Execution-based Prediction Using Speculative Slices

Craig Zilles and Gurindar Sohi
Computer Sciences Dept. University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706-1685, USA
[zilles,sohi]@cs.wisc.edu

Abstract

A relatively small set of static instructions has significant leverage on program execution performance. These problem instructions contribute a disproportionate number of cache misses and branch mispredictions because their behavior cannot be accurately anticipated using existing prefetching or branch prediction mechanisms.

The behavior of many problem instructions can be predicted by executing a small code fragment called a speculative slice. If a speculative slice is executed before the corresponding problem instructions are fetched, then the problem instructions can move smoothly through the pipeline because the slice has tolerated the latency of the memory hierarchy (for loads) or the pipeline (for branches). This technique results in speedups up to 43 percent over an aggressive baseline machine.

To benefit from branch predictions generated by speculative slices, the predictions must be bound to specific dynamic branch instances. We present a technique that invalidates predictions when it can be determined (by monitoring the program's execution path) that they will not be used. This enables the remaining predictions to be correctly correlated.

1 Introduction

Wide-issue microprocessors are capable of remarkable execution rates, but they generally achieve only a fraction of their peak instruction throughput on real programs. This discrepancy is due to *performance degrading events* (PDEs), largely branch mispredictions and cache misses. This work attempts to avoid (or reduce the latency of) stalls due to branch mispredictions and cache misses.

Performance degrading events tend to be concentrated in a subset of static instructions whose behavior is not predictable with existing mechanisms [1, 8]. Current branch predictors attempt to identify patterns in branch outcomes and exploit them to accurately predict those branches whose behaviors exhibit these patterns. Similarly, caches service most requests by exploiting temporal and spatial locality, and hardware prefetching detects and anticipates simple memory access patterns. Because the "simple" instructions are handled accurately by existing mechanisms, most of the branch mispredictions and cache misses are concentrated in the remaining static instructions whose behavior is more "complex."

In Section 2, we demonstrate that a small set of frequently executed static instructions, which we refer to as *problem instructions*, are responsible for causing a majority of PDEs. In addition, we show that avoiding the mispredictions and cache misses caused by problem instructions gives much of the benefit of a perfect cache and branch predictor.

To avoid cache misses and branch mispredictions associated with problem instructions, we construct a code fragment that mimics the computation leading up to and including the problem instruction. We call such a piece of code a *speculative slice* [19]. Like a *program slice* [18], a speculative slice includes only the operations that are necessary to compute the outcome of the problem instruction. In contrast, a speculative slice only needs to approximate the original program (*i.e.*, it need not be 100 percent accurate), resulting in significant flexibility in slice construction.

Constructing efficient, accurate speculative slices is feasible for many problem instructions in the benchmarks studied. In Section 3, we describe the slices that we constructed by hand and the optimization techniques used. Typically, the slices cover multiple problem instructions and generate a prefetch or prediction every 2-4 dynamic instructions executed by the slice; branch predictions generated by the slices are greater than 99% accurate.

Speculative slices execute as *helper threads* on a multithreaded machine. These helper threads accelerate the program's execution microarchitecturally, by prefetching data into the cache and generating branch predictions, but they do not affect architected state; a slice has its own registers and performs no stores. Forking the slice many cycles before the problem instructions are encountered by the main thread provides the necessary latency tolerance. After copying some register values from the main thread, the slice thread executes autonomously. The necessary extensions to a multithreaded machine to support these helper threads are described in Section 4.

An important component of this additional hardware binds branch predictions generated by helper threads to the correct branch instances in the main thread. When a prediction is generated by a slice, it is computed using a certain set of input values and assumptions about the control flow taken. Due to control flow in the program not included in the slice, not all of the generated predictions will be consumed.

Table 1. ***Simulated machine parameters***

Front End	A 64KB instruction cache, a 64Kb YAGS [6] branch predictor, a 32Kb cascading indirect branch predictor [5], and a 64-entry return address stack. The front end can fetch past taken branches. A perfect BTB is assumed for providing target addresses, which are available at decode, for direct branches. All nops are removed without consuming fetch bandwidth.
Execution Core	The 4-wide machine has a 128-entry instruction window, a full complement of simple integer units, 2 load/store ports, and a single complex integer unit, all fully pipelined. The pipeline depth (and hence the branch misprediction penalty) is 14 stages. For simulation simplicity, scheduling is performed in the same cycle in which an instruction is executed. This is equivalent to having a perfect cache hit/miss predictor for loads, allowing the scheduler to avoid scheduling operations dependent on loads that miss in the cache. Store misses are retired into a write buffer. The 8-wide machine is similar, except it has a 256-entry window and 4 load/store units.
Caches	The first-level data cache is a 2-way set-associative 64KB cache with 64 byte lines and a 3-cycle access latency, including address generation. The L2 cache is a 4-way set-associative 2MB unified cache with 128-byte lines and a 6-cycle access. All caches are write-back and write-allocate. All data request bandwidth is modeled, although writeback bandwidth is not. Minimum memory latency is 100 cycles.
Prefetch	In parallel with cache accesses, a 64-entry unified prefetch/victim buffer is checked on all accesses. A hardware stream prefetcher detects cache misses with unit stride (positive and negative) and launches prefetches. In addition, when bandwidth is available, sequential blocks are prefetched (before a stride is detected) to exploit spatial locality beyond 64 bytes.

A mechanism is required to determine which predictions should be ignored. In Section 5, we present our correlation technique that monitors the path taken by the main thread to identify when a prediction can no longer be used by its intended branch.

In Section 6, we present the performance results attained by executing speculative slices in the context of a simultaneous multithreading (SMT) processor. We show that the slices accurately anticipate the problem instructions they cover and provide speedups between 1% and 43%.

2 Problem Instructions

In this section, we present a brief characterization of problem instructions. Because such a characterization depends on the underlying microarchitecture (*e.g.*, better predictors and caches reduce the number of problem instructions), we first present our hardware model and simulation methodology. In Section 2.2, we demonstrate that there is a set of static instructions that accounts for a disproportionate number of branch mispredictions and cache misses. Then, in Section 2.3, we show that this concentration of PDEs causes these instructions to have significant performance impact on the execution. In Section 2.4, we present a source-level example of these instructions to provide a conceptual understanding of why these instructions are difficult to handle with conventional caches and branch predictors.

2.1 Methodology

The underlying microarchitecture is an aggressive, heavily-pipelined, out-of-order superscalar processor. It includes large branch predictors, large associative caches, and hardware prefetching for sequential blocks. Table 1 provides the details of our 4- and 8-wide simulation models, which are based loosely on the Alpha architecture version of SimpleScalar [2]. We perform our experiments on the SPEC2000 benchmarks. In general, these programs exhibit hard-to-predict branches and some non-trivial memory access patterns. We have used input parameters that roughly maintain the working set size of the reference inputs, while reducing the run lengths to 9 to 40 billion instructions.

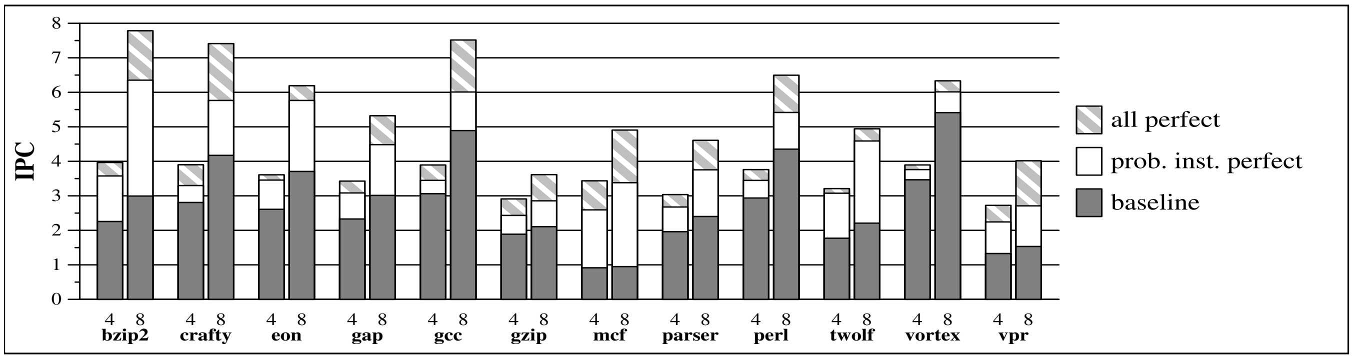
2.2 Characterization

For each benchmark, we attribute PDEs (*i.e.*, cache misses and branch mispredictions) to static instructions. If a static instruction accounts for a non-trivial number of PDEs and at least 10% of its executions cause a PDE, it is characterized as a problem instruction. This classification is somewhat arbitrary and serves merely as a demonstration that an uneven distribution of PDEs exists. Table 2 characterizes the problem instructions occurring in full runs of the benchmarks, showing that these instructions are responsible for a disproportionate fraction of performance degrading events.

Table 2. ***Coverage of performance degrading events by problem instructions.*** *The first group of columns shows how many static loads and stores (**#SI**) were marked as problem instructions, what fraction of all memory operations these instructions comprise (**mem**), and the fraction of L1 cache misses covered (**mis**). The second three columns give similar information for branches (**br** = % of dynamic branches, **mis** = % of mispredictions).*

	Memory Insts			Control Insts		
Program	**#SI**	**mem**	**mis**	**#SI**	**br**	**mis**
`bzip2`	75	5%	97%	86	30%	83%
`crafty`	103	2%	70%	108	10%	40%
`eon`	*insufficient misses*			45	23%	88%
`gap`	140	4%	98%	193	12%	69%
`gcc`	429	1%	59%	530	4%	43%
`gzip`	30	21%	96%	79	16%	62%
`mcf`	127	32%	97%	34	23%	71%
`parser`	157	5%	82%	164	14%	51%
`perl`	140	2%	67%	82	9%	68%
`twolf`	211	10%	93%	167	51%	91%
`vortex`	148	1%	49%	142	1%	58%
`vpr`	186	14%	92%	95	16%	75%

Figure 1. ***Performance impact of problem instructions.*** *The bottom bar shows the IPC of the baseline configuration, the second bar shows the performance improvement of avoiding the data cache misses and branch mispredictions associated with the problem instructions presented in Table 2, and the top bar shows the performance achievable if all data cache misses and branch mispredictions are avoided. Data shown for both 4- and 8-wide configurations for the SPEC2000 benchmarks.*



2.3 Performance Impact

To observe how the PDEs caused by problem instructions affect performance, we augmented our simulator to give the appearance of a perfect branch predictor and perfect cache on a per static instruction basis. Figure 1 shows that removing the PDEs from problem instructions provides a substantial increase in performance. In fact, these instructions account for much of the performance discrepancy between the baseline model and the *all perfect* model. Because the impact of the PDEs is larger in the 8-wide machine, the potential speedups from perfecting problem instructions are larger. Notice that even with perfect data caches and perfect branch prediction, most benchmarks fail to achieve the peak throughput of the machine due to the limited instruction window, real memory disambiguation, and operation latency.

2.4 Source-Level Example

We present a concrete example of problem instructions from the SPEC2000 benchmark `vpr`. This example performs a heap insertion. A *heap* is a data structure that supports removal of the minimum element and random insertions [4]. The structure is organized as a binary tree and maintains the invariant that a node's value is always less than that of either of its children. The heap is stored as an array of pointers, such that if a node's index is *N*, its children are found at indices *2N* and *2N+1*.

Insertions are performed (in the function `add_to_heap`, shown in Figure 2) by adding the new element to the end of this array (at `heap_tail`, line 1), creating a new leaf node of the tree. At this point, we need to ensure that the heap invariant is maintained, so we need to check to see if the new node (`ifrom`) is less than its parent (`ito`) and, if so, swap the two pointers (lines 7-9). This test is performed recursively until the heap invariant is satisfied (line 6) or the root of the tree is reached (line 5).

In `vpr`, the heap size is thousands of elements, so the whole structure does not fit in the L1 cache. This means that accessing the `cost` field of the `ito` element (in line 6) typically incurs a cache miss. Due to variation in the key values associated with the inserted elements, different insertions "trickle" different distances up the tree. An average loop count of 2-3 iterations causes the comparison branch (also in line 6) to be unbiased and frequently mispredicted.

This example demonstrates a number of common attributes of problem instructions. Problem loads typically involve dereferencing at least one pointer; dereferencing pointer *chains* is common. This can make them difficult to prefetch using existing methods. Problem branches are typically unbiased and involve a test on a recently loaded data element. Problem instructions are frequently dependent upon one another, as the problem branch in the example is data-dependent on the problem load, and the load is control-dependent on the problem branch from the previous iteration. Lastly, problem instructions often occur in tight loops. In the next section, we discuss how to tolerate problem instructions.

3 Speculative Slices

Given that problem instructions are not easily predicted with state-of-the-art predictors, a different approach is

Figure 2. ***Example problem instructions from heap insertion routine in*** **`vpr`*.***

```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

  void add_to_heap (struct s_heap *hptr) {
    ...
1.   heap[heap_tail] = hptr;     branch
2.   int ifrom = heap_tail;      misprediction
3.   int ito = ifrom/2;
4.   heap_tail++;                          cache miss
5.   while ((ito >= 1) &&
6.         (heap[ifrom]->cost < heap[ito]->cost))
7.       struct s_heap *temp_ptr = heap[ito];
8.       heap[ito] = heap[ifrom];
9.       heap[ifrom] = temp_ptr;
10.      ifrom = ito;
11.      ito = ifrom/2;
    }
  }
```

required to predict them correctly. The key insight is that the program is capable of computing all addresses and branch outcomes in the execution; it just does so too late to tolerate the latency of the memory hierarchy (for loads) or the pipeline (for branches). To predict the behavior of problem instructions, we execute a piece of code that approximates the program. We extract the subset of the program necessary to compute the address or branch outcome associated with the problem instruction. We refer to this subset as the *slice* of the problem instruction.

Our previous research [19] observed that it is not profitable to build conservative slices (*i.e.*, those that compute the address or branch outcome 100% accurately) because such slices must include a significant portion of the whole program. Instead, by using the slices to affect only microarchitectural state (through prefetching and branch prediction hints), we remove the correctness constraint on the slices. This speculative nature provides significant flexibility in slice construction, enabling smaller slices that can provide more latency tolerance with lower overhead.

We have found it feasible to construct efficient and accurate slices for many problem instructions. In this section we characterize the speculative slices we constructed, discussing the ways in which slices enable early execution before describing the slice construction process using our running example. The slices for this paper were constructed manually. They are meant to serve as a proof of concept and to suggest the set of techniques that should be used when constructing slices automatically. Automatic slice construction is discussed briefly in Section 3.3.

3.1 How Speculative Slices Enable Early Execution

The benefit of executing a speculative slice is derived from *pre-executing* (*i.e.*, computing the outcome of) a problem instruction significantly in advance of when it is executed in the main program. Before discussing our speculative slices, we review how a slice enables early execution of an instruction relative to the whole program:

- Only instructions necessary to execute the problem instruction are included in the slice. Thus, the slice provides a faster way to fetch the computation leading to a problem instruction than does the whole program.
- The slice avoids misprediction stalls that impede the whole program. Control flow that is required by the slice is if-converted, either arithmetically or through predication (*e.g.*, conditional move instructions). Unnecessary control flow is not included.
- The code from the original program can be transformed to reduce overhead and shorten the critical path to computing a problem instruction. Transformations can be performed on the slice that were not applied by the compiler for one of two reasons: (1) not provably safe for all paths — speculative slices neither have to be correct nor consider all paths — or (2) not feasible in the whole program due to limited resources (*e.g.,* registers).

3.2 Speculative Slice Construction

Slice construction is most easily understood in the context of a concrete example. Using the code example from `vpr`, we demonstrate many of the characteristics of slices and the techniques we use to construct them.

Slice Structure. Because the two problem instructions in this example are inter-dependent, we construct a single slice that pre-executes both. In general, slice overhead can be minimized by aggregating all problem instructions that share common dataflow predecessors into a single slice.

The problem instructions are in a loop. This is a common occurrence because one of our criteria for selection problem instructions is execution frequency. Because the loop body is small — only 15 instructions — the necessary latency tolerance cannot be achieved by executing a separate slice for each loop iteration. Instead the loop is encapsulated in the slice, and we attempt to fork the slice long before the loop is encountered in the main program.

Selecting a Fork Point. Fork points should be "hoisted" past unrelated code. The `add_to_heap` function is only called by the `node_to_heap` function (shown in Figure 3), so it is inlined by the compiler. Before calling `add_to_heap`, the `node_to_heap` function allocates a heap element and sets its fields. Because the slice only requires the `cost` field (passed in as a parameter to `node_to_heap)` and the `heap` itself (which is seldomly modified before the problem instructions are executed), the slice can be forked at the beginning of the `node_to_heap` function. This fork point is sufficiently before (60 dynamic instructions) the first instance of the problem branch to derive benefit from the slice in our simulated machine.

Selecting a fork point often requires carefully balancing two conflicting desires. Maximizing the likelihood that the problem instruction's latency is fully tolerated requires that the fork point be as early as possible, but increasing the distance between the fork point and the problem instructions can both increase the size of the slice and reduce the likelihood that the problem instructions will be executed. Often there is a set of "sweet spots" where the latency tolerance is maximized for a given slice size and accuracy.

Optimizations. Beyond removing the code that is unnecessary to execute the problem instructions, the size of our slice has been reduced by applying two optimizations. Although these particular transformations would be correct to apply to

Figure 3. ***The `node_to_heap` function, which serves as the fork point for the slice that covers `add_to_heap`.***

```
void node_to_heap (..., float cost, ...) {
   struct s_heap *hptr;  ◄──── fork point
   ...
   hptr = alloc_heap_data();
   hptr->cost = cost;
   ...
   add_to_heap (hptr);
}
```

the original code, it would be difficult for the compiler, which must preserve correctness across all possible inputs, to prove that this is so. Since the slice cannot affect correctness, it merely must discern that these transformations are correct most of the time, so that performance is not detrimentally affected.

- **Register Allocation:** The series of values loaded by `heap[ifrom]->cost` is always the value `cost`, which is passed in as an argument to `node_to_heap`. This fact could be detected by memory dependence profiling. Our slice takes the value `cost` as a *live in* value and removes all loads from `heap[ifrom]` and their corresponding stores to `heap[ito]`. This optimization removes 8 static instructions from the slice.
- **Strength Reduction:** The `ito = ifrom/2` statements in the original code are implemented by the compiler with a right shift operation to avoid a long latency division. To ensure the correct semantics of the division operator, a 3 instruction sequence is used that adds 1 to the value before the shift, if the value is negative. Value profiling could determine that the value added is always 0 (because `ifrom` is never negative). By using strength reduction to remove the addition of the constant zero, the division operation is reduced down to just the right shift, removing 4 static instructions from the slice.

Other profitable optimizations included removing register moves, eliminating unnecessary operand masking, exploiting invariant values, if-conversion, and reverse if-conversion. We have found removing communication through memory (like the register allocation example above) to be the most important.

Slice Termination. When a slice contains a loop, as our example does, we must determine how many iterations of the loop should be executed. The obvious means — replicating the loop exit code from the program — is often not the most efficient. Many loops have multiple exit conditions and inclusion of this computation in the slice can significantly impact slice overhead.

To avoid the possibility of a "runaway" slice when exit conditions are excluded, each slice is assigned a maximum iteration count. This number is derived from a profile-based estimate of the upper-bound of the number of iterations executed by the loop. When this maximum is close to the average iteration count for a loop, as is the case for our example, overhead can often be minimized by removing all loop exit computation from the slice and completely relying on the maximum iteration count to terminate the slice. Even when accurate back-edge branches are included in the slice, the latency of evaluating branch conditions often causes extra iterations of the loop to be fetched. Once the instructions have been fetched, the cost of executing them is typically small. Exceptions will also terminate slices; hence, linked list traversals will automatically terminate when they dereference a null pointer.

Slice Description. Figure 4 shows the original assembly code that corresponds to the source in Figure 2, and our slice that covers this region is shown in Figure 5. The un-optimized slice is shaded in Figure 4 and the problem instructions are in bold in both the original and slice code.

The optimized slice consists of only 8 static instructions. Typically a slice has fewer instructions than 4 times the number of problem instructions it covers. This small static

Figure 4. ***Alpha assembly for the `add_to_heap` function.*** *The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*

```
node_to_heap:
    ... /* skips ~40 instructions */
2   lda     s1, 252(gp)    # &heap_tail
2   ldl     t2, 0(s1)      # ifrom = heap_tail
1   ldq     t5, -76(s1)    # &heap[0]
3   cmplt   t2, 0, t4      # see note
4   addl    t2, 0x1, t6    # heap_tail ++
1   s8addq  t2, t5, t3     # &heap[heap_tail]
4   stl     t6, 0(s1)      # store heap_tail
1   stq     s0, 0(t3)      # heap[heap_tail]
3   addl    t2, t4, t4     # see note
3   sra     t4, 0x1, t4    # ito = ifrom/2
5   ble     t4, return     # (ito < 1)
loop:
6   s8addq  t2, t5, a0     # &heap[ifrom]
6   s8addq  t4, t5, t7     # &heap[ito]
11  cmplt   t4, 0, t9      # see note
10  move    t4, t2         # ifrom = ito
6   ldq     a2, 0(a0)      # heap[ifrom]
6   ldq     a4, 0(t7)      # heap[ito]
11  addl    t4, t9, t9     # see note
11  sra     t9, 0x1, t4    # ito = ifrom/2
6   lds     $f0, 4(a2)     # heap[ifrom]->cost
6   lds     $f1, 4(a4)     # heap[ito]->cost
6   cmptlt  $f0,$f1,$f0    # (heap[ifrom]->cost
6   fbeq    $f0, return    #  < heap[ito]->cost)
8   stq     a2, 0(t7)      # heap[ito]
9   stq     a4, 0(a0)      # heap[ifrom]
5   bgt     t4, loop       # (ito >= 1)
return:
    ... /* register restore code & return */
```

note*: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.*

Figure 5. ***Slice constructed for example problem instructions.*** *Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.*

```
slice:
1    ldq    $6, 328(gp)    # &heap
2    ldl    $3, 252(gp)    # ito = heap_tail
slice_loop:
3,11 sra    $3, 0x1, $3    # ito /= 2
6    s8addq $3, $6, $16    # &heap[ito]
6    ldq    $18, 0($16)    # heap[ito]
6    lds    $f1, 4($18)    # heap[ito]->cost
6    cmptle $f1,$f17,$f31  # (heap[ito]->cost
                           #  < cost) PRED
     br     slice_loop

## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

size translates to a small dynamic size relative to the program. This slice performs a prefetch or generates a prediction about every 3 instructions executed, and this is representative of the slices we have built. The slice only has 2 live-in registers, `cost` and the global pointer (used to access `heap`). In general, the natural slice fork points tend to require a small number of live in values; rarely are more than 4 values required. Table 3 characterizes a representative sample of the slices we have constructed.

3.3 Automatic Slice Construction

For speculative slice pre-execution to be viable, an automated means for constructing slices will be necessary. Although a few of our slices benefit from high-level program structure information that may be difficult to extract automatically, most of the slices and optimizations only use profile information that is easy to collect. The most difficult issues are those that involve estimating a slice's potential benefit, which is needed to maximize that benefit as well as to decide whether a slice will be profitable. Roth and Sohi [13] automatically selected un-optimized slices from an execution trace using the approximate benefit metric *fetch-constrained dataflow-height*. Automated slice optimization is important future work.

4 Slice Execution Hardware

To derive all benefits from the slices the hardware must provide the following:

- a set of resources with which to execute the slice.
- a means to determine when to fork the slice.
- a means for communicating initial state.
- a means for marking prediction generating instructions.
- a means for correlating predictions to branches in the main thread.

In this section, we describe how our example hardware implementation handles the first four of these issues. Due to its relative complexity, the prediction correlation hardware is described in its own section (Section 5).

4.1 Execution Resources

Rather than inserting the slices into the program inline, we execute them as separate, "helper" threads [3, 13, 14] on a simultaneous multithreading (SMT) [16] processor. In this way we avoid introducing stalls and squashes into the main thread due to the cache missing loads (*i.e.,* not just prefetches) and control flow in the slices.

Using idle threads in an SMT processor avoids adding execution hardware specifically for slices. The helper threads compete with the main thread for execution resources (*e.g.*, fetch/decode bandwidth, ALUs, cache ports). Because the helper threads enable the main thread to use resources more efficiently (by avoiding mispredictions and stalls), *the application with the slices can be executed with fewer resources than the application alone can.* Fetch resources are allocated to threads using an ICOUNT-like policy [16] that is biased toward the main thread.

Table 3. ***Characterization of Slices.*** *For a representative set of slices constructed we provide the size in static instructions (**static size**), the number of **live-in** register values, the number of problem loads prefetched (**pref**), problem branches predicted (**pred**), and the number of **kills** used for branch correlation (see Section 5). If the slice contains a loop, the number of each category within the loop is shown in parentheses, and the iteration limit is shown in the final column.*

Prog.	static size	live ins	pref	pred	kills	max iter
`bzip2`	8 (7)	4	1 (1)	2 (2)	1 (1)	2000
`crafty`	7	2	0	1	1	—
`eon`	8	1	0	6	3	—
`gap`	8 (5)	2	0	3 (3)	1 (1)	85
`gcc`	4 (3)	1	0	2 (2)	1 (1)	31
`mcf`	12 (12)	1	4 (4)	1 (1)	2 (1)	98
`twolf`	8 (5)	2	0	2 (2)	1 (1)	7
`vpr`	31 (15)	3	5 (3)	5 (2)	3 (1)	18
`vortex`	4	1	1	0	0	—

The helper threads and the main thread share the first-level data cache, so when data is prefetched by the slice it is available to the main thread. With a sufficiently large cache, the timing of the prefetches is not critical.

4.2 Slice Forking

Slices are forked when a given point — the *fork point* — is reached by the main thread. There are two ways of marking a fork point: inserting explicit fork instructions or designating an existing instruction as a fork point and detecting when that instruction is fetched. In order to maintain binary compatibility, we study the second approach in this work, but the hardware can be simplified by the former approach.

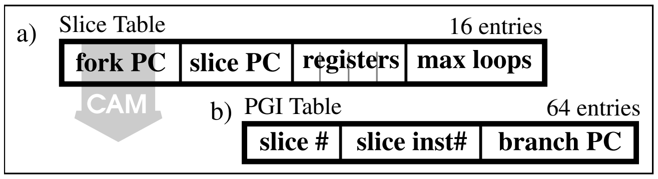
Our simulated hardware includes a small hardware structure called the *slice table (*shown in Figure 6(a)), which is maintained in the front end of the machine[1]. The first field, the *fork PC*, is a content-addressable memory (CAM) that is compared to the range of program counters (PC) fetched each cycle. If a match occurs, an idle thread is allocated to execute the slice. If no threads are idle, the fork request is ignored. The slice table provides the starting PC for the thread (*slice PC).* The instructions that comprise the slice are stored as normal instructions in the instruction cache.

4.3 Register Communication

In addition to the slice code, the helper thread needs a few "root" data values (as discussed in Section 3.2). To perform this inter-thread communication efficiently, we exploit SMT's centralized physical register file, by copying register map table entries and synchronizing the communication using the existing out-of-order mechanism [13, 17]. The small number of copies, logically performed when the fork

1. Slice table entries cannot be demand loaded; without a loaded entry the processor will not know that a fork point has been reached. One possible implementation would be to load the entries associated with a page of code when an instruction TLB miss occurs.

Figure 6. ***The slice and prediction generating instruction (PGI) tables, which reside at the front-end of the pipeline.*** *One entry in the slice table (a) is stored for each slice. The PGI table (b) has one entry for each prediction generating instruction in a slice. The fields are described in the text. These structures together require less than 512B of storage.*



point is renamed, are completed in the background by stealing rename ports from instructions that do not rename a full complement of registers. Because the main thread may overwrite a copied register before the slice is done with it, we slightly modify the register reclamation process to ensure that physical registers are not prematurely freed. The processor keeps a reference count for those registers.

No inter-thread synchronization is performed for memory, as it would require significant alteration of the load-store queue. Instead, slices are selected such that values loaded by the slice have not been recently stored by the main thread.

4.4 Prediction Generating Instructions

To improve branch prediction accuracy, a speculative slice computes branch outcomes to be used as predictions for problem branches in the main thread. The instruction in the slice that computes such an outcome is referred to as a *prediction generating instruction* (PGI). PGIs are identified by entries in the *PGI table* (Figure 6(b)), which are loaded along with slice table entries. In addition, each entry identifies the problem branch in the main thread to which it corresponds. When a PGI is fetched, it is marked with an identifier that indicates that the value it computes should be routed back to the front end of the machine and provided to the prediction correlator (described in the next section).

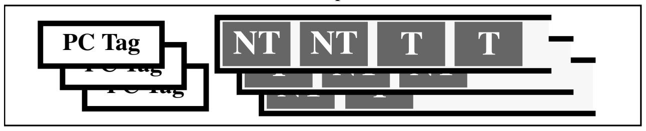
5 Branch Prediction Correlation

To derive any benefit from a branch prediction computed by a speculative slice, the prediction must be assigned to the intended dynamic instance of the problem branch. Since problem branches are unbiased, failure to accurately correlate predictions with branches will severely impact prediction accuracy. To maximize the benefit of the prediction, correlation is performed at the fetch stage. Conceptually, the branch correlator (Figure 7) consists of multiple queues of predictions, each tagged with the PC of a branch. When a branch is fetched and it matches with a non-empty queue, the processor uses the prediction at the head of the queue in place of the one generated by the traditional branch predictor.

Although conceptually simple, there are three issues that complicate the implementation of a prediction correlator:

- **Conditionally-executed branches:** If predictions are generated for all potential dynamic instances of a branch,

Figure 7. ***A conceptual view of a control-based prediction correlation mechanism.*** *When a branch is fetched by the main thread, its PC is compared to the correlator's PC tags. If a match occurs, the prediction at the head of the queue overrides the traditional branch predictor.*



and not all of them are executed due to control flow, a mechanism is required to kill the predictions that will not be used.
- **Mis-speculation recovery:** Because the correlator is manipulated when instructions are fetched by the main thread, it must be able to undo any action performed while the main thread was on an incorrect path.
- **Late Predictions:** A prediction can be generated after the branch for which it is intended was fetched. If this is not detected, the late prediction can potentially be incorrectly correlated with future dynamic instances of the branch.

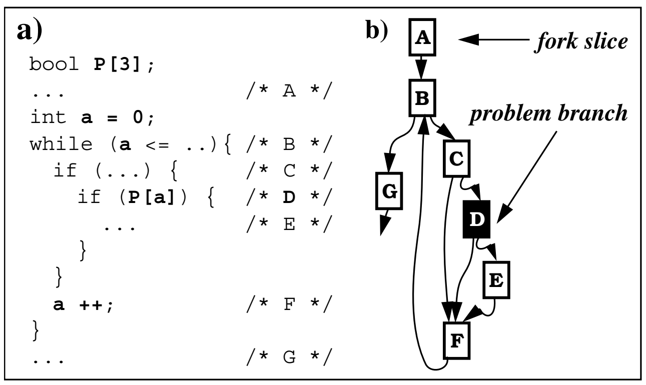
The following sub-sections describe each of these issues in turn, providing more details on the problems and how our mechanism solves them.

5.1 Conditionally-executed Branches

Achieving sufficient latency tolerance often necessitates "hoisting" the fork of a slice above branches that determine if a problem branch will be executed. When the fork point and the problem branch are no longer control-equivalent (*i.e.*, a problem branch is not executed exactly once for each fork point executed), we may generate more predictions than there are problem branches to consume them. If any unused predictions are left in the queue, the predictions will become mis-aligned, severely impacting prediction accuracy.

Figure 8 shows an example of a conditionally executed problem branch. This problem branch (in block **D**) exists inside of a loop and is only executed on the iterations in which the `if` statement in block **C** evaluates to `true`. Assuming that this loop is rarely executed more than three

Figure 8. ***Example of a conditionally executed problem branch within a loop.*** *Code example (a) and control flow graph (b).*



a)

```
bool P[3];
...                 /* A */
int a = 0;
while (a <= ..){ /* B */
  if (...) {        /* C */
    if (P[a]) {     /* D */
      ...           /* E */
    }
  }
  a ++;             /* F */
}
...                 /* G */
```

times before exiting, we create a slice that generates a prediction for each of the first three iterations. Assuming, without loss of generality, that the slice is executed in a timely fashion, the queue begins with one prediction for each of the first three loop iterations. If the problem branch in the first iteration is not executed, the problem branch in the second iteration will be matched with the prediction generated for the first iteration. Furthermore, if the loop exits without consuming all of the generated predictions, these predictions could potentially cause future mis-correlations to occur.

This problem could be avoided by including the control flow that determines whether the problem branch will be executed (*e.g.*, the conditions computed by blocks **B** and **C** in the example in Figure 8) in the slice. We have found that including this additional code, the *existence sub-slice* [19], in the slice substantially increases the overhead of the helper thread and potentially impacts the latency of generating the predictions. Given that these control conditions are already going to be evaluated by the main thread, replication of this code is unnecessary. Instead, our mechanism observes the control flow as it is resolved by the main thread.

Instead of "consuming" predictions when they are used by a problem branch, we use the main thread's execution path to deallocate predictions when it can be determined that the prediction is no longer valid. For each prediction, we identify the *valid region* — the region in which the intended branch instance can still be reached — in the control flow graph (CFG), and when the execution leaves this region the prediction is killed. Figure 9(a) shows an unrolled CFG for the loop in our example and the valid regions for the first two predictions. Note that there are two types of exits from a region: 1) the arcs $\overline{CF}$, $\overline{DE}$, and $\overline{DF}$ kill the prediction for a single iteration, and 2) the arc $\overline{BG}$ (the loop exit) kills all predictions for the loop. We refer to the first kind as a *loop iteration kill* and the second type as a *slice kill*.

Although there are potentially a large number of exits from a valid region, we have found it unnecessary to monitor all such edges. In practice, a prediction does not need to be killed exactly when its valid region is exited; it merely must be killed before the next instance of the problem branch is fetched. Because of the reconvergent nature of control flow in real programs, all paths tend to be channeled back together at distinct points (*e.g.*, all paths through a function eventually return from the function). For this reason, we merely need to select a reconvergent point that occurs between the two instances of the problem branch. More precisely, we select a point (or set of points) in the program that post-dominates all of the valid region exits and dominates the next execution of the problem instruction. In practice, we have found selecting such points to be straightforward. In our example, block **F** can serve as a loop iteration kill and block **G** as a slice kill. Figure 9(b) shows the operation of our mechanism for this example loop.

In this work, we identify existing instructions in the main thread for use as kills; the slice table provides these kill PCs to the prediction correlator when a slice is forked. Typically, a small number of kill instructions is sufficient to perform correlation correctly. We have found that often the best loop iteration kill block is the block that is the target of loop back-edges. In this case, the first instance of the block should not kill any predictions.

5.2 Mis-speculation Recovery

Modern processors employ speculation in many respects, often squashing and re-fetching instructions to recover from mis-speculation. In order to properly correlate predictions, it is necessary to undo any actions performed on the correlator by the squashed instructions (much like branch history must be restored). The obvious solution is to restore predictions if the killing instruction gets squashed. Thus predictions are

Figure 9. ***Killing predictions when they are no longer valid****. Figure (a) shows the regions in which the predictions for the first and second iterations are valid on an unrolled control flow graph. Block F (the loop back-edge) can be used to kill the prediction for each iteration, and block G (the loop exit) should be used to kill all predictions. Figure (b) shows how loop iteration kills (block F) and slice kills (block G) can be used to ensure correct prediction correlation. The path taken is ABCFBCDFBG.*

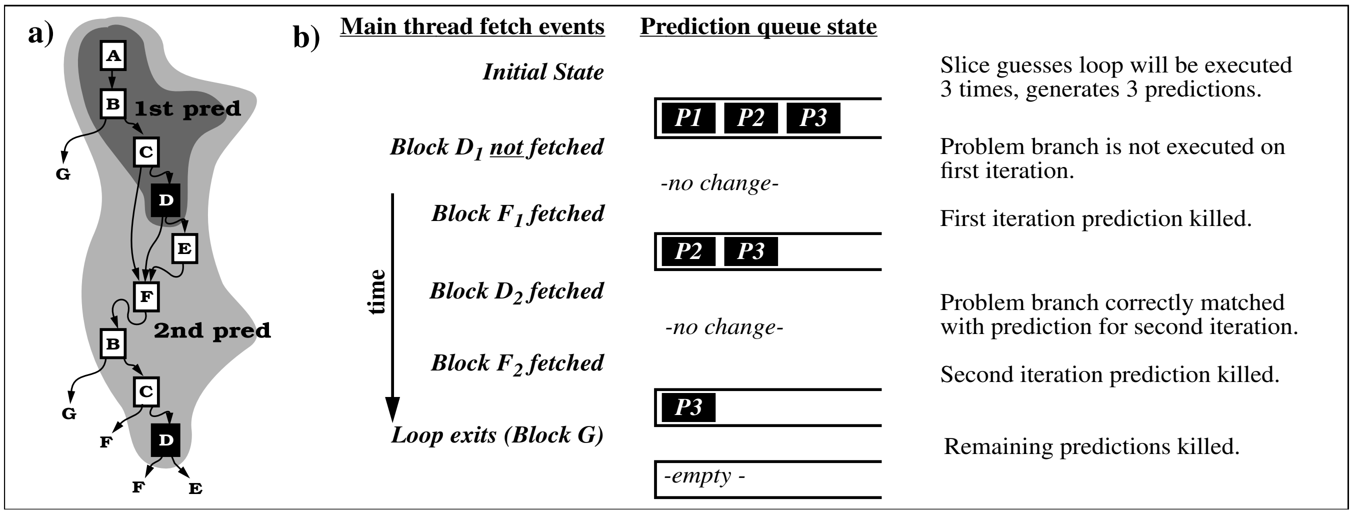
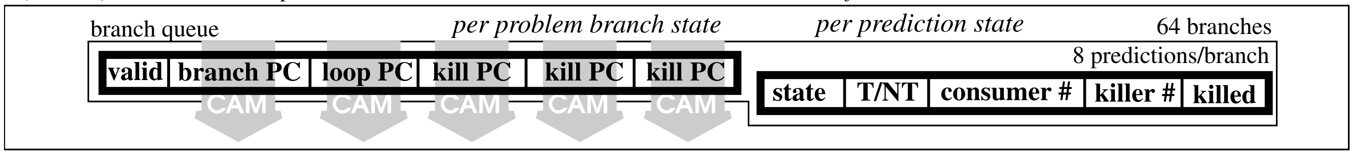

a)
A
B
1st pred
C
G
D
E
F
2nd pred
B
G
C
F
D
F
E
b)
Main thread fetch events
Prediction queue state
Initial State
P1 P2 P3
Slice guesses loop will be executed 3 times, generates 3 predictions.
Block D_1 not fetched
-no change-
Problem branch is not executed on first iteration.
time
Block F_1 fetched
P2 P3
First iteration prediction killed.
Block D_2 fetched
-no change-
Problem branch correctly matched with prediction for second iteration.
Block F_2 fetched
P3
Second iteration prediction killed.
Loop exits (Block G)
-empty -
Remaining predictions killed.

Figure 10. ***Prediction correlation hardware.*** *A single entry of the branch queue contains a valid bit (**valid**), the PC of the branch in the main thread (**branch PC**), the PCs of loop kill and slice kill instructions in the main program (**loop PC** and **kill PC**), and state for 8 predictions. Each prediction has a **state** (one of Invalid, Empty, Full, and Late), the direction of the prediction (**T/NT**), the Von Neumann number (VN#) of the consuming instruction (consumer #) if the prediction is late, the VN# of the instruction that killed this prediction (**killer #**), and whether the prediction has been **killed**. The table contains about 1KB of state.*



not deallocated until the kill instruction retires, but merely marked as killed and ignored for future correlations. We store the *Von Neumann number* (VN#, a sequence number used for ordering instructions) of the kill instruction with the prediction. If the kill VN# is in the range of squashed VN#s, we clear the kill bit, restoring the prediction.

5.3 Late Predictions

Late predictions can cause mis-correlation if the prediction arrives after the associated kill. Two factors affect the latency of generating a prediction: (1) the pipeline depth, because the prediction generating instruction (PGI) must reach the execute stage before the prediction is computed, and (2) the execution latency of the slice (recall that problem branches often depend on problem loads). Because not all slices are forked sufficiently early, our mechanism gracefully supports late predictions.

To avoid mis-correlation, we allocate a prediction in the queue when the PGI is fetched, rather than executed, because it is easy to ensure that the PGI is fetched before its kill. This entry is initialized to the *Empty* state and changed to the *Full* state when the PGI executes. If an empty entry is matched to a branch, the traditional predictor is used. Kills behave the same whether the entry is *Empty* or *Full*.

Late predictions can also be used for early "resolution," because a late prediction may still arrive significantly in advance of when the problem branch is resolved. If a prediction in the *Empty* state is matched to a problem branch, it transitions to the *Late* state and the VN# of the branch and the prediction it used are stored in the prediction entry. Later, when the PGI is executed, if the computed branch outcome does not match the stored prediction and the branch has yet to be resolved, the prediction is reversed and fetch is redirected. Because speculative slices are not necessarily correct, extra squashes can be introduced. These rare occurrences are corrected when the branch is resolved.

5.4 Prediction Correlation Hardware

Given prediction entries that support conditionally executed branches, mis-speculation recovery, and late predictions, it is straightforward to construct an accurate prediction correlator. Figure 10 shows one possible implementation of such a correlator. More efficient implementations, as well as those that support recursion, are possible, but they cannot be covered here due to space considerations. Also, much of the CAM circuitry can be avoided by putting explicit kill annotations into the program, as was described for fork instructions in Section 4.2.

6 Performance Results

Significant performance benefits can be achieved through speculative slice pre-execution. Because the slices in this work have been constructed manually, we have only generated slices for a portion of the problem instructions in each benchmark. We have profiled the runs of the SPEC2000 integer benchmarks and have selected a 100 million instruction region of the execution in the dominant phase of each simulation[2]. We have identified the problem instructions in this region and constructed slices to cover some of them. We warm-up the caches and branch predictors by running 100 million instructions. For these experiments we assume that only a single program is running, hence all remaining thread contexts are idle. If multiple applications are running, higher throughput will likely be achieved by using the thread contexts to run those applications instead of slices.

We present results for three experiments: 1) a base case with the microarchitecture described in Section 2.1, 2) the base case augmented with speculative slices (on a 4 thread SMT), and 3) a constrained limit study to understand how effectively the slices are achieving their goals. This constrained limit study is performed by "magically" avoiding the PDEs from the problem instructions covered by our slices. The speedups of the *slice* and *limit* cases, relative to the base case, are shown in Figure 11 for a 4-wide machine (the 8-wide results, omitted for space, are similar). Supplemental data for the benchmarks with non-trivial speedups is provided in Table 4; the others are discussed in Section 6.2.

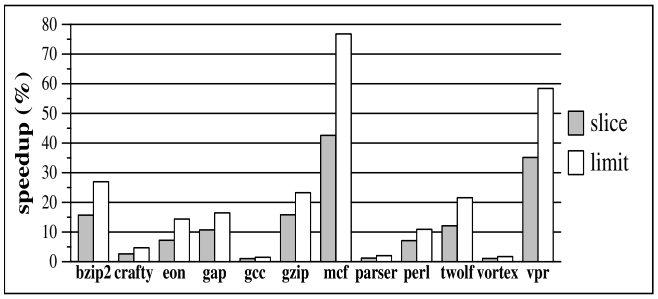
Substantial speedups are achieved (note that many of these programs are within 30-50% of peak throughput, as shown in Figure 1) by reducing the number of mispredictions and cache misses (up to 72% and 64%, respectively). For most benchmarks, the speedup is largely derived from avoiding mispredictions; only `gap`, `mcf`, and `vpr` get a majority of their benefit from prefetching. The speedups are on the order of half of the speedup of the limit case.

6.1 Analysis

Achieving the speedup attained by the limit study necessitates that the predictions and prefetches are both accurate and timely. Accuracy is seldom a problem; our slices and

2. Because the data shown here is only for a sample of the program's execution, it cannot be compared directly with data presented in Section 2.

Figure 11. ***Speedup achieved by the slice-assisted execution and a limit study that perfects the same set of problem instructions.*** *Data shown for a 4-wide machine.*



prediction correlation mechanism exceed a 99% prediction accuracy when they override the traditional predictor (4 of the 8 benchmarks in Table 4 have no induced mispredictions). On the other hand, forking a slice sufficiently early to tolerate the full latency of a misprediction or cache miss can be problematic (discussed in Section 3.2). Around a quarter of the slices constructed are consistently late, reducing their benefit. One such example is prefetching the tree for `mcf`; the work performed at each node is insufficient to cover the latency of the sequential memory accesses.

Furthermore, executing slices is not free; there is the opportunity cost of stealing resources — mostly instruction fetch opportunities — from the main thread that contributes to the performance discrepancy in Figure 11. This overhead depends not only on the number of dynamic instructions executed by a slice, but also on the cost of stealing an instruction opportunity from the main thread. If the main thread has insufficient ILP, is stalled on an untolerated cache miss, or is fetching down a false path, the cost of executing the slice can be small. On the other hand, if the main thread is efficiently utilizing the processors resources (indicated by a high base IPC), executing a slice can be expensive, making many potential slices unprofitable.

Overhead is aggravated because not every instance of a problem instruction will miss or be mispredicted. Although many problem loads consistently miss in the cache, branch predictors are good enough that even problem branches are predicted 70-85% correctly. Overhead is consumed whenever a slice is executed, but benefit is derived only when a PDE is avoided; this magnifies the overhead by the inverse of the miss/misprediction rate.

As can be seen in Table 4, the amount of overhead can be substantial. As much as 10-15% of the instructions fetched belong to slices, but in all cases the total number of instructions fetched is reduced. Experiments (data not shown) estimate that this execution overhead represents about half of the discrepancy between our results and the limit study. Some of the fetched slice instructions fail to “retire,” generally because the associated fork point was squashed. Fetch bandwidth wasted on squashed slices is not a large concern

Table 4. ***Characterization of program execution with and without speculative slices for benchmarks with non-trivial speedups.*** *Each benchmark was run for 100 million instructions.*

		bzip2	eon	gap	gzip	mcf	perl	twolf	vpr
Base	Program instructions fetched (M)	187.8	154.0	127.5	179.8	357.5	133.9	200.0	175.8
	Branch mispredictions (M)	1.0	0.3	0.3	1.1	2.6	0.1	1.2	0.7
	Load misses (M)	0.7	0.0	0.1	0.1	6.8	0.2	1.4	1.3
Base + Slices	Program instructions fetched (M)	148.0	133.9	112.8	127.9	273.4	121.9	164.6	130.6
	Slice instructions fetched (M)	11.5	6.5	4.0	15.3	40.8	2.0	6.6	16.3
	Slice instructions “retired” (M)	11.3	5.1	3.9	9.8	40.7	1.9	3.3	14.5
	# Fork Points (K)	306	455	135	928	8	138	594	893
	# Fork Points Squashed (K)	53	141	12	334	3	18	254	78
	# Fork Points Ignored (K)	-	-	-	-	-	-	0	15
	Problem Branches Covered	8	9	5	25	3	12	12	6
	Predictions generated (K)	3,303	1,300	2,365	2,867	2,540	687	3,610	2,131
	Mis-predictions “covered” (K)	393	261	221	687	413	161	471	547
	Mis-predictions removed (K)	370	257	215	680	388	120	380	510
	Total mis-predictions removed (%)	37%	52%	64%	64%	15%	35%	33%	72%
	Incorrect predictions (K)	1	-	-	5	-	-	0	3
	Late predictions (%)	15%	40%	1%	11%	4%	20%	1%	31%
	Problem Loads Covered	4	0	2	0	8	1	1	7
	“prefetches” performed (K)	1,876	-	222	-	20,253	379	272	4,689
	Cache misses “covered” (K)	547	-	37	-	4,940	58	346	1,299
	Net reduction in misses (K)	396	-	37	-	3,798	51	172	1,261
	Net reduction in total misses (%)	46%	-	60%	-	55%	30%	12%	64%
	Fraction of speedup from loads	~10%	-	~50%	-	~80%	~20%	~10%	~50%

because it only reduces the number of wrong path instructions fetched by the main thread. Only two programs, `twolf` and `vpr`, ignore fork requests on a machine with 3 idle helper threads, but most programs benefit from having more than one idle thread. Often there is one long-running, "background" slice and a number of periodic, localized slices.

6.2 Slice Construction Failures

For the selected phase of execution, we were unable to achieve significant speedups on 3 of the benchmarks considered: `gcc`, `parser`, and `vortex`[3]. These benchmarks provide examples of difficulties in slice construction.

Many problem branches in `gcc` are in functions that process `rtx` structures. Typically these functions have a switch statement based on the node type, and they recursively descend the tree-like `rtx` on a subset of the cases. This switch statement is frequently a problem branch. The unpredictability of the traversal, coupled with fact that computing the traversal order is a substantial fraction of these functions, makes generating profitable slices difficult.

`Parser` has two main problem localities: a hash table and memory deallocation. Key generation for the hash table is computationally intensive, over 50 instructions, and it occurs right before the problem instructions. These instructions would be replicated, causing substantial slice overhead. The memory allocator is organized as a stack, and much of the work of deallocation is deferred until the deallocated chunk becomes the top of the stack. Thus, when the top of stack element is deallocated, a long cascade of deallocations is performed. To prefetch the sequential accesses, the fork point must be hoisted up significantly, but the unpredictability of which call to `xfree` will cause the cascade obstructs hoisting without causing many useless slices.

The problems with `vortex` are more mundane. Its base IPC is high, within 13% of peak throughput for our 4-wide machine, which makes the opportunity cost of slice execution high. This is aggravated by low miss/misprediction rates by some problem instructions.

In some cases, notably `parser`, minor source-level restructuring could likely enable these slices to be profitable.

6.3 Summary

Execution-based prediction using speculative slices appears to be very promising. By adding only a small amount of hardware (a few kilobytes of storage), we were able to significantly improve performance on an already aggressive machine for many of the benchmarks studied. In considering other possible implementations, we can qualitatively reason about some aspects of performance:

- The speculative optimizations applied to slices have a two-fold benefit: overhead is reduced by reducing slice size, and timeliness is improved by reducing the critical path through the slice.
- Programs and processors with low base IPCs (relative to peak IPC) are more likely to benefit from slices because the opportunity cost of slice execution is lower.
- Overhead can be reduced by not executing slices for problem instructions that will not miss/mispredict. Some of our slices do this statically; if the problem instructions behave differently in different calling contexts and the fork point is hoisted into the calling procedure, only the profitable contexts fork slices. Obvious future work is gating the fork using confidence [8].
- Execution overhead could be eliminated by having dedicated resources to execute the slice at the expense of additional hardware. In addition, existing slices may be improved and additional slices may be profitable without resource constraints.

7 Related Work

Besides our previously mentioned research to understand backward slices [19], the closest related work is *speculative data-driven multithreading* (DDMT) [13]. In that work, Roth and Sohi describe a multithreaded processor that can fork *critical computations* (much like our slices) to tolerate memory latency and resolve branches early. The differences between the DDMT work and this research are largely derived from the former being implemented around a technique called *register integration* [12]. Integration allows the results of speculatively executed instructions to be incorporated into the main thread when a data-flow comparison determines that they exactly match instructions renamed by the main thread. In this way, mispredicted branches that were pre-executed are resolved at the rename stage (as opposed to at fetch as with our control-based scheme) and the main thread avoids the latency of re-executing instructions that have been pre-executed. This data-flow comparison requires one-to-one correspondence in the computations, precluding optimization of slices. Furthermore, in this work we consider pre-executing computations that contain control flow (loops and if-converted general purpose control flow).

Three earlier works provide more restricted implementations of pre-execution, including support for linked data structures [10], virtual function call target computation [11], and conditional branches in "local loops" [7]. Farcy, *et al.* describe a simple prediction correlator that supports mis-speculation and late predictions, but not conditionally executed branches [7]. The prediction correlation technique proposed by Roth, *et al.* [11] is, in some sense, the complement of the one described here; it uses the path through the program to attempt to determine when a prediction should be used, while we use the path to invalidate predictions.

Sundaramoorthy, *et al.* propose a different approach to producing a reduced version of the program with the potential to execute instructions early [15]. Their *slipstream* archi-

3. We also did not significantly improve `crafty`'s performance. Many of `crafty`'s problem instructions occur in two functions, `FirstOne` and `LastOne`, which find first and last set bits in 64-bit integers, respectively. Because the Alpha architecture has instructions for these functions, we did not bother optimizing them.

tecture executes a program on two cores of a chip multiprocessor (CMP), speculatively removing "ineffectual" computations from the first execution (the A-stream) and verifying the speculation with the second execution (the R-stream). Prediction correlation between the executions is performed by the A-stream supplying a complete "trace" of predictions to the R-stream. If the R-stream detects an incorrect prediction in the trace, the trace and the A-stream execution are squashed.

The uneven distribution of performance degrading events has previously been observed. Abraham, *et al.* [1] quantified the concentration of cache misses by static instruction in SPEC89 benchmarks. Mowry and Luk [9] found that path and context could be used to predict which dynamic instances of instructions were likely to cache miss. Jacobsen, *et al.* [8] characterized confidence mechanisms to identify the static branches, as well as the particular dynamic instances of those branches, likely to be mispredicted.

SSMT [3] and Assisted Execution [14] propose the concept of helper threads (*i.e.*, using idle threads in a multithreaded machine to improve single thread performance by executing auxiliary code), and explore helper thread implementations of local branch prediction and stride prefetching, respectively.

8 Conclusion

We have shown that a small set of static instructions, which we call problem instructions, are responsible for a significant amount of lost performance. These instructions are not easily anticipated with existing caching, prefetching, and branch prediction mechanisms because their behavior does not exhibit any easily exploited regularity.

Because the program correctly computes the outcome of all instructions, we can predict the behavior of problem instructions by building approximations of the program called speculative slices. These speculative slices are executed, in advance of when the problem instructions are encountered by the main thread, to perform memory prefetching and generate branch predictions. The effects of the slices are completely microarchitectural in nature, in no way affecting the architectural state (and hence correctness) of the program. We have found that efficient and accurate speculative slices can be constructed for many problem instructions. With moderate hardware extensions to a multithreaded machine, these slices can provide a significant speedup (up to 43 percent).

In order to benefit from branch predictions generated by speculative slices, we must correlate these predictions with problem branches as they are fetched by the program. The final contribution of this paper is a prediction correlation mechanism that uses the path taken by the program to kill predictions when they can no longer be used. This technique is accurate and can potentially be used to correlate other types of predictions (*e.g.*, value predictions).

9 Acknowledgements

The authors would like to thank Ras Bodik, Adam Butts, Charles Fischer, Milo Martin, Manoj Plakal, Dan Sorin, and Amir Roth for commenting on drafts of this paper. This work was supported in part by National Science Foundation grants CCR-9900584 and EIA-0071924, donations from Intel and Sun Microsystems, and the University of Wisconsin Graduate School. Craig Zilles was supported by a Wisconsin Distinguished Graduate Fellowship.

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

[3] 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.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] 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.

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

[7] 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.

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

[9] T. Mowry and C.-K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *Proc. 30th International Symposium on Microarchitecture*, pages 314–320, Dec. 1997.

[10] 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.

[11] 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.

[12] A. Roth and G. Sohi. Register Integration: A Simple and Efficient Implementation of Squash Reuse. In *Proc. 33nd International Symposium on Microarchitecture*, Dec. 2000.

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

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

[15] 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.

[16] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[17] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proc. 25th International Symposium on Computer Architecture*, pages 238–249, Jun. 1998.

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

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