spawned. As scout threads execute, they utilize resources left unused by the main thread. Since performance improves only indirectly when a scout thread prefetches data there is no need to communicate results from the scouts to the main processor core. In the sections that follow we discuss the various components of the slice-processor micro-architecture in detail.

3.1. Selecting Candidate Instructions

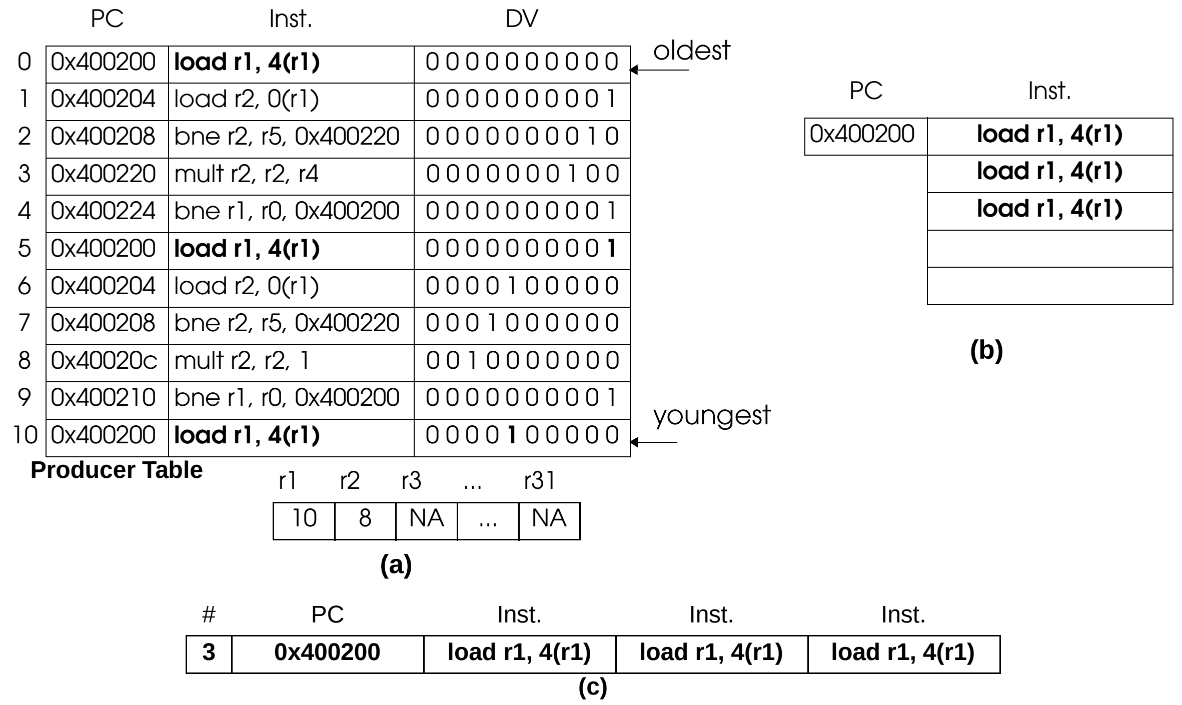
We select candidate instructions at *commit time* via a simple miss predictor. This is a PC-indexed table where entries are 4-bit counters. If the load has missed (in the L1), we increase the counter by 4, otherwise we decrease the counter by 1. A load is selected once the counter value exceeds 8. Predictor entries are allocated only when a load has missed. Similar predictors already exist in modern architectures for a different purpose. For example, the ALPHA 21264 uses a cache miss predictor for scheduling the consumers of the load [5]. However, existing miss predictors tend to favor hits over misses (common case) which is the opposite of what we want.

3.2. Detecting Slices

Upon detecting a candidate instruction we need to detect the computation slice that lead to it. This is done in the *Slicer*. Conceptually, the slicer performs a reverse walk of the dataflow graph identifying all instructions in the computation slice. It then produces a sequence containing these instructions in program order, oldest (lead) to youngest (candidate load). We present an implementation that requires no changes to existing processor mechanisms. For simplicity, we chose to ignore both memory and control-flow dependences. The implementation we simulated works as follows:

The slicer maintains a record of the past *N* committed instructions. The number of entries *N* limits how far back in the program trace we can look for identifying a slice. In our experiments we restricted *N* to 32. Instructions enter the slicer as their are committed and in program order. Each instruction occupies a single slicer entry. The entries are maintained in a circular queue. If all entries are occupied we simply discard the oldest. Slicer entries include the following fields: (1) PC, (2) Instruction, and (3) Dependence Vector, or DV. The PC field identifies the PC of the instruction stored, the instruction field contains the complete opcode, and the DV contains dependence information. The DV is a bit vector with a bit per slicer entry (i.e., 32 for our 32-entry slicer). It identifies the immediate parents of the corresponding instruction.

We explain the operation of the slicer using the example of figure 3. Shown is the slicer at the point when a candidate load instruction “load r1, 0(r1)” shown in boldface (entry 10) has just committed. (We use a simple linked list example for ease of explanation. The same mechanism will work for more elaborate address calculations.) At this point the slicer contains the instructions from the two preceding loop iterations (notice that the two iterations followed a different control-flow path). The DV of each entry identifies its immediate parents. For example the load in entry 10 depends only on the preceding instance of the same load at entry 5. Accordingly, the 5th bit (from right) of



	PC	Inst.	DV
0	0x400200	**load r1, 4(r1)**	0 0 0 0 0 0 0 0 0 0
1	0x400204	load r2, 0(r1)	0 0 0 0 0 0 0 0 0 1
2	0x400208	bne r2, r5, 0x400220	0 0 0 0 0 0 0 0 1 0
3	0x400220	mult r2, r2, r4	0 0 0 0 0 0 0 1 0 0
4	0x400224	bne r1, r0, 0x400200	0 0 0 0 0 0 0 0 0 1
5	0x400200	**load r1, 4(r1)**	0 0 0 0 0 0 0 0 0 **1**
6	0x400204	load r2, 0(r1)	0 0 0 0 1 0 0 0 0 0
7	0x400208	bne r2, r5, 0x400220	0 0 0 1 0 0 0 0 0 0
8	0x40020c	mult r2, r2, 1	0 0 1 0 0 0 0 0 0 0
9	0x400210	bne r1, r0, 0x400200	0 0 0 0 0 0 0 0 0 1
10	0x400200	**load r1, 4(r1)**	0 0 0 0 **1** 0 0 0 0 0

Producer Table

r1	r2	r3	...	r31
10	8	NA	...	NA

(a)

PC	Inst.
0x400200	**load r1, 4(r1)**
	load r1, 4(r1)
	load r1, 4(r1)

(b)

#	PC	Inst.	Inst.	Inst.
3	**0x400200**	**load r1, 4(r1)**	**load r1, 4(r1)**	**load r1, 4(r1)**

(c)

Figure 3: The Slicer: (a) Detecting a slice. (b) Constructing the slice. (c) A Slice-Cache entry.

its DV is 1. The load at entry 5 has a DV whose 0th bit 1 since it only depends on the load at entry 0. We will explain how we construct the DV vectors shortly. First, we focus on how we construct slices.

Slice construction *conceptually* proceeds in two phases (1) detecting the slice, and (2) extracting the instructions for storage in the slice cache. Detection is done by propagating a dependence vector or DV as follows: Starting from the candidate load we send its DV to the immediate preceding slicer entry (entry 9). Entry 9 checks the 9th bit of the incoming DV. Because it is zero, it just propagates the DV *as is* to entry 8. The instruction in entry 9 is not part of the slice. This process continues, until we hit an entry that corresponds to a set bit in the propagated DV. In our example, this is entry 5. This instruction belongs in the slice and for this reason it ORs its own DV with the DV it received before passing it on to entry 4. This way the instructions it depends on will also be included in the slice. This step-by-step process continues until we reach the oldest instruction in the slicer.

We then proceed to the slice construction phase where the opcodes of the slice instructions are assembled into a continuous sequence shown in part (b) and are identified by the PC of the oldest instruction which is the *lead*. A naïve implementation is to scan the entries in order inserting marked instructions as they are encountered.

In our experiments we have restricted the number of entries in a slice to 8 which is 1/4 of the size of the slicer. There are two reasons why we chose to do so. First, using a small number of instructions reduces the latency of constructing the slice-entry. Moreover, intuitively, the larger the fraction of original instructions that appear in the slice, the lower the probability that the scout-thread will run ahead of the main sequential thread. While what we have described is a sequential process where each slicer entry has to be processed in order, in practice it should be possible to process multiple slicer entries within a single cycle. In Appendix A we describe a dynamic logic implementation of the slicer.

Constructing Dependence Vectors: Constructing DVs (at insertion time) is a process similar to register renaming. We use a *producer table* (PT) having one entry per register. A PT entry points to the slicer entry of the most recent producer of that register. PT entries are set as instructions are inserted into the slicer. To construct an instruction's DV we simply lookup the producers of its source registers using the PT. For simplicity we choose to track only the integer registers in the PT. On a MIPS-like ISA such as the one we use, address calculations are usually performed using integer registers. An example of the producer table state is shown in figure 3(a). The entry for r1 points to entry 10 of the slicer, while the entry for r2 points to entry 8. Entries 3 to 31 are marked as "not available" since no instruction currently in the slicer is producing any of these registers.

Instructions Exiting the Slicer: Upon removing an instruction from the slicer, we also need to clear all references to it by subsequent slicer entries. Simply resetting all DV bits in the corresponding column is sufficient (this can be done instantaneously for all entries by providing a column reset in the DV array). We also need to update the PT. If the PT entry for the target register still points to this instruction, then we have to clear it (not available). This functionality can be incorporated in the PT cell (compare and reset on a match).

3.2.1 Slicer Considerations

Observations on Slice Form: Notice that the generated slice contains three instances of the same instruction. This is possible as it corresponds to a slice of the actual dynamic execution trace. Moreover, notice that no control flow instructions were included in the slice. While in some cases the resulting slice may be incorrect, this has only performance implications. Moreover, not including control-flow allows us to exploit slices whose instructions span over intermediate, unrelated branch instructions. In addition to not including control flow we also do not track memory dependences. As a result stores never appear in a slice. This obviates the need for handling speculative stores executed from scout threads. We do so for simplicity. However, including memory dependences requires no modifications to the slice detection mechanism. We only need to setup the DV appropriately as stores and loads commit. At commit time, the actual dependences amongst loads and stores are known (or can be determined) since all addresses have been calculated.

Keeping up with the Main Core's Commit Rate: Finally, the slicer needs to accept new instructions at the rate they are committed. However, we expect that slice detection will require multiple cycles preventing the slicer from accepting newly committed instructions in the meanwhile. The solution we chose uses two copies of the slicer: the *working slicer* and the *shadow slicer*. The working slicer always accepts entries from the commit stage. Upon detecting a candidate instruction, we copy all entries from the working slicer into the shadow slicer. By physically interleaving the two copies, we can copy all entries in parallel. Detection proceeds in the shadow slicer, while the working copy is kept up-to-date with newly committed instructions.

What we have described is just one way of performing slice detection. Further optimizations and alternative methods are possible. For example, it could be possible to construct complete DV vectors identifying the whole slice iteratively during register rename or for that matter construct the whole slice during decode (by associating slices with registers). Moreover, we may detect and record memory dependences in the load/store scheduler.

3.3. The Slice-Cache

Once slices are detected they are stored into the slice-cache for future reference. The slice-cache is a PC-indexed structure where entries contain a number of instructions comprising a slice. An example is shown in figure 3(c). The entry contains the PC of the lead instruction and the opcodes of all instructions in the slice. We store the actual instructions to avoid having to read them from the instruction cache. Finally, the number of instructions within the slice is also stored in the slice cache entry (shown under the column marked with #). The maximum number of instructions within a slice cache entry is set to equal to the maximum slice the slicer can detect (8 for most experiments). There are strong similarities between a slice-cache and a trace-cache. While the actual storage mechanism may be exactly the same, the slice-cache contains discontinuous portions of the dynamic execution trace.

3.4. Scout-Thread Spawning and Execution

Having identified scout slices and having stored them in the slice cache the next step is detecting an appropriate time for spawning a scout thread. Scout thread spawning occurs when the main sequential thread decodes a lead instruction. We spawn scout threads at decode, since the scout thread may need to read values produced by instructions before the lead one. Once the lead instruction has been decoded, it is guaranteed that all preceding instructions have also been decoded and renamed. Consequently, the scout thread can detect any dependences and wait accordingly (we will explain how shortly).

Lead instructions are simply identified by accessing the slice cache. If a matching entry is found, then we spawn a scout thread with the instructions found in the slice-cache entry. Scout threads are assigned to any of the *Scout Execution Units,* or *Scouts,* in a round-robin fashion. In our experiments we have used 8 scouts which we managed as a circular queue. For simplicity, if no scout is currently available we simply overwrite the oldest one even if it has not completed execution. Recall, that scout threads are speculative and the main thread never inspects their results. For this reason, they can be discarded at any time. Each scout contains a small instruction buffer which stores the instructions in the scout-thread. In our experiments we limit this to 8 instructions per scout. The scouts share the same functional units as the main processor. Execution of scout instructions proceeds only when there is sufficient issue-bandwidth and resources left from the main sequential thread.

In detail, spawning a scout thread is done as follows: Having read the slice out of the slice cache, we store it in the next in order scout unit. At that point we also make a copy of the main core's integer register rename table. We need this information so that we can start renaming the instructions within the slice. This is necessary, as some of the instructions within the slice may have to wait for values produced by preceding instructions from the main thread. This communication is one-way. A scout thread may read a value produced by the main thread. However, we never propagate a result from a scout thread to the main thread. We have experimented with both in-order and out-of-order instruction queues for the scout units. In either case, register results produced by scout instructions are stored in temporary storage within the scout unit. For example, we can store them into instruction queue entry of the producer. Since there are very few entries in the scout unit (8 in our experiments), a simple reservation style implementation suffices. Note that since slices do not contain stores, it is not necessary to buffer memory results separately. Loads access the L1 and the load/store queue of the conventional processor. This does not require additional ports as scout thread instructions execute only when these resources are left unused by the main thread.

Since we focus only on loads that miss, performance may improve as a result of prefetching. Accordingly, it is not necessary to communicate scout thread results to the main thread. An instruction that executes in a scout unit needs to communicate *only* with subsequent instructions within the same scout unit. However, additional paths are required to route instructions and data from the scout units to the main core's functional units.

In an processor implementing simultaneous multithreading [13] it should be possible to execute scout threads as separate threads in the conventional core. However, care must be taken to consider the overheads associated with creating an additional context for every scout thread. Moreover, in this case, care must be taken to reduce resource contention amongst the scout threads and the main thread. Such an organization is beyond the scope of this paper.

4. EVALUATION

While the idea of pre-executing program slices is appealing, the entire concept may be mute if it does not improve bottom-line performance. In this section we evaluate the performance of an implementation of slice processors using detailed, execution-driven simulation of an aggressive, dynamically-scheduled superscalar processor. The rest of this section is organized as follows: We start by describing our methodology in section 4.1. In section 4.2 we measure the potential of our method assuming an ideal slicer (i.e., zero detection latency) and experiment with both out-of-order and in-order scout units. Having demonstrated the potential of slice processors, we then investigate the impact of slice detection latency in section 4.3. To provide additional insight on the inner-workings of the slices, we present measurements on their size, frequency and resulting executed instructions in section 4.4. In sections 4.5 and 4.6 we study the effect of having a different number of scout units and of allowing longer or shorter slices. In section 4.7 we study how our slice processor interacts with a stride prefetcher. Finally, in section 4.8 we present a few of the most frequently executing slices to demonstrate that our slice processor can exploit elaborate slices.

4.1. Methodology

In our experiments we used the *Olden* pointer intensive programs and *gcc* and *perl* from SPEC95. The olden benchmarks have been used extensively in previous memory prefetching studies since they are pointer-intensive and since some of them exhibit noticeable data miss rates. We selected gcc and perl from the SPEC95 programs since they too exhibit noticeable data cache miss rates. Since in this work we used slice processors to prefetch data from memory, performance may improve only when memory stalls are a problem to start with. We have used modified input data sets to achieve reasonable simulation times. In addition, we ran all benchmarks for the first 300M committed instructions. Most benchmarks run to completion with this limit. In the interest of space, we use the abbreviations shown in table 1 in our graphs. Table 1 also reports the number of dynamic loads and the resulting miss rate.

To evaluate our slice processor we have extended the Simplescalar simulator. Our base configuration is an aggressive, 8-way dynamically-scheduled superscalar processor with the characteristics shown in table 2. We have used a 16k data cache to compensate for the reduced data sets. Moreover, our base processor implements perfect memory dependence speculation so it will send load requests as early as possible (while this reduces the positive impact of prefetching, previous work has shown that it is possible to approximate this using memory dependence prediction). Our base processor has a deep pipeline (12 cycles minimum). The slice processor augments the base configuration with a slicer, a slice cache, a candidate selector and a number scout units. The characteristics of these units are also shown in table 2. We have experimented with various slicer latencies and with both out-of-order and in-order scout units.

4.2. Performance Potential

To evaluate the potential of slice processors we first consider a 32-entry slicer that requires zero time to detect a slice. Moreover, this slicer can detect up to 8 slices simultaneously, one per committed instruction. This slicer is obviously unrealizable, but we use it to provide a measure of the performance potential of slice processors. We couple this ideal slice detector unit with realistic single-issue scout units that are either in-order or out-of-order.

Figure 4 compares the performance of the two alternatives reporting relative performance over the base processor. The dark bar corresponds to out-of-order scout units, while the gray bar to in-order ones. Some programs such as bisort and perimeter, do not benefit by the slice processor. This should not come as a surprise as these programs did not exhibit noticeable data cache miss rates to begin with. The rest of the programs where data miss rates are noticeable, do see benefits, often significant. Health in particular, more that doubles its performance (the actual performance improvements are 2.41 and 2.4 for out-of-order and in-order scout units respectively). Health's dominant data structure is a linear list which is accessed repeatedly. It is questionable whether this really represents a realistic application, but nevertheless, it demonstrates the ability of slice processors to identify this class of memory references. Em3d gains almost 27% while perl sees an improvement of about 8%. On the average over all benchmark, the slice processor improves performance by about 11%[2] While not visible, the slice processor with out-of-order scout units performs slightly worse than the base for power (the slowdown is less than 0.5%). Increased memory system contention and data cache pollution are the primary causes. References initiated by scout threads tie up memory resources (e.g., ports) and may indirectly delay subsequent requests from the main thread. In addition, prefetched memory blocks may evict cache blocks that were still needed. A potential solution would be to add a separate prefetch buffer. However, such an investigation is beyond the scope of this work.

In Figure 4 we can also see that the performance impact of in-order versus out-of-order scout units is minor. For most programs, in-order scout units either perform as well as the out-of-order ones, or trail them in performance by 1-2%. Interestingly, in-order scout units perform better than out-of-order ones for power. Out-of-order scout threads tend to make more progress more quickly. Unfortunately, since these are speculative threads, the additional work so performed is not always beneficial.

Since the performance differences between in- and out-of-order scout units are really small, and the in-order scout units are less expensive to implement, in the rest of the evaluation we will restrict our attention to in-order scout units.

4.3. Slicer Latency

To quantify the effect of an actual slicer, we modeled slicers with realistic latencies. Once a slice detection is initiated (a candidate load has committed), the slicer becomes busy for a number of cycles during which (1) no other slice detection can be initiated and (2) the slice cannot be initiated as it has not been

2. All speedup averages reported in this paper are computed using the harmonic mean.

Table 1: Programs used in our experimental evaluation. We report the dynamic committed load count and the resulting L1 data cache miss rates.

Benchmark	*Ab.*	*Loads*	*L1 Miss Rate*	*Benchmark*	*Ab.*	*Loads*	*L1 Miss Rate*
Olden							
bh	bh	73.8M	0.9%	**power**	pow	98.1M	0.1%
bisort	bis	4.9M	0.6%	**treeadd**	tad	89.0M	1.3%
em3d	em3	77.9M	27.0%	**tsp**	tsp	109.6M	2.2%
health	hlt	87.4M	17.0%	SPEC '95			
mst	mst	41.1M	6.2%	**gcc**	gcc	181.9M	2.9%
perimeter	per	56.7M	1.2%	**perl**	prl	95.6M	2.0%

Table 2: Base configuration details. We model an aggressive 8-way, dynamically-scheduled superscalar processor having a 256-entry scheduler and an 128-entry load/store queue..

Base Processor Configuration			
Branch Predictor	64K GShare+64K bi-modal with 64K selector	**Fetch Unit**	Up to 16 instr/cycle (4 branches). 64-entry buffer, non-blocking
Instruction Scheduler	256 entries RUU-like	**Load/Store Queue**	128 entries, 4 loads or stores/cycle Perfect disambiguation
Issue/Decode/Commit Bandwidth	any 8 instructions / cycle	**Functional Unit Latencies**	same as MIPS R10000
L1 - Instruction cache	64K, 2-way SA, 32-byte blocks, 3 cycle hit latency	**L1 - Data cache**	16K, 4-way SA, 32-byte blocks, 3 cycle hit latency
Unified L2	256K, 4-way SA, 64-byte blocks, 16-cycle hit latency	**Main Memory**	Infinite, 100 cycles
Slice-Processor Units			
Slicer	32-entry, not pipelined 8 instructions max per detected slice Simulated latencies: 0, 8, 16 and 32	**Slice-Cache**	1K-entry 4-way set associative 8 instructions max. per slice
Scouts	8 Units, 8 instructions max per unit In-order or out-of-order, single-issue	**Candidate Selector**	4K-entry, 4-way set associative 4-bit counters

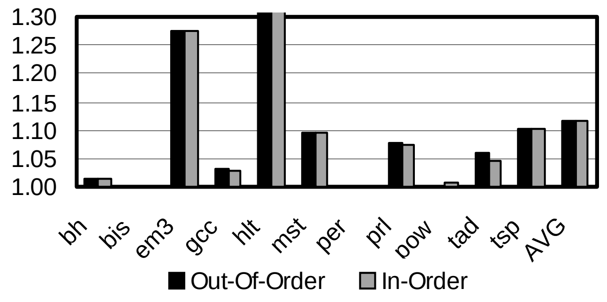


Figure 4: Speedup over base configuration for slice processors using in- and out-of-order scout units. The values for health are 2.41 (left bar) and 2.4 (right bar). Reported also is the harmonic average.

constructed yet. We simulated three slice detection latencies: 8, 16 and 32 cycles. We feel that 16 and 32 cycles are actually fairly pessimistic.

Figure 5 plots the relative performance over the base processor, of a slice processor with in-order slice execution units and realistic slice detection unit latencies. From left to right, we report slicer latencies of 0, 8, 16 and 32 cycles. We include the 0-cycle latency slicer for ease of comparison. Performance remains mostly unaffected even when the slice detection latency is as high as 32 cycles. There are two reasons why. First, in most programs, once detected, slices tend to execute many times. Delaying detection by a few cycles, may prevent spawning scout threads for the very few initial instances of the slice. Second, even when slices are not executed numerous times, slice detection takes place when a frequently missing load is encountered. In this case, often the processor is anyhow stalled waiting for memory to respond. As a result, few new instances of the slice are encountered while slice detection is in progress. Even in this case, very little is lost from not detecting the slice instantaneously. In some cases, using a higher latency slicer improves performance slightly (e.g., in tsp). A longer-latency slicer may not detect some slices that are detected by a shorter-latency slicer. These slices dot not always improve performance.

The results of this section are crucial to the viability of slice processors as they show that even under pessimistic slice detection latency assumptions, slice processors are able to sustain sizeable performance benefits.

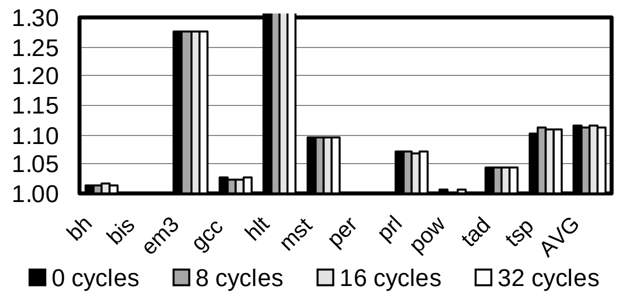


Figure 5: Performance improvements with finite slicer latencies and in-order scouts. We compare slicers with a latency of 8, 16 and 32 cycles. For ease of comparison we also include performance with a 0-cycle latency slicer. The values for health are 2.4, 2.4, 2.39 and 2.39 from left to right.

4.4. Slice Statistics

To gain some insight on the inner-workings of the slice processor, in this section we present a number of relevant statistics about the slices detected and executed in our slice processor. In particular, we report the number of unique slices detected, the instruction count distribution of slices and their instruction distance distribution. We also report the instruction overheads resulting from scout thread execution.

It is clear that the number detected slices and their length will directly affect the slice cache design. Table 3 lists the number of unique slices detected by the slice detector unit during the execution of each benchmark. It can be seen that with exception of gcc, very few unique slices are detected suggesting the most of the misses come from few loads. While the number of loads that miss may be small, there might be multiple slices that lead to a frequently missing load. To reduce the number of detected slices, we have implemented a straightforward filtering scheme (to minimize slicer pressure and slice cache conflicts). In particular, in the candidate selector we keep a single bit that indicates whether we have detected a slice for the corresponding instruction. We do not initiate slice detection if this bit is set. We have also experimented with allowing slice detection to proceed every time we encounter a frequently missing load. We found that there was very little variation in performance.

Figures 6(a) and 6(b) plot the detected and executed cumulative distribution of slice occurrence versus slice length in instructions. The distribution varies from 2 to 32 instructions for detected slices and 2 to 8 instructions for executed slices (we execute slices of up to 8 instructions). We can see that the detected slice length distribution suggests that a significant fraction of all detected slices (10%) are longer than ten instructions. This fraction is larger for mst and perimeter. However, we restricted execution to only those slices that were up to 8 instructions in length (later on we show results for longer slices). As we explained in section 3.2, we did so as slices that contain a large fraction of all instructions are less likely to run ahead of the main thread. For most programs, the majority of executed slices have up to 4 instructions. Em3d, perimeter, power and tsp exhibit strongly biased distributions with virtually all executed slices having 4 instructions. Other programs tend to exhibit more of a linear distribution with noticeable fractions of longer slices

Table 3: Number of unique slices detected per benchmark.

Program	*Unique Slices*	*Program*	*Unique Slices*
Bh	205	***Perimeter***	43
Bisort	13	***Perl***	381
Em3d	58	***Power***	73
Gcc	8742	***Treeadd***	53
Health	208	***Tsp***	117
Mst	72		

In addition to how many instructions appear in slices, another interesting statistic is the dynamic instruction distance of a slice. We define *dynamic instruction distance* of a slice to be the number of dynamic instructions that appear in between the slice's lead (oldest in program order) and the candidate load (youngest in program order). This includes all instructions both those that belong in the slice and those that do not. This distance is loosely related to the probability of a scout thread running ahead of the main thread. A longer distance increases the chances of running ahead of the main thread. Figures 7(a) and 7(b) report the cumulative distributions for detected and executed slices respectively. These distributions vary from 1 to 32 instructions as our slicer has 32 entries. Compared to the number of instructions appearing in the slices (figures 6(a) and 6(b)), we can see that even though slices contain few instructions, they actually spread over a lot more instructions. For example, more than 50% of executed slices in gcc spread over 24 or more dynamic instructions of the main thread.

Table 4 shows the overhead instructions executed by scout threads. We express these as a fraction of all committed instructions by the main thread. Overhead ranges greatly, from single digit up to about 44%. However, with the exception of power, this overhead computation does not hurt performance. When loads miss often, chances are that the main core is under-utilized. Consequently, in such cases, the demands placed by scout threads can be easily satisfied. Furthermore, scout execution consumes resources only not used by the core processor, and in this way can affect the performance only in an indirect way. Lastly there is a good correlation between overhead computation and performance improvement for most of the benchmark programs, indicating that the slice processor is rarely wasting resources.

4.5. Sensitivity to the number of Scout Units

We also experimented with varying the number of scout units. In particular, we also simulated slice processors with 4 and 16 scout units. Using a smaller number of scout units clearly reduces the resource requirements for slice processors. However, it may also result in many slices being prematurely overwritten long before they had a chance of prefetching data.

Figure 8(a), compares the performance improvements over the base processor for 4, 8 and 16 scout units from left to right. Noticeable variation is observed for em3d, perl and treeadd. Interestingly, for treeadd increasing the number of scout units

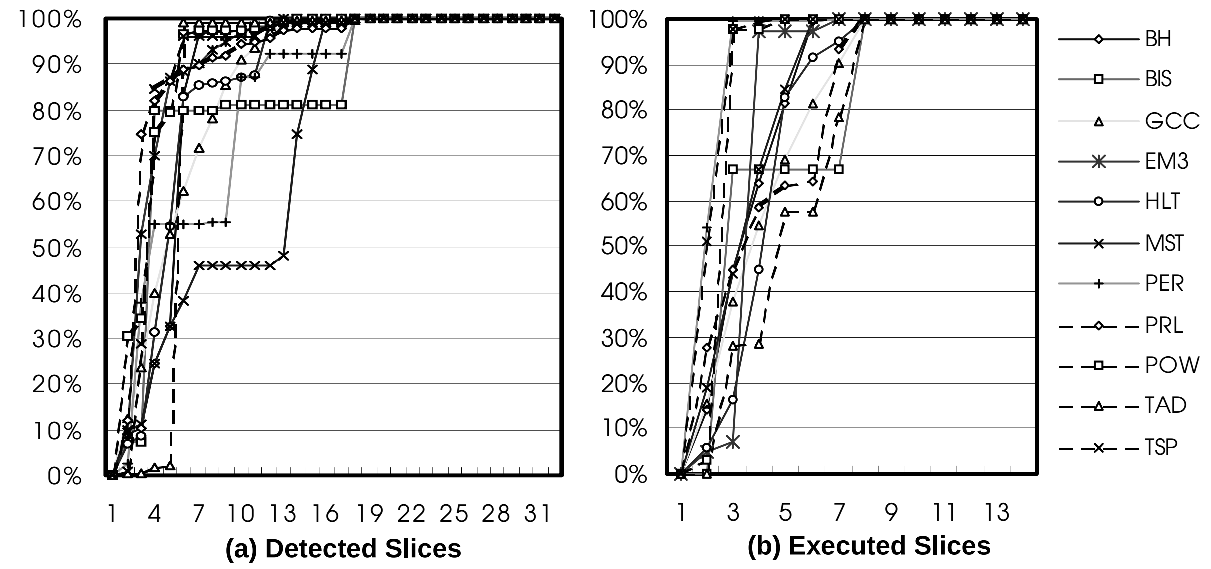


Figure 6: Cumulative distribution of slice instruction count. (a) Detected slices, (b) executed slices.

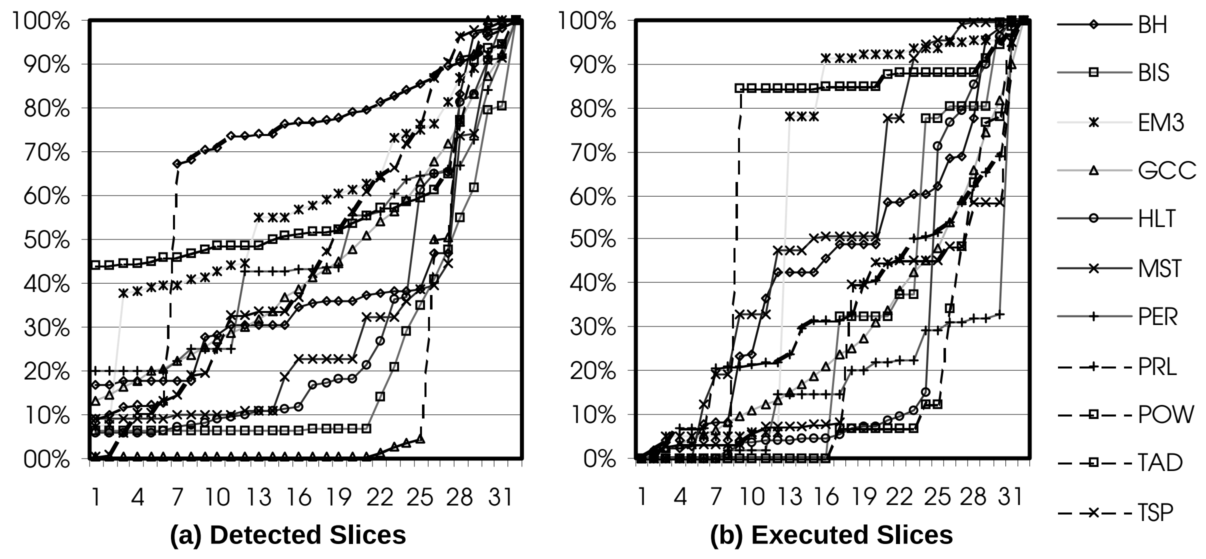


Figure 7: Dynamic instruction distance for (a) detected slices and (b) executed slices. We define instruction distance to be the number of dynamic instructions between the lead instruction (oldest instruction in the slice) and the candidate load (youngest instruction in the slice). This includes all instructions independently on whether they belong in the slice or not.

actually hurts performance. Again, this is the result of increased memory pressure and data cache pollution. Treeadd's core data structure is, as the name implies, a tree. Since our slices are oblivious to intermediate control-flow, they correspond to specific paths through the tree's branches. When more scout threads are simultaneously active, it is likely that some (if not most of them) will be prefetching down the wrong path polluting the data cache and consuming memory resources. When the number of scout units is small, however, fewer scout threads are active, and few of them get to prefetch deep down a mis-speculated path. Since em3d's access sequence is more predictable (linked list), having more scout units helps (8 vs. 4), However, at some point we saturate the memory system so adding more units does not impact performance (i.e., going from 8 to 16). For some programs (e.g. perl) we observe fluctuations in performance depending on the number of scout units.

4.6. Slice Length Sensitivity

Finally, we also take a look at the sensitivity of our slice processor to the length of the slices that can be detected and executed. It would seem that having longer slices may increase the chances of a scout thread running ahead of the main thread. However, as we explained, care must be taken to also take into

Table 4: Slice Execution Overhead for each benchmark. Reported is the number of instructions executed by scout threads as a fraction of the instructions committed by the main thread.

Program	Overhead Instr. (%)	Program	Overhead Instr.(%)
Bh	7.9%	***Perimeter***	11.5%
Bisort	0.5%	***Perl***	20.5%
Em3d	17.4%	***Power***	1.1%
Gcc	35.5%	***Treeadd***	39.9%
Health	44.3%	***Tsp***	18.3%
Mst	10.6%		

account the fraction of overall computation such slices represent. Also, longer slices may spread over multiple branches, giving rise to the danger of capturing computations whose utility is limited to a specific control-flow paths.

Figure 8(b) reports performance relative to the base processor for slices of up to 4, 8 and 16 instructions from left to right. The slicer in all experiments is left at 32-entries. For most programs, allowing longer slices increases performance. In mst and to a lesser extend in tsp we observe a large jump in performance when we move from 8 instruction slices to 16 instruction ones. As we have seen in figure 6(a), many detected slices in these programs had more than 8 instructions. Note, that using longer slices does not always improve performance. For example, performance drops (albeit only slightly) in perl when we move from 8 instruction slices to 16 instruction ones.

We have also experimented with larger slicer windows and found that after a while, slice utility drops. Even though we were able to capture slices that spread over more instructions, they were less likely to prefetch useful data. In addition we have experimented with *partial window slicers*, which attempt to extend the reach of the slices by admitting in the detection window only integer instructions and loads. Based on the observation that address calculation is achieved with arithmetic and logical instructions and by loads, partial window slicers do *not* admit branches, stores and floating point instructions in their detection window. In this way the resulting slice for a slicer of 32 entries may contain instructions that are farther than 32 instruction apart in the dynamic instruction stream of the program. However, our experiments showed that the partial slicers overall performed very similarly to regular slicers with the same number of entries

4.7. Interaction with a Stride Prefetcher

Finally, we present a preliminary investigation of how slice processors interact with an out-come-based prefetcher. For this purpose, we have augmented our base configuration with a stride prefetcher. Our stride prefetcher is loosely based on the model by Farkas at al. [4]. In particular, it uses a 4K-entry PC-indexed table to detect memory operations that follow a stride pattern. It uses ttwo bit saturating counters with a threshold of 2 for initiating prefetching requests only for instructions that exhibit repeating stride access patterns. Once such an instruction is identified, prefetching requests are sent whenever it is executed. The target address is calculated as “addr x stride x lookahead” where “addr” is the address just accessed, “stride” is the identified stride, and “lookahead” has been tuned to maximize benefits for the given configuration (32 in our case). Prefetching requests are queued only if no other request for the same block already exist. They compete for load/store ports with the main core (scout threads also did the same).

Figure 9 reports performance improvements for three configurations. Starting from the left is the slice processor. It has a 32 cycle slicer, scout threads of up to 8 instructions and 8 scout units. The second, is the base configuration augmented with the aforementioned stride prefetcher. Finally, we combined the slicer processor with the stride prefetcher. The stride prefetcher offers significant performance benefits for em3d, perl and treeadd. In all three programs, combining the stride prefetcher with the slice processor leads to further performance improvements. However, combining the two is not always beneficial (e.g., gcc). This is primarily the result of increased contention for memory ports and of cache polution..

The results of this section demonstrate that our slice processor is capable of prefetching access patterns that foil an existing stride-based prefetcher. As we explained in section 2, more sophisticated outcome-predictors are possible. Similarly, further improvements may be possible over the first-cut mechanisms used in our slice processor. Accordingly, while these results are promising, further investigation is required to better understand the interaction of slice processors and outcome-based predictors.

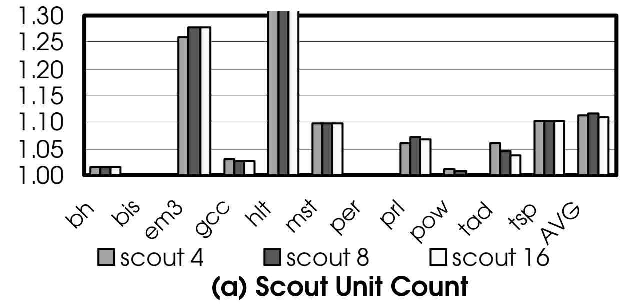
4.8. Slice Examples

Figure 10 shows some frequent slices from perl, gcc and em3d. The slice from perl (part (a)) implements a switch statement. We show both the C code (excerpt) and the resulting slice in machine code. Note that the instructions in the slice do not form a continuous sequence. Part (b) shows a slice from gcc where an array element is accessed. The index of the element is stored in a structure pointed to by some other variable. Finally, in part (c) we show the most frequently occurring slice in em3d that corresponds to a linked list traversal. This slice contains 4 instances of a self-recurring load.

5. RELATED WORK

Slice-processors utilize small helper threads that run in parallel with a main, sequential thread. *Simultaneous subordinate micro threading* (SSMT) first proposed a model of executing such helper threads [1]. In SSMT, helper threads are stored in microcode memory and are executed upon encountering special thread launching instructions in the main thread. As an example, it is shown how SSMT can be used to implement a sophisticated branch prediction algorithm. *Assisted execution* is a similar proposal [13].

The idea of identifying slices and pre-executing them to improve performance is not new. Farcy, Temam and Espasa suggested to dynamically extract slices for mis-predicted branches and used them to pre-calculate future branches for the purposes of branch prediction [3]. Their computation detection mechanism works by identifying a restricted set of well behaved computations. Zilles and Sohi suggested using pre-execution in a



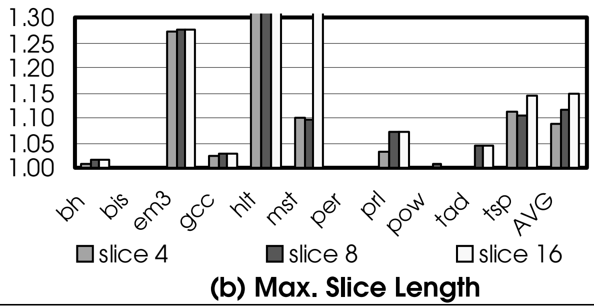


Figure 8: (a) Performance for various in-order scout unit counts. From left to right: 4, 8 and 16 scout unit slice processor performance. The values for health are 2.35, 2.40 and 2.40 from left to right. (b) Performance for slices of 4, 8 and 16 maximum instructions (left to right) for in-order scout units. The values for health are 1.73, 2.40 and 2.58 from left to right, and right-most value for mst is 1.48.

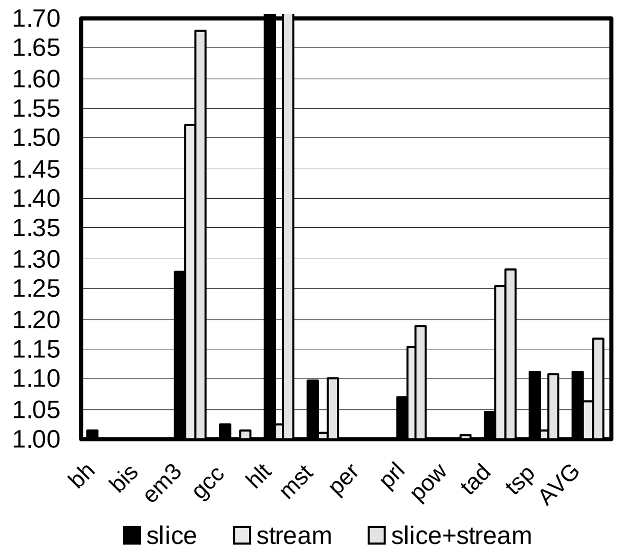


Figure 9: Comparing a slice processor with an outcome-based stride prefetcher ("stream"). The two out-of-range numbers for health are both 2.39. Reported is relative performance over the base configuration.

general sense for both branches and loads. They studied the characteristics of the computation slices leading to branches and loads including both control flow and memory dependences [15]. In parallel with this work, they also demonstrated that using optimized speculative slices identified by profiling for both loads and branches can greatly improve performance [16]. Moshovos suggested the possibility of generalized operation-based predictors as an improvement over outcome-based ones [8].

Roth and Sohi proposed the *Speculative Data-Driven Multithreading (SDDM)* execution model [11]. SDDM is a general model where performance critical slices leading to branches or frequently missing loads are pre-executed. *Register integration* mechanism incorporates the results of pre-execution directly into the main sequential thread avoiding the need for re-executing these instructions. It also filters incorrect pre-computed results. In their study and for the purposes of demonstrating the potential of the SDDM model, slices are identified through profiling. Our slice-processor performs automatic and dynamic detection of slices leading to loads. Moreover, execution of slices impacts the main sequential thread only indirectly and as a result no integration mechanism is necessary (at the expense of re-executing instructions).

A number of other proposals investigate dynamic creation of helper-threads. In particular the *Instruction Path Coprocessor* proposal introduces a coprocessor that can dynamically optimize program traces [2]. It is shown the this approach can be used to implement previously proposed prefetchers targeting arrays and linked lists. It may be possible to use such a coprocessor to identify slices and pre-execute them.

Many prefetching methods have been proposed, both static and dynamic. We do not review them due to space limitations. Most of them rely on some form of outcome-based address prediction. A number of operation-based address predictors also exist. For example, stride predictors fall in this category. Roth, Moshovos and Sohi also proposed an operation-based predictor for linked data structures which is based on detecting load to load dependences [9]. Mehrotra at el., also proposed operation-based predictors for arrays and linked lists [7]. Roth, Moshovos and Sohi extended their operation-based prefetcher for pre-executing indirect branches [10]. In all aforementioned proposals, the class of predictable operations is fixed in the design. Our slice processor architecture is capable of extracting and predicting generalized operations. *Slipstream Processors* also use a helper thread to run-ahead of the main sequential thread in effect pre-executing instructions [4]. The helper thread is formed by removing predictable computations from the main sequential thread.

As described, our slice processor is capable of a restricted form of access/execute decoupling. Smith proposed access/execute decoupled architectures [12]. In these architectures, program execution comprises two threads, one responsible for accessing memory and one responsible for computations. Partitioning of instructions between the two threads is done statically. Our slice processor implements a restricted form of access/execute decoupling while detecting and spawning access threads dynamically and speculatively.

```
        char *s = bufptr;
        ...
        switch (*s)
        ...
lb      r3, 0(r18)
sll     r2, r3, 2
lui     r1, 0x1000
addu    r1, r1, r2
lw      r2, 15816(r1)
```

(a) slice from perl

```
... = array[*ptr->field)

lw      r19, 0(r16)
lw      r3, 4(r19)
addu    r4, r0, r3
sll     r6, r4, 1
lw      r2, -19544(r28)
addu    r6, r2, r2
lh      r2, 0(r2)
```

(b) slice from gcc

```
list = list->next

lw      r4, 8(r4)
lw      r4, 8(r4)
lw      r4, 8(r4)
lw      r4, 8(r4)
```

(b) slice from em3d

Figure 10: Examples of slices that are detected and executed by the slice processor.

6. CONCLUSION

We have described the micro-architecture of the Slice-Processor. Slice-Processors are an implementation of operation prediction where rather than detecting repeating patterns in the outcome-stream of programs we instead attempt to predict repeating patters in the computation (or operation) used to produce these outcomes. We have explained that operation prediction has the potential to predict outcomes that foil existing outcome-based predictors. However, we have also emphasized that operation-prediction should not be viewed as a replacement for outcome-based predictors.

We have restricted our attention to using slice processors for memory prefetching and focused on describing the various structures necessary for detecting, extracting and executing slices. In particular, we have provided a detailed description of the structures necessary for implementing a simple "last operation"-based predictor. We have provided a detailed description of the slice detection unit (the slicer) and studied its performance under various latency assumptions. We have demonstrated that a simple slice processor implementation can improve performance by about 11% on the average for a set of pointer-intensive benchmarks and for perl and gcc from Spec95. In particular, this implementation is capable of detecting slices of up to 8 instructions over a region that spans at most 32 instructions. The particular implementation uses 8 single-issue scout execution engines (scouts) capable of in-order execution. We have also found that allowing slices of up to 16 instructions can improve performance by 15%. We have provided additional statistics to gain additional insight on the type and frequency of slices that are commonly detected. Finally, we compared our slice processor with a stride-based prefetcher and found that often they benefit from each other.

Many directions for further research and improvements exist including applying the idea of operation-prediction to other forms of prediction such as branch and value prediction. Also, our slice processor currently implements a simple *"last-operation"* prediction scheme. Building on the experience with outcome-based predictors, it may be possible to use pattern detectors to enhance accuracy and utility (e.g., slice A appears after slices B and C have appeared). Nevertheless, we have demonstrated that a simple implementation of slice processors can improve performance significantly even under pessimistic assumption about slice detection (up to 32 cycles of latency to scan over 32 instructions) and execution (sharing resources with the main thread).

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers, Doug Carmean, Stephan Jourdan and Eric Sprangle for their insightful comments. Andreas Moshovos is on leave of absence from Northwestern University. This work was supported in part by an NSF CAREER award and by funds from the University of Toronto. This work was also supported by the Greek Secretarial for Research and Technology (GSRT) and the European Social Fund through the PENED 99 program under contract 99ED408.

References

[1] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. 26th Intl. Symposium on Computer Architecture*, May 1999.

[2] Y. Chou and J. Shen. Instruction path coprocessors. In *Proc. 27th Intl. Symposium on Computer Architecture*, June, 2000.

[3] A. Farcy, O. Temam, and R. Espasa. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proc. 31st Annual International Symposium on Microarchitecture*, Dec. 1998.

[4] K. Farkas, P. Chow, N. Jouppi and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. 24th International Symposium on Computer Architecture*, June 1997.

[5] K. Sundaramoorthy, Z. Purser and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[6] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 architecture. In *Proc. of the Intl. Conference on Computer Design*, Dec. 1998.

[7] S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proc. 10th Intl. Conference on Supercomputing*, Sept. 1997.

[8] A. Moshovos. *Memory Dependence Prediction, Chapter 6, Future Directions: Operation Prediction*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI, Dec. 1998.

[9] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. 8th*

International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1998.

[10] A. Roth, A. Moshovos, and G. S. Sohi. Improving virtual function call target prediction via dependence-based pre-computation. In *Proc. Intl. Conference on Supercomputing-99*, June 1999.

[11] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th International Symposium on High Performance Computer Architecture*, Jan, 2001.

[12] J. E. Smith. Decoupled Access/Execute computer architectures. In *Proc. 9th Intl. Symposium on Computer Architecture*, April 1982.

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

[14] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[15] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. 27th International Symposium on Computer Architecture*, June 2000.

[16] C. Zilles and G. Sohi. Execution-based Prediction using Speculative Slices In *Proc. 28th International Symposium on Computer Architecture*, June 2001.

Appendix A

Here we describe a sketch of an implementation of the slice detection mechanism. In section 3.2, we have described slice detection as a step-by-step process where a DV (dependence vector) is propagated in reverse program order among all instructions in the slicer. The bits that are set in the DV identify the instructions that are currently part of the slice. Upon receiving the DV, a slicer entry inspects whether the bit that corresponds to it is set. If so, it has to OR the received DV with its own and propagate it to the next entry. Otherwise, it simply propagates the DV as is.

Figure 11 shows an abstract dynamic logic implementation that should be able to process multiple slicer entries per cycle. Shown within the dotted box is the circuitry for entry #1 (entries are numbered starting from 0). The entry contains a number of storage elements, shown as the boxes numbered as 0 through 3 (for ease of explanation, we assume a 4-entry slicer). These storage elements hold the DV for the entry. As we explained, the DV identifies the immediate parents of this entry.

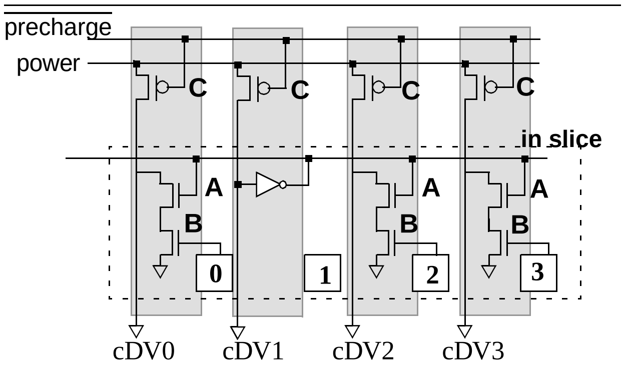


Figure 11: An implementation of the Slice Detection Circuitry. We assume a 4-entry slicer and show the circuit for entry 1.

Detecting the slice proceeds as follows: A set of vertical cDV (cumulative dependence vector) lines cross over all entries, one per DV bit. These lines are initially pre-charged using the transistors marked with C. Each slicer entry simply observes the cDV line that corresponds to it. Since we are showing entry #1, a NOT gate is connected to line cDV1 (an additional activate line may be required to disable the NOT gate during pre-charge). The output of the NOT gate connects to a set of the discharge chains one per each other storage element (DV). As a result, if cDV1 is discharged, the output of the NOT gate will be set to 1, activating all pass-transistors marked with A. If the corresponding storage element also holds a 1, then the pass-transistor B will also be activated and the corresponding cDV line will be discharged too (we omit the pull-down transistors that are used to isolate A and B during precharge). In effect, we have implemented a conditional OR function, discharging the cDV lines that correspond to the immediate parents of this entry if the incoming cDV indicated that this entry belongs in the slice. Provided that we allow sufficient time, this discharging will eventually identify all instructions in the slice. The whole detection process is initiated by discharging the DV line for the candidate instruction (i.e., the youngest one).