Heuristics for Profile-driven Method-level Speculative Parallelization

John Whaley and Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{jwhaley,kozyraki}@stanford.edu

Abstract

Thread level speculation (TLS) is an effective technique for extracting parallelism from sequential code. Method calls provide good templates for the boundaries of speculative threads as they often describe independent tasks. However, selecting the most profitable methods to speculate on is difficult as it involves complicated trade-offs between speculation violations, thread overheads, and resource utilization. This paper presents a first analysis of heuristics for automatic selection of speculative threads across method boundaries using a dynamic or profile-driven compiler. We study the potential of three classes of heuristics that involve increasing amounts of profiling information and runtime complexity. Several of the heuristics allow for speculation to start at internal method points, nested speculation, and speculative thread preemption.

Using a set of Java benchmarks, we demonstrate that careful thread selection at method boundaries leads to speedups of 1.4 to 1.8 on practical TLS hardware. Single-pass heuristics that filter out less profitable methods using simple speedup estimates lead to the best average performance by consistently providing a good balance between over- and under-speculation. On the other hand, multi-pass heuristics that perform additional filtering by taking into account interactions between nested method calls often lead to significant under-speculation and perform poorly.

1 Introduction

With uniprocessors running into fundamental ILP and VLSI limitations [1], single-chip multiprocessors (CMPs) provide a realistic path towards scalable performance using the ever-growing transistor budgets [13, 15, 11]. The problem now becomes how to best utilize the parallel resources on the chip. Manually writing explicitly programs is a tedious and error-prone task for the average programmer.

Thread-level speculation (TLS) is an effective approach for extracting suitable levels of parallelism from sequential code for CMPs [18, 16]. With TLS, the processor executes computational tasks in parallel, optimistically assuming that sequential semantics will still be preserved. Special hardware detects if the assumption is violated and initiates re-execution of any offending tasks. TLS is particularly attractive because it relieves the programmer and the compiler of proving independence between threads and placing locks to orchestrate accesses to shared data.

Parallel execution with TLS requires that speculative threads are identified in the code. One option is to have the programmer manually select speculative threads [17]. Alternatively, a dynamic or profile-driven compiler can automatically select the optimal set of speculative threads in a sequential program given runtime information collected from previous runs. In this case, the programmer must only write an ordinary sequential program and all legacy source code can reap the benefits of TLS support in CMP systems.

This paper presents a first analysis of practical heuristics that allow a profile-driven compiler to automatically select speculative threads in a sequential program. We focus on method calls that describe independent tasks within a program. Limit studies have shown that there are significant amounts of method-level parallelism in common applications [16, 21]. Several researchers have exploited method-level parallelism with TLS hardware. Yet, to our knowledge, this work is the first to describe, tune, and evaluate a comprehensive set of heuristics that compiler writers can employ in order to extract method-level speculative parallelism. The heuristics presented here are complementary and fairly orthogonal to the well-studied compiler techniques for speculative parallelization of loops [23, 7].

The main contributions of this work are:

- We describe three classes of heuristics that use increasing amounts of profiling information and involve increasing runtime complexity. Simple heuristics rely on straight-forward measurements such as the method runtime. Single-pass heuristics filter methods that are unlikely to be profitable using estimates of speedup and runtime savings. Multi-pass heuristics remove additional methods by considering the interactions between nested method calls. Several heuristics allow for speculation to start at intermediate method points, nested speculation, and speculative thread preemption.
- We evaluate the performance potential of these heuristics with Java applications running on a simulated CMP with TLS support. The heuristics lead to speedups of 1.4 to 1.8 over sequential execution on a single processor, which is 80% to 95% of the performance possible when using an ideal oracle. We also show that the worst case buffering requirements by the selected speculative threads are less than 16 Kbytes,

which is reasonable for both current and future CMP designs.

- We demonstrate that single pass heuristics achieve the best average performance as they provide a reasonable balance between under- and over-speculation. Multi-pass heuristics often lead to under-speculation and mediocre performance gains due to the complexity of the interactions between nested method calls.

The rest of the paper is structured as follows. In section 2 we give an overview of method-level speculation and the challenges it involves. Section 3 presents the investigated heuristics. We explain the experimental methodology in section 4. Section 5 presents the evaluation results. Section 6 discusses related work and Section 7 concludes.

2 Method-Level Speculation Overview

TLS targets parallelism in sequential programs by dividing the code into threads to be speculatively executed in parallel. Since the program is not explicitly written for parallel execution, the TLS system must preserve the original sequential semantics. The hardware tracks all memory addresses that threads write or read and detects when a less speculative thread writes to an address that a more speculative thread has already read. This is a violation of sequential semantics and causes the younger thread and all other more speculative threads to be squashed and restarted [8]. In this study, we always restart threads at the beginning of their execution. A more aggressive model would be to restart them at the first instruction causing the violation [22], but this requires significant hardware complexity.

A complete TLS compiler should select speculative threads from both loops and method calls in the sequential code in a complementary manner. Loop-level thread selection is a well-studied topic [23, 5, 7]. In this paper, we focus on thread selection at method boundaries. There are two reasons why method calls are well-suited for speculation. First, methods are used by programmers to divide the code into separate units of computation. Although it is not always the case, separate methods frequently perform independent tasks. Second, method calls communicate primarily through memory with only a small and well-defined number of parameters or return values going through registers. Hence, there is little need for analysis of the liveness of register-allocated variables when creating speculative threads [23].

2.1 Speculation Model

Our model for method-level speculation is to optimistically execute the code following a method return while the called method is still executing as shown in Figure 1. Speculation leads to speedups when no violations occur (Figure 1.c), but it may also cause slowdowns in the case of violations and re-executions (Figure 1.d). At method return the only state communicated through registers is the return value. We use value prediction to allow speculation on non-void methods, which is especially beneficial for methods that return an error code, as they typically return a constant on successful program runs.

We do not always initiate speculation at the beginning of a method call. Instead, the speculative thread may be forked at any point within the method body. Hence, speculative threads that would otherwise violate if they were forked at the beginning of the method can start later in the method, so that the read in the speculative thread will happen after the write in the non-speculative thread. This technique requires no additional hardware or software support beyond normal method speculation as the values to start the speculative thread are easily available in the caller's and/or callee's stack frame. We use the term *fork point* for the location in the method where speculation starts. The *join point* is right before the method return in order to accommodate methods with multiple callers. When the join point is reached, the non-speculative thread finishes and the speculative thread executing beyond the method return becomes non-speculative (or less speculative).

Finally, our model allows for nested speculation. A speculative thread can fork more speculative threads when it reaches a fork point. Any additional threads are strictly nested within the thread that forked them. This happens automatically due to the nested nature of method calls and returns in imperative programming languages.

2.2 Challenges of Method-Level Speculation

TLS leads to maximum benefits if the selected speculative threads are very likely to be independent and each includes a non-trivial amount of work. Such a selection assigns processors with useful threads that will not lead to violations. It also minimizes the impact of the overheads associated with forking, joining, and restarting threads.

Thread selection at method-level boundaries involves several challenges. Unlike with loop-based speculation, speculative threads from method calls are dissimilar, which makes it difficult to evaluate the likelihood of violations. Furthermore, threads may have arbitrary lengths with some of them being particularly long and some of them being particularly short. A long running thread is more likely to violate, which is particularly wasteful if it happens towards the end of the thread execution. On the other hand, a short speculative thread may not have sufficient work to amortize the cost associated with starting and stopping the thread in a modern system.

Long threads can be problematic even without violations. TLS hardware must retire speculative threads in sequential order. It is possible that a long running thread can delay several shorter, more speculative threads that have completed their execution but cannot retire. Consequently, precious hardware resources may be unavailable for new threads. This problem can be lessened by hardware support for multiple speculative buffers per CPU, but this can be fairly expensive to implement. A long running speculative thread may also overflow the available buffer space for speculative writes. In this case, the TLS hardware will have to stall this thread until it becomes non-speculative. In other words, long threads may lead to unnecessary serialization.

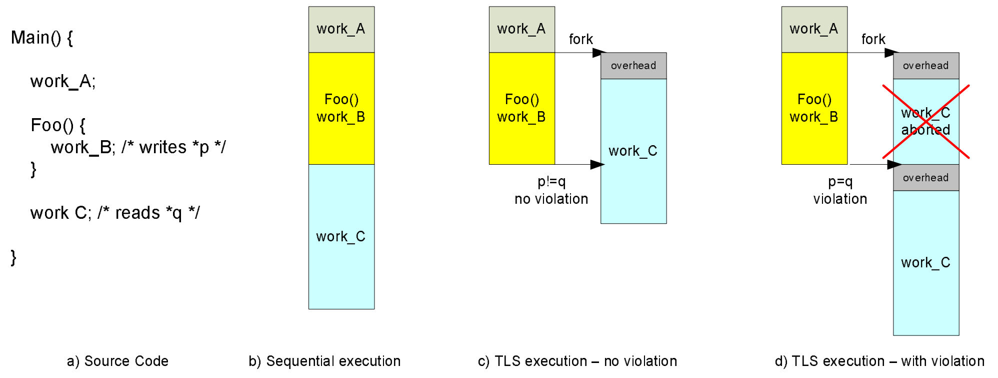

Figure 1. Example of method-level TLS. In (b), Foo() and the subsequent code (work_C) execute sequentially. In (c), the code executes in parallel without violations because the two pointers p and q are not aliased. In (d), speculation results into a dependency violation and re-execution because the p and q are aliased.

3 Heuristics

A dynamic or profile-driven compiler can select profitable methods for speculation using runtime information about the application behavior. Profiling data is the key to overcoming the challenges of method-level speculation in ways that a static compiler is unlikely to match. An ideal selection is accurate with high probability and fast to make. Both features depend on the amount of runtime data the compiler employs and the number of passes it must make over the data before reaching a decision.

We investigate a set of heuristics that allow a profile-driven compiler to select profitable fork points for method-level speculation. All heuristics attempt to select speculative threads with low probabilities of violations, which is the key performance issue. The heuristics do not attempt to minimize buffer space requirements for the speculative threads. Nevertheless, our experimental methodology tracks the maximum buffer requirements for each heuristic, which provides an understanding of the tradeoffs between buffer size and performance.

We categorized our heuristics into three classes based the amount of runtime information they use and the complexity involved in their decisions. *Simple heuristics* are the most straightforward. They determine fork points with a simple analysis of information such as sequential runtime or dynamic count of stores for each method. The simplistic analysis can often lead to over-speculation. *Single-pass heuristics* attempt to eliminate some unprofitable selections by estimating the potential speedup or runtime savings by speculating at each method call. They involve a single-pass over the addresses accessed by the method and the code beyond its return point. *Multi-pass heuristics* are the most complex. They attempt to filter even more inappropriate selections by adjusting the speedup expectations when nested speculation is used. These heuristics require multiple passes over the profile data. The number of passes is bounded by the depth of the method call graph. Single-pass and multi-pass heuristics allow for speculation to start at points other than the very beginning of a method.

This work focuses on describing the idea and evaluating the potential of the heuristics through simulation. Their exact implementation in a specific compiler is beyond the scope of this paper. Nevertheless, the runtime information necessary can be collected during a profiling stage using hardware with address analysis capabilities [24, 3]. Then, the heuristics can be implemented dynamically by a service thread [10] or off-line using traditional profile-driven compilation techniques.

3.1 Simple Heuristics

Runtime Heuristic (SI-RT): This heuristic speculates on all methods with an average runtime between a minimum (MIN) and a maximum (MAX) threshold. The idea is to find methods that do enough work to amortize the speculation overhead, but are not too likely to cause violations and load imbalance. The only information needed is the average runtime of each method under consideration, which is trivial to collect using hardware counters. The drawback is that method runtime is only loosely correlated to the likelihood of violations between speculative threads.

Store Heuristic (SI-SC): This heuristic speculates on methods with a total dynamic count of stores less than a maximum (MAX) threshold. The rationale is that methods with low store counts are unlikely to cause many violations. The dynamic store count can also be monitored using hardware counters. This heuristic also suffers from the problem that dynamic store count is only somewhat correlated to violations, since it does not capture the overlap between the write sets of different threads. Additionally, by selecting methods with low numbers of stores, there is a hazard of choosing methods that are too short.

3.2 Single-Pass Heuristics

Best Speedup Heuristic (SP-SU): This heuristic speculates on methods that are predicted to lead to relative speedup above a determined threshold (THRES). We define speedup as the ratio of the sequential execution time for the method and its subsequent code over the time necessary to execute the two in parallel. For the subsequent code, we assume a portion with average runtime equal to that of the method itself. Like all single-pass heuristics, this heuristic allows the fork point to be placed at the first point within the method that eliminates violations in the subsequent code. SP-SU requires a single pass over the address stream for the method and its subsequent code in order to place the fork point. The addresses for the method are examined in reverse order. It also requires an estimate of average runtimes for the relative speedup calculation. Its major shortcoming is that it is possible for very short methods to score very highly in terms of speedup. The corresponding threads may prevent longer methods with smaller relative speedups but larger contributions to the overall runtime from running.

Most Cycles Saved Heuristic (SP-CS): This heuristic targets a shortcoming of SP-SU by speculating on methods which are predicted to save the largest number of cycles when executed speculatively. Instead of dividing the sequential runtime by the predicted parallel runtime, we simply subtract them. Then, we speculate on all methods with a predicted number of saved cycles greater than a threshold (THRES). The fork point is placed so that the predicted probability of violation is less than a selected ratio (RATIO). The idea here is that, eventually, we are really trying to trim from the runtime as many clock cycles as possible. The complexity and runtime information needed for this heuristic are identical to that for SP-SU.

3.3 Multi-Pass Heuristics

Simple and single-pass heuristics assume that the speculation decision for one method call is independent of the decision for all other methods. With nested speculation, the decision to speculate on a specific method depends on the decision for any parent methods in the call graph. The multi-pass heuristics explore these interactions by making multiple passes over the profiling data. With each pass, the decision for a specific method is reflected upon the information used to decide for its children in the following pass. Multi-pass heuristics attempt to limit over-speculation that may result from double-counting the potential benefits of speculation for both parent and child methods. Their main drawback is the additional complexity of the multiple passes.

The usefulness of the multi-pass mechanism is illustrated in Figure 2. We initially select to speculate on `Foo` using the first set of fork/join points. We will execute the work in `Foo` (`work_A`, `work_B`, and `work_C`) in parallel with (`work_D`). Next, we consider whether to speculate on `Bar`. With the single-pass heuristics, we would make the decision in isolation. We would consider the potential speedup if we executed the work in `Bar` (`work_B`) in parallel with the work after it ends (`work_C` *and* `work_D`). The multi-pass heuristics take into account that the code after the end of `Foo` (`work_D`) has already been selected for speculation. Hence, the decision for `Bar` will consider the speedup from executing in parallel `work_B` and `work_C`.

Best Speedup Heuristic with Parent Info (MP-SU): This heuristic builds upon SP-SU. Each method is evaluated for its predicted relative speedup and the one with the best speedup is chosen for speculation. Then, we revisit each of its child methods and adjust the runtime of their speculative region to take into account that the code following the parent method will be executed speculatively as well. After all of the children methods are adjusted, the predicted relative speedups are calculated again, the method with the best speedup is chosen for speculation, and the process repeats for its own children. The process continues until no methods remain with a predicted speedup greater than the selected threshold (THRES). Despite the additional passes and complexity, the information required by MP-SU is identical to that for its single-pass equivalent.

```
Foo(){
   /* fork point 1*/
   work_A;

   Bar(){
      /* fork point 2*/
      work_B;
   }

   /* join point 2*/
   work_C;
}

/* join point 1*/
work_D;
```

Figure 2. Illustration of a nested speculation.

Most Cycles Saved Heuristic with Parent Info (MP-CS): This heuristic builds on SP-CS. We start with the method which saves the largest number of cycles above the selected threshold (THRES) and has a predicted violation rate below the selected ratio (RATIO). Then we update the number of cycles expected to save by speculating on any of its child methods and repeat the whole process. By relying on clock cycles saved, MP-CS hopefully selects useful methods that are significant contributors to the overall execution time.

Most Cycles Saved Heuristic with No Nesting (MP-CSNN: Similar to MP-CS, this heuristic speculates first on the method that can save the largest number of cycles. It only selects methods with a predicted number of saved cycles greater than a threshold (THRES) for which the predicted violation rate is less than a determined ratio (RATIO). Once a method is selected, its children are immediately excluded from selection for speculation in the multi-pass process. In other words, this heuristic disallows nested speculation to completely eliminate the potential pitfall of double-counting the benefits. While MP-CSNN uses multiple passes as well, it is faster to implement than MP-SU and MP-CS as it quickly disqualifies many methods.

4 Experimental Methodology

This section summarizes the CMP model, benchmarks, and tools used to evaluate the potential of the heuristics. The evaluation of TLS hardware alternatives or compiler implementation issues is beyond the scope of this paper.

4.1 CMP Architecture Model

The CMP architecture includes four Pentium III processors (3-way out-of-order CPU). Each processor includes 32-KByte L1 instruction and data caches. The processors access a shared, on-chip, 256-KByte L2 cache over a cache-coherent bus. The L2 bus allows inter-processor commu-

Benchmark	Description
compress	Lempel-Ziv compression
jack	Java parser generator
javac	Java compiler from the JDK 1.0.2
jess	Java expert shell system
mpeg	Mpeg layer 3 audio decompression
raytrace	Raytracer that works on a dinosaur scene
barnes	Hierarchical N-body solver
water	Simulation of water molecules

Figure 3. The Java benchmarks used in this study.

nication and detection of inter-thread dependencies during TLS execution. Four processors are sufficient for this evaluation because limit studies for method-level speculation have shown that few applications can attain more than a 4x speedup [16]. Additionally, most upcoming CMPs will include 2 to 4 processors.

We assume that each processor includes a infinite store buffer that tracks speculative updates at word granularity during TLS execution. Store buffers are drained when a speculative thread becomes non-speculative. For the results in Section 5, we use a single speculative buffer per processor. We have performed experiments with double-buffering as well, which lead to similar results from the point of the heuristics, hence we omit them for brevity. We set the overhead for forking, retiring, or squashing a speculative thread to 70 clock cycles, which is a reasonable estimate for a CMP with hardware support for coarse-grain TLS [9, 6].

4.2 Benchmarks and Simulation Tools

We used a large set of Java benchmarks because Java is an object-oriented language which favors a large number of method calls that could stress our heuristics. In addition, a virtual machine with just-in-time compilation capabilities is a good place for implementing the heuristics for method-level speculative parallelization. For brevity, we only present the results for the 8 most representative benchmarks shown in Figure 3. The first six belong to the SpecJVM98 suite, while the remaining two are Java versions of SPLASH-2 benchmarks.

We use trace-driven simulation for our evaluation. First, we run benchmarks on native Pentium hardware with instrumentation code and produce a detailed trace. We run each benchmark multiple times within the same execution to eliminate JIT overheads and use a large heap space to avoid interference from the garbage collector. The trace includes method markings (calls/returns) and the addresses for all loads and stores. It also includes a timestamp for each address that represents the cycle count in the native run. Timestamps are used to account for timing events unrelated to inter-thread communication (e.g. non-memory instructions). They are re-calibrated to remove the instrumentation overhead before using traces for analysis.

The main evaluation tool is a trace analyzer that performs two tasks. First, it analyzes the trace to select the methods to speculate on according to the studied heuristic. As part of this process, the analyzer averages profiling data from the multiple invocations of each method. For each selected method, its fitness according to the heuristic is entered into a scoreboard structure for later use. The fitness is expressed as a number from 0 to 1 (higher is better). Next, the analyzer simulates the execution of the specific trace on the CMP architecture. This step involves parsing the behavior of loads and stores in the memory hierarchy of the 4-processor CMP while taking into account the overhead of forking, retiring, and squashing threads identified in the previous step. Detailed statistics are maintained for the outcome of speculation and the utilization of the four processors.

The analyzer also models speculative thread preemption, which is used to avoid performance loss due to excessive imbalance. Short threads can tie up processor and buffer resources while waiting for longer, less speculative threads to retire. This may prevent other, more profitable, speculative threads from being forked. The preemption technique uses the scoreboard information. When a method fork is reached and no processor is available, its fitness is compared to that of the currently running threads. If some of the running threads have lower fitness and are more speculative than the one about to be forked, we squash the one with the lowest fitness and fork the new thread. For nested speculation, we adjust the fitness of the child under consideration by multiplying with that of the parent method. Currently, we do not adjust the fitness of running threads based on their execution time so far. When a thread completes, we adjust its fitness in the scoreboard to reflect whether it has successfully retired or was squashed due to a violation. This preemption technique can be implemented in the TLS runtime that controls thread forking without a significant increase in fork overhead.

Finally, we can select the methods that the trace analyzer considers based on the expected support for return value prediction in the CMP architecture. We currently model three types of prediction: none, single value, and perfect. "None" does not speculate on any non-void method. "Single value" only speculates on methods that return a single value throughout the execution of the program (e.g. an error code). "Perfect" allows for speculation on any method and assumes that its return value can be perfectly predicted. We discuss the interaction of the heuristics with return value prediction in Section 5.

The trace-based methodology allows for fast, high-accuracy simulations. The traces contain all the data needed to simulate the memory system behavior such as L1 and L2 hits/misses, inter-processor communication, and dependency violations. For non-memory instructions, we assume that their timing is the same as in the native runs and do not simulate their execution. Parallel execution changes the latency of certain loads (inter-processor transfers or L2 hits instead of L1 misses) and those may affect some non-memory instructions. Still, out-of-order processors can tolerate the latency of on-chip accesses. Only off-chip accesses (L2 misses) cause significant changes in the behavior of the processor [12]. For the benchmarks we studied, the 256-KByte L2 cache is large enough to hold their working set at all times, hence there is no difference in L2 misses between parallel and sequential execution. The speed advantage of our methodology was critical during the tuning phase of the heuristics, which required thousands of simulations for parameter sweeps.

4.3 Oracle Heuristic

As an extra comparison target, we have implemented a perfect oracle for selecting speculative threads. The oracle allows us to measure the maximum method-level parallelism available in these applications. Such a heuristic is impossible to implement in a practical system. The oracle has the following properties. It can make a separate decision for each individual invocation of a method. The fork points selected always lead to correct speculation (no violations). There is no overhead for forking or retiring threads. Effectively, the oracle heuristic is only limited by true dependencies between methods. The implementation of the oracle relies on the fact that the analyzer has access to the whole execution trace for each benchmark and includes the architecture model.

5 Evaluation

This section presents the evaluation of the thread selection heuristics on the simulated 4-way CMP system.

5.1 Heuristic Tuning

Before we apply the heuristics to the benchmarks and compare the performance they deliver, we needed to determine the optimal parameter values to be used with each heuristic. To accomplish this, we ran sweeps of all of the parameter values for each of the heuristics. Due to space considerations, we only present the results for some of the sweeps for the jess benchmark using the SI-RT, SP-SU, MP-SU, and MP-CS heuristics in Figure 4. Jess was fairly representative of the behavior we saw with most benchmarks during parameter tuning. Figure 4 presents both speedups over sequential execution and violation counts.

The SI-RT heuristic achieved the best results with the MIN and MAX runtime parameters chosen to be 10^3 cycles and 10^7 cycles respectively. The SI-SC heuristic generally performed best with a MAX threshold set to 10^5 store accesses. The parameters indicate that relatively large speculative threads are often needed to hide thread overhead and avoid load imbalance.

The SP-SU heuristic achieved the greatest speedup when the THRES value for the predicted relative speedup is set as low as possible (1.2 to 1.001). This suggests that this heuristic is very conservative in its selection, and when it is used, its threshold should be very low to allow it to be more aggressive. The MP-SU heuristic, on the other hand, achieved its greatest results with the THRES value set considerably higher, at 1.4. Even though the multi-pass heuristics are always more conservative than their single-pass equivalents when cycles saving are used, this is not always the case when speedups are concerned. The SP-CS and the MP-CS heuristics achieve their best results when the threshold value (THRES) is approximately 10^5, and when minimum predicted success rate of speculation is approximately 0.3. In other words, the allowed violation ratio (RATIO) is 0.7.

Figure 4 also shows the effect of the return value prediction method used with the trace-based simulations for method-level speculation. For the SP-SU and MP-SU heuristics, we present the speedup with the three supported prediction schemes. Speculating only on functions that are declared to return void severely limits the effectiveness of method-level speculation as most functions return some value. Adding speculation for functions that always return the same value gets us almost as much benefit as perfect return value prediction. This is due to the high frequency of virtually static error codes and the fact that in many cases the return value of a method is not used at all. Furthermore, our heuristics tend to select large methods for speculation, which rarely return anything but an error code. For most other benchmarks, the speedup difference between constant and perfect value prediction is even smaller. Since we are evaluating the potential of the speculation heuristics, the remainder of the numbers we present assume perfect return value prediction.

5.2 Heuristics Analysis

Figure 5 shows the speedup achieved using each of the proposed heuristics with the parameters values identified in the previous section. The speedups are relative to the execution time of the original sequential code for each benchmark. The first observation is that all practical heuristics lead to positive speedups. They are able to identify profitable threads that lead to sufficient useful parallelism to hide the speculation overheads. The maximum speedup achieved with the heuristics ranges from 1.4 to 1.8. This is a significant improvement over an out-of-order processor, especially for hard integer applications like javac and jess. For some of the lowest performing benchmarks such as raytrace and water, there is additional loop-level, speculative parallelism one can extract. Nevertheless, loop-level TLS is beyond the scope of this paper. It is also interesting to compare the practical heuristics to the perfect oracle. The maximum speedup with the heuristics is within 80% to 95% of the oracle speedup, which incurs no violations and no thread overhead. The result suggests that the proposed heuristics are quite effective.

Figure 5 also allows us to compare the different heuristics. While there is not single obvious winner across all benchmarks, some heuristics perform noticeably better than others. In general, the single-pass heuristics (SP-SU and SP-CS) perform as well or better than both simple and multi-pass heuristics. The multi-pass heuristics are more conservative in their selection of speculation candidates, and therefore suffer from under-speculation. The simple heuristics suffer from over-speculation for certain benchmarks. Still, simple heuristics are consistently better than multi-pass heuristics which suggests that over-speculation is better than under-speculation. Another interesting observation is that the Most Cycles Saved heuristics (SP-CS, MP-CS, MP-CSNN) tend to perform slightly better than their Best Speedup counterparts (SP-SU, MP-SU). This confirms our expectation that the absolute number of cycles saved is more important than the relative speedup of each speculative method candidate. Overall, the SP-CS heuristic (single-pass most cycles saved with no parent info) exhibits the best average performance.

Figure 6 analyzes the behavior of speculative threads

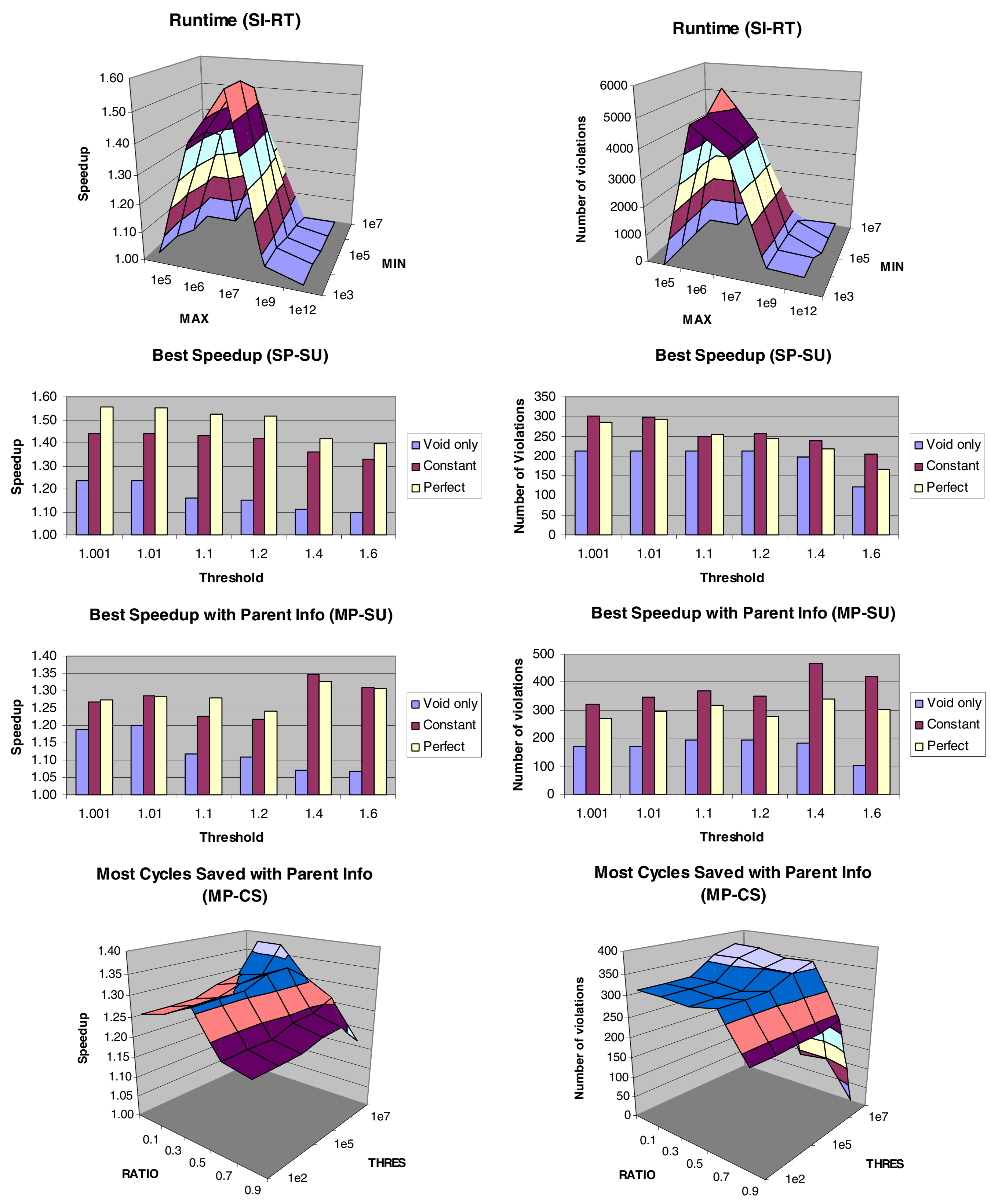

Figure 4. Results of the parameter sweep on the jess benchmark. The graphs on the left present speedups over sequential execution, while the graphs on the right present the number of violations.

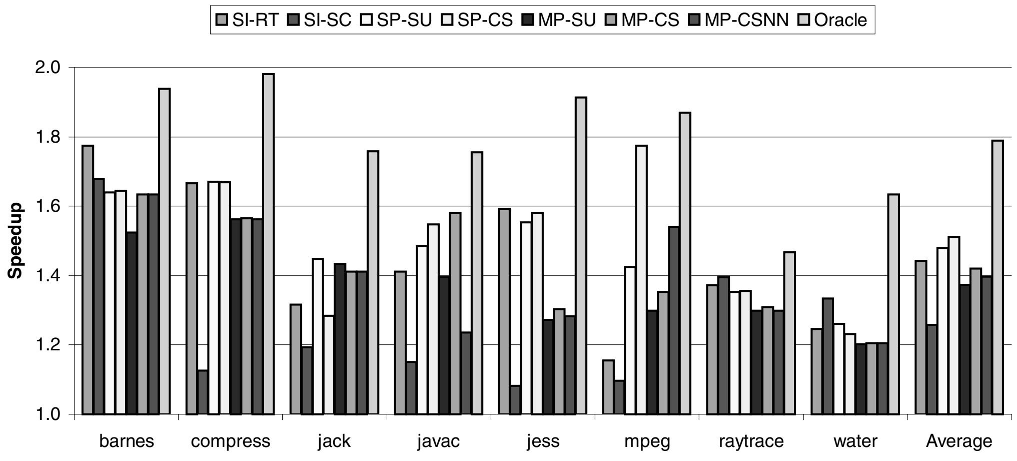

Figure 5. Overall speedups for each benchmark and heuristic.

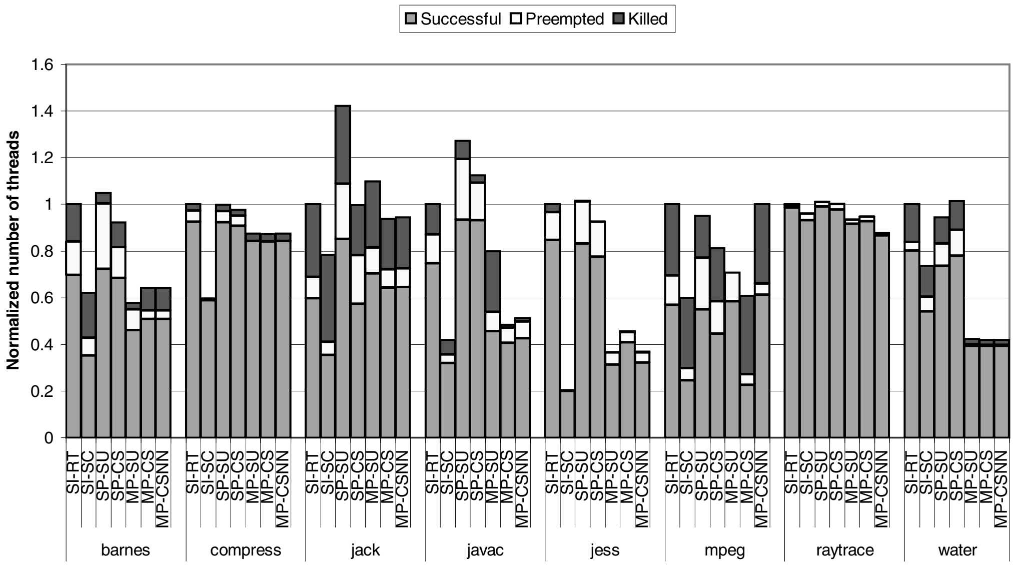

Figure 6. The breakdown of speculative threads generated for each benchmark.

forked with each heuristic. The height of each bar (total number of threads) is normalized to the number of speculations performed by the simple runtime heuristic SI-RT. Each bar is split into three sections: threads that completed successfully, threads that were preempted by another speculative thread considered more profitable at the time, and threads that were squashed due to violations. Single-pass and multi-pass heuristics tend to be more conservative than simple heuristics in terms of forking threads. For benchmarks like jack and mpeg, the selectivity in speculation is positive as it eliminates some of the violating threads of the simple heuristics without affecting significantly the profitable threads. For benchmarks like water or jess, on the other hand, the selectivity may lead to excessive under-speculation. This is particularly true for the multi-pass heuristics. A large number of preempted threads has no effect on performance (e.g. jack, javac, and mpeg). If processors are idling, it is not harmful to start a speculative thread on them even if it is likely that it will be preempted soon.

Figure 7 shows the average execution time breakdown across the 4 processors in the CMP system[1]. The height of each bar is normalized to the sequential execution time. Each bar has three segments: the time spent doing useful work on successful threads, the time idling due to the lack of speculative threads or due to a nested speculative thread reaching the end of a speculative region, and the time spent on work wasted due to thread violations, thread preemptions, and speculation overheads. The average useful time is always 25% of original time as we take initial useful work in the sequential program and split it across 4 processors. The amount of wasted time correlates with the complexity of the heuristic. Simple heuristics have a significant wasted component, which more advanced heuristics are of-

[1]Because Figure 7 provides an average across all four CPUs, the time it presents does not always correlate to the speedups in Figure 5. If two processors are making progress while the rest are idling, the speedup is two, while the average time breakdown shows an even break between idle and useful time.

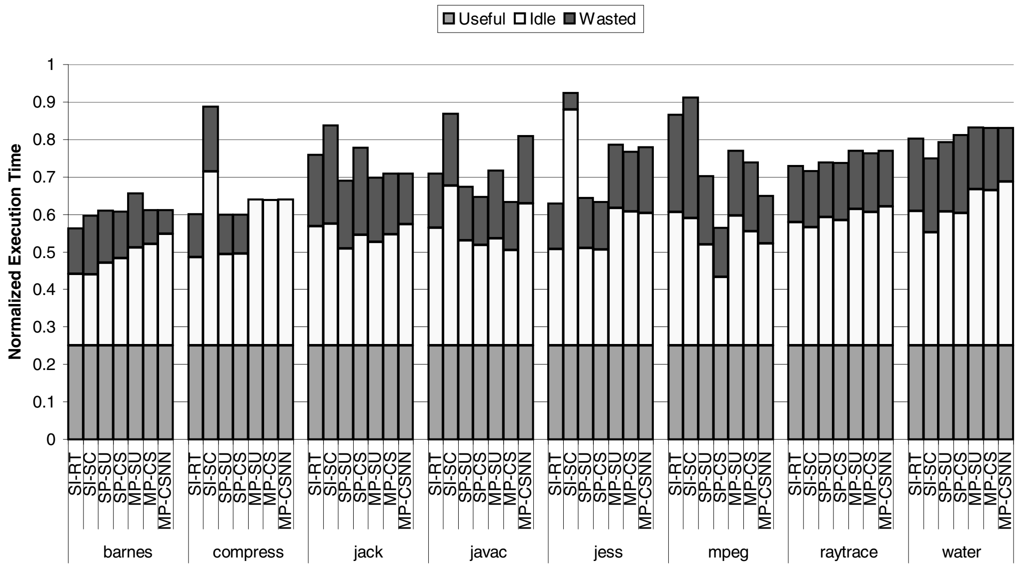

Figure 7. The breakdown in execution time for each benchmark and heuristic. The y-axis is normalized to the execution time of the sequential benchmark. Each bar is broken up into useful time, idle time, and wasted time.

ten able to eliminate, as seen with the barnes benchmark. However, in some cases the multi-pass heuristics simply increase the idle time due to under-speculation without making a significant change to the wasted time (e.g. compress). The single-pass heuristics achieve the best balance of under-speculation (idle time) and over-speculation (wasted time).

Figure 8 shows the maximum store buffer size required for speculative execution with each heuristic. We track speculative updates at word granularity both for buffering and violation detection purposes. The data suggest that a buffer size of 16 Kbytes per processor would be sufficient for all benchmarks with all heuristics. This buffer requirement can be easily met by the first level for both current and future CMP designs. Certain heuristics such as SI-RT require significantly less speculative buffering and can be fully satisfied with a couple of Kbytes. If the store buffer capacity is limited in a specific TLS implementation, a compiler may want to use a heuristic like SI-RT in order to avoid buffer overflows which cause significant performance loss due to serialization or main memory accesses [8].

6 Related Work

There has been significant research on TLS since the original proposal [18]. Most of the work has focused on hardware issues, while evaluations have mostly focused on loop-level parallelism [9, 14, 6, 19]. Compiler support for TLS has also focused on loops as a primary source for speculative threads [23, 7]. JRPM uses hardware supported profiling and heuristics to automatically select speculative threads from loops [3, 5]. A similar system could be used to implemented the heuristics discussed in this paper.

The potential of method-level speculation have been studied by [16, 21]. The heuristics used in these studies to select speculation points are fairly simple compared to our heuristics. Warg et al. examined a heuristic based solely on minimum runtime [22], which is similar to our SI-RT heuristic. Our work examines a much wider variety of heuristics. The speedups in [22] are not directly comparable to ours due to the different assumptions about the hardware systems. First, they assume a single-issue, in-order machine with perfect, one-cycle, memory access, while we assume an out-of-order processor with a cache hierarchy. They also assume multiple speculative buffers per CPU in the system. Hence, the reported speedups in [22] are much higher than ours but probably less realistic. Chen et al. investigated method-level speculation in a JVM environment [4], but selected all speculation points by hand.

The Multiscalar and SpMT compilers [20, 2] take advantage of both loop-level and method-level speculation. They operate at a basic block granularity as they use special hardware support for fine-grain parallelism. Their heuristics use profiling data such as dynamic instruction count. They do not compare the heuristics used to any others (or a perfect oracle) and provide little insight to how to tune them for performance. Our work is complementary to these studies as we evaluate alternative heuristics for thread selection. However, our work focuses on method-level speculation only.

7 Conclusions

TLS allows sequential code to benefit from the upcoming CMP chips. To realize this potential, we need compilers capable of selecting speculative threads that maximize the benefits of TLS while minimizing its performance overheads. This paper presented and analyzed a set of heuristics for automatic selection of speculative threads at method boundaries by a dynamic or profile-driven compiler. The heuristics involve increasing amounts of profiling information and selection complexity. As heuristics get more complex, the goal is to filter out methods that are unlikely to lead to profitable speculation.

	barnes	compress	jack	javac	jess	mpeg	raytrace	water
SI-RT	0.31	0.18	0.39	2.05	0.26	0.76	1.64	0.20
SI-SC	12.02	6.47	0.19	3.51	0.15	13.02	1.64	1.45
SP-SU	8.11	6.48	0.39	1.08	0.30	13.02	1.64	0.55
SP-MC	0.31	6.48	0.39	2.57	0.30	15.29	1.64	0.22
MP-SU	12.01	6.48	0.39	0.30	0.30	1.27	1.27	1.38
MP-CS	12.02	6.48	0.39	0.30	0.30	1.64	1.27	1.38
MP-CSNN	12.02	6.48	0.39	2.57	0.30	13.02	1.27	1.38

Figure 8. Speculative store buffer size for various combinations of heuristics and benchmarks. All the values are Kbytes.

We evaluated the potential of the heuristics with a set of Java applications. The heuristics lead to execution speedups of 1.4 to 1.8 even for difficult integer applications, which is within 80% to 96% of the performance possible with a perfect oracle. We demonstrated that single-pass heuristics lead to the best average performance by consistently providing a good balance between over- and under-speculation. On the other hand, multi-pass heuristics often lead to significant under-speculation and are frequently outperformed by the simple heuristics. The buffering requirements of the generated speculative threads are reasonable for current and future CMP designs.

References

[1] V. Agarwal et al. Clock rate versus IPC: the End of the Road for Conventional Microarchitectures. In *the Proceedings of the Intl. Symposium on Computer Architecture*, June 2000.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *the Proceedings of the Symposium on Parallel Architectures and Algorithms*, June 2002.

[3] M. Chen and K. Olukotun. TEST: A tracer for extracting speculative threads. In *the Proceedings of the Intl. Symposium on Code Generation and Optimization*, Mar. 2003.

[4] M. K. Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *the Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[5] M. K. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *the Proceedings of the Intl. Symposium on Computer Architecture*, June 2003.

[6] M. Cintra et al. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the Intl. Symposium on Computer Architecture*, June 2000.

[7] J. Dou and M. Cintra. Compiler Estimation of Load Imbalance Overhead in Speculative Parallelization. In *the Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Oct. 2004.

[8] M. Gatzaran et al. Tradeoffs in Buffering Memory Sate for Thread-Level Speculation in Multiprocessors. In *the Proceedings of the Intl. Symposium on High Performance Computer Architecture*, Feb. 2003.

[9] L. Hammond et al. Data Speculation Support for a Chip Multiprocessor. In *the Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[10] T. Heil and J. Smith. Relational Profiling: Enabling Thread Level Parallelism in Virtual Machines. In *the Proceedings of the Intl. Symposium on Microarchitecture*, Dec. 2000.

[11] R. Kalla. Simultaneous Multi-threading Implementation in POWER5. In *the Conference Record of Hot Chips 15 Symposium*, Aug. 2003.

[12] T. Karkhanis and J. Smith. A Day in the Life of a Cache Miss. In *the Proceedings of the 2nd Workshop on Memory Performance Issues*, June 2002.

[13] P. Kongetira. A 32-way Multithreaded Sparc Processor. In *the Conference Record of Hot Chips 16*, Aug. 2004.

[14] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *the Proceedings of the Intl. Conference on Supercomputing*, June 1999.

[15] C. McNairy. Montecito: The next Product in the Itanium Processor Family. In *the Conference Record of Hot Chips 16*, Aug. 2004.

[16] J. T. Oplinger et al. In Search of Speculative Thread-Level Parallelism. In *the Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[17] M. K. Prabhu and K. Olukotun. Using Thread-Level Speculation to Simplify Manual Parallelization. In *the Proceedings of the Conference on Principles and Practices of Parallel Programming*, June 2003.

[18] G. S. Sohi et al. Multiscalar Processors. In *the Proceedings of the Intl. Symposium on Computer Architecture*, June 1995.

[19] J. G. Steffan et al. A Scalable Approach to Thread-Level Speculation. In *the Proceedings of the Intl. Symposium on Computer Architecture*, June 2000.

[20] T. N. Vijaykumar and G. S. Sohi. Task Selection for the Multiscalar Architecture. *Journal of Parallel and Distributed Computing*, 58(2):132–158, Aug. 1999.

[21] F. Warg and P. Stenstrom. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *the Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[22] F. Warg and P. Stenstrom. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *the Proceedings of the Int. Parallel and Distributed Processing Symposium*, 2003.

[23] A. Zhai et al. Compiler Optimizatoin for Scalar Value Communication Between Speculative Threads. In *the Proceedings of the Intl. Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[24] G. Zilles and G. Sohi. A Programmable Co-Processor for Profiling. In *the Proceedings of the Intl. Symposium on High Performance Computer Architecture*, Jan. 2001.