Dynamically Dispatching Speculative Threads to Improve Sequential Execution

YANGCHUN LUO, Advanced Micro Devices

ANTONIA ZHAI, University of Minnesota

Efficiently utilizing multicore processors to improve their performance potentials demands extracting thread-level parallelism from the applications. Various novel and sophisticated execution models have been proposed to extract thread-level parallelism from sequential programs. One such execution model, Thread-Level Speculation (TLS), allows potentially dependent threads to execute speculatively in parallel.

However, TLS execution is inherently unpredictable, and consequently incorrect speculation could degrade performance for the multicore systems. Existing approaches have focused on using the compilers to select sequential program regions to apply TLS. Our research shows that even the state-of-the-art compiler makes suboptimal decisions, due to the unpredictability of TLS execution. Thus, we propose to *dynamically* optimize TLS performance.

This article describes the design, implementation, and evaluation of a runtime thread dispatching mechanism that adjusts the behaviors of speculative threads based on their efficiency. In the proposed system, speculative threads are monitored by hardware-based performance counters and their performance impact is evaluated with a novel methodology that takes into account various unique TLS characteristics. Thread dispatching policies are devised to adjust the behaviors of speculative threads accordingly.

With the help of the runtime evaluation, where and how to create speculative threads is better determined. Evaluated with all the SPEC CPU2000 benchmark programs written in C, the dynamic dispatching system outperforms the state-of-the-art compiler-based thread management techniques by 9.4% on average. Comparing to sequential execution, we achieve 1.37X performance improvement on a four-core CMP-based system.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—*Multiple-instruction-stream, multiple-data-stream processors (MIMD)*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Dynamic optimization, thread-level speculation, multicore

ACM Reference Format:

Luo, Y. and Zhai, A. 2012. Dynamically dispatching speculative threads to improve sequential execution. ACM Trans. Architect. Code Optim. 9, 3, Article 13 (September 2012), 31 pages.
DOI = 10.1145/2355585.2355586 <http://doi.acm.org/10.1145/2355585.2355586>

1. INTRODUCTION

The wide deployment of multicore and multithreading architectures has brought forth significant volume of computation power. Efficiently utilizing these parallel

This work is supported in part by a grant from National Science Foundation under CNS-0834599, EIA-0220021; a contract from Semiconductor Research Cooperation under SRC-2008-TJ-1819, and gift grants from HP, IBM and Intel.

Authors' addresses: Y. Luo, Advanced Micro Devices, 1237 East Arques Avenue, Sunnyvale, CA 94085; email: yangchun.luo@amd.com; A. Zhai, Department of Computer Science and Engineering, University of Minnesota, 200 Union Street, Keller Hall 4-192, Minneapolis, MN 55455; email: zhai@cs.umn.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1544-3566/2012/09-ART13 \$15.00

DOI 10.1145/2355585.2355586 <http://doi.acm.org/10.1145/2355585.2355586>

Fig. 1. By speculation, code regions that are otherwise nonparallelizable can execute in parallel.

architectures and exploiting their performance potentials demand efficient extraction of thread-level parallelism (TLP) in diverse applications. However, automatically extraction of TLP is challenging.

Automatic thread extraction done by the compiler is desirable, but is challenging due to ambiguous memory aliasing and its effectiveness has been limited for general-purpose applications. In the presence of complex control flow and ambiguous pointer usage, traditional parallelization schemes must conservatively ensure correctness by synchronizing all potential dependences, which inevitably limits parallel performance.

In Figure 1(a), the compiler attempts to partition the sequential execution into two parallel threads T1 and T2. As a result, two pointer-based memory accesses, a logically earlier store and a logically later load, are allocated to two different threads, and their relative order is inverted in parallel execution. This partitioning is safe only if these two instructions access different memory locations. However, their memory addresses are unknown at the compilation time. Hence the compiler is forced to give up parallelizing this code region. Traditional compilers have very limited capability in parallelizing pointer-based code.

1.1. Speculative Thread Execution Model

A number of execution models have been proposed to extract parallelism in applications with ambiguous dependences. This article focuses on one such model: Thread-Level Speculation (TLS) [Knight 1986; Akkary and Driscoll 1998; Franklin and Sohi 1992; Cintra and Torrellas 2002; Dubey et al. 1995; Sohi et al. 1995; Gupta and Nim 1998; Hammond et al. 1998; Marcuello and González 1999; Oplinger et al. 1999; Steffan et al. 2000; Tsai et al. 1999]. TLS allows the compiler to parallelize a sequential program without first proving the independence among the extracted threads. During runtime, the underlying hardware keeps track of each memory access, determines whether any data dependence is violated, and reexecutes the violating thread(s) as needed.

Figure 1(b) illustrates the concept of speculative execution. The threads are numbered according to their original sequential order. If no dependence is violated,

the speculative thread commits (thread 2); otherwise, it is squashed and reexecuted (thread 3). TLS empowers compilers to parallelize program regions that were previously nonparallelizable.

1.2. Limitations of the Static Approaches

To extract speculative threads, the compiler or the runtime system must perform trade-off analysis to determine the performance impact of parallelization. Existing work on TLS mostly relies on compilers to statically analyze programs, and then extract speculative threads [Liu et al. 2006; Zhai et al. 2002, 2004; Wang et al. 2005; Johnson et al. 2004; Vijaykumar and Sohi 1998; Marcuello and González 2002; Johnson et al. 2007]. We refer to them as *static* thread management.

In static approaches, compilers often analyze extensive profile information to estimate the performance impact of speculative threads and then determine where and how to create speculative threads. Being able to perform global and even interprocedure analyses, compilers can extract coarse-grain parallelism in which speculative threads may contain several thousand instructions. However, it is difficult and sometimes impossible for compilers to accurately estimate the performance impact even when extensive profile information is available. We have identified four key reasons.

1.2.1. Profiling Information Is Insufficient for Estimating the Costs of Speculation, Synchronization, and Other Overheads. The performance impact of speculation is determined by the number of useful execution cycles in the speculative threads that can overlap with the execution of the nonspeculative thread. To determine this overlap, the compiler must determine the size of consecutive speculative threads, the cost of speculation failures and the timing of their occurrence, the cost of synchronization, and the cost of managing speculation.

However, these factors are often difficult to estimate even with accurate profiling information. For example, the rate of speculation failures not only depends on the number of interthread data dependences, but also on the timing of its occurrence. For loops with complex control flow, it is difficult to determine whether the load/store instructions in consecutive iterations are dependent; and if they are, which loads will cause dependence violations at runtime. Probability-based data and control dependence profiling, which is used in many TLS compilers, is insufficient to come up with such estimations.

1.2.2. Performance Impact of Speculative Threads Depends on the Underlying Hardware Configuration. Because speculative threads must share the underlying hardware resources, the configuration of the underlying hardware can change the behaviors of these threads. In particular, interaction between the speculative threads and the cache components has a profound impact on performance. On the one hand, speculative threads, even when they fail, can potentially bring data items into the cache and improve the performance of the nonspeculative thread. On the other hand, speculative threads can modify data items that are shared with the nonspeculative thread and introduce misses that otherwise do not exist. Furthermore, the Chip-Multiprocessor (CMP) architecture effectively increases total cache size. However, as data is spread across multiple caches, CMP introduces coherent misses that do not exist in sequential execution. The impact of such cache behaviors is difficult for the compiler to determine even with accurate profile information.

1.2.3. Speculative Thread Behaviors Are Input-Dependent. The performance of speculative threads is often dependent on the characteristics of the input data. TLS takes advantage of probabilistic data dependences by speculatively assuming that these dependences do not exist. This mechanism is beneficial only if these data dependences are

infrequent. Frequently occurring data dependences should be synchronized. Choosing the threshold that separates frequent and infrequent dependences is a delicate matter, since a high threshold leads to excessive speculation failures, and a low threshold leads to serialization of the threads. However, once this threshold is chosen and the set of frequently occurring dependences are synchronized, this decision is compiled into the binary, even if the decision is not proper for some input sets.

When extracting speculative threads for 256.BZIP2, we collected profiles using the train input set and decided which dependences to speculate on. When the program executes with the three ref input sets source, graph, and program, we found that the percentage of total execution cycles that are wasted due to speculation failure was 25%, 40%, and 31%, respectively.

Therefore, speculative threads that are created to improve performance under one workload may potentially degrade performance when the input set changes.

1.2.4. Speculative Threads Experience Phase Behavior. For some applications, it has been reported that the same codes may exhibit different performance characteristics as the program enters different phases of execution [Nagpurkar et al. 2006]. We refer to this behavior as phase behavior. In the context of TLS, phase behavior can be manifested as changing the effectiveness of speculative threads: speculative threads that improve performance during certain phases of execution can potentially degrade performance during other phases of execution. Probabilistic profiles cannot capture this behavior, as speculative decisions that are compiled statically into the binary cannot adapt to this behavior.

Phase change is a natural phenomenon in real-world applications, and can occur as a result of ordinary programming constructs. For example, in algorithms that search for the maximum or minimum in a large data set, the frequency of updating the global variables decreases as the algorithm progresses. Thus a loop that is not fit for speculative execution earlier in the program can become a good candidate during later phases of the execution.

To summarize, the impact of speculative threads is often multifold: they can commit speculative work, move data between various memory and cache components, and compete for shared resources with the nonspeculative thread. A crucial task in creating speculative threads is to allocate work to each thread, taking into consideration interthread dependences and resource requirements. Such decisions are not unique and it is difficult for the compiler to make decisions that are optimal for all programs under all workloads on a large variety of machines.

1.3. Dynamic Thread Dispatching System Overview

Dynamically managing speculative threads is an attractive alternative. When the execution of speculative threads is monitored, it is possible for the runtime system to accurately determine their impact and adjust their behavior accordingly. To build such a system, the following issues need to be addressed:

1.3.1. Thread Monitoring. The performance profile of speculative threads must be collected dynamically. Such profiles can be application-dependent, such as loop iteration count; architecture-dependent, such as memory access latency; or both, such as cache miss rate. Hardware-based performance counters [Eyerman et al. 2006; Mericas 2006] are programmed to collect such information.

1.3.2. Thread Evaluation. Once the profile is collected, it is analyzed to determine the efficiency of speculative threads. The interaction between the main and the speculative threads often complicates the evaluate process. For instance, data brought in by the speculative threads can be used by the main thread but can also displace useful data

needed by the main thread. Therefore, we need a robust methodology. The evaluation results are stored in a hardware-based table that is kept in sync with all processor cores.

1.3.3. Thread Adjustment. Once the effectiveness of speculative threads is determined, the dispatching system can decide where to create speculative threads to maximize application performance. We will experiment the design space by creating various dispatching policies and compare their effectiveness in optimizing application performance.

1.4. Contributions

This article explores the feasibility and effectiveness of a runtime system that dynamically adjust the behavior of speculative threads based on their performance impact. Under the context of dynamic performance optimization for TLS, this article makes the following contributions:

- We propose an execution framework that allows the runtime system to evaluate speculative thread execution and dynamically adjust their runtime behaviors.
- We propose and evaluate dynamic performance evaluation methodologies that analyze execution cycle breakdown of speculative threads to determine their efficiency. We also discuss how hardware counters could be programmed to collect such cycle breakdown.
- We propose, implement and evaluate various dynamic performance dispatching policies that determine the priority in which the runtime system evaluates speculative threads for parallel execution. By evaluating these policies, we identify important runtime information and compiler annotations that are key to optimizing speculative threads.

The rest of this article is organized as follows: we first describe the compiler and simulation infrastructure in Section 2, and then discuss how to determine the performance impact for speculative threads in Section 3. Section 4 describes the necessary runtime support that facilitates performance profile collection and decision making. Dynamic thread dispatching policies are proposed in Section 5 and their performance evaluated in Section 6. Related work is discussed in Section 7. Finally we present our conclusions and future work in Section 8.

2. EXPERIMENT INFRASTRUCTURE

We have evaluated all benchmark programs written in C from SPEC CPU2000 suite. The execution of these benchmarks is simulated using *ref* input sets on a detailed cycle-accurate architectural simulator configured as a *four* core chip-multiprocessor. Due to their distinct differences, we treated the PLACE and ROUTE input sets of VPR as two benchmarks, referred as VPR-P and VPR-R in later sections.

2.1. Architectural Support for Speculation

Speculative threads are supported on a Chip-Multiprocessor (CMP) with the STAM-Pede extension [Steffan et al. 2000, 2005]. All processing cores have private first-level caches that are connected through a bus to the shared second-level cache. The STAM-Pede protocol extends the cache coherence protocol with two new states, speculatively shared (*SpS*) and speculatively exclusive (*SpE*), and transitions to and from these states. All the speculative threads are assigned with a unique ID that determines the logical order in which the threads commit. The thread ID of the sender piggybacks on all invalidation messages.

If a cache line is speculatively loaded, it enters the *SpS* or *SpE* state, and becomes susceptible to dependence violations. If an invalidation message arrives from a logically earlier thread for that cache line, speculation fails; and the thread is squashed. To recover a speculation failure, all cache lines in the *SpE* state are invalidated and all *SpS* lines transit into the shared state; the thread is then reexecuted from the beginning. When a thread is spawned to a busy core that is executing an earlier thread, the new thread is suspended and will be resumed when the earlier thread commits.

2.2. Compilation Infrastructure

Our compiler infrastructure is built on the Open64 Compiler [Open64 Developers 2001], an industrial-strength open-source compiler targeting Intel's Itanium Processor Family. We extended Open64 to extract speculative parallel threads from loops.

The compiler estimates the parallel performance of each loop based on the cost of synchronization and the probability and cost of speculation failure, using loop nesting profile, edge frequency profile, and data dependence frequency profile. The compiler then chooses to parallelize a set of loops that maximize the overall program performance based on such estimations [Wang et al. 2005].

To dynamically optimize where speculative threads should be spawned, we simply force the compiler to create a different executable in which every loop is parallelized. All binary codes are compiled with -O3 optimization level. In addition, TLS-specific optimizations, such as interthread register and memory resident value communication, and reduction operations, are applied by the compiler to those parallelized loops [Zhai et al. 2002; 2004; Wang 2007].

2.3. Simulation Infrastructure

We build our simulation infrastructure based on a trace-driven, out-of-order superscalar processor simulator. The trace-generation portion is based on the PIN instrumentation tool [Luk et al. 2005], and the architectural simulation portion is built on SimpleScalar [SimpleScalar LLC 2004].

The trace generator instruments all instructions to extract information such as instruction address, registers used, memory address for memory instructions, opcode, and etc. The entire trace of the instruction stream is output to disk files.

The simulator reads the trace file and translates the Itanium code bundles generated by the compiler into Alpha-like codes. The simulated pipeline model is based on SimpleScalar [SimpleScalar LLC 2004]. In addition to modeling register renaming, reorder buffer, branch prediction, instruction fetching, branching penalties, and memory hierarchy performance, we also extend the infrastructure to account for different aspects of speculative thread execution, including explicit synchronization through signal/wait, cost of thread commit/squash, and etc. Table I shows the architecture parameters of the TLS-enabled Chip Multiprocessor (CMP). Private L1 caches, each belonging to one core, are connected with a shared unified L2 cache through a bus.

The overhead of dynamic thread dispatching, such as trying an ineffective loop, is faithfully included in our simulation. To reduce simulation time, we have adopted the SimPoint-based sampling technique [Perelman et al. 2006] with 100 million instructions per sample and up to 10 samples per benchmark. Up to one billion instructions can be simulated for each benchmark. With the sample size of 100 million instructions, the side effect of warming-up is negligible.

3. DETERMINING THE PERFORMANCE IMPACT OF SPECULATIVE THREADS

In this section, we propose a methodology to quantitatively determine the efficiency of speculative execution at runtime. Our techniques build on execution cycle breakdowns that can be obtained through hardware-based, programmable performance monitors.

Table I. Processor Parameters

Pipeline Stages	Fetch/Issue/Ex/WB/Commit
Fetch/Issue/Commit Width	6 / 4 / 4
ROB/LSQ Size	128 / 64 entries
Integer Units	6 units / 1 cycle
Floating Point Units	4 units / 12 cycles
Private L1-Data Cache	64KB, 4-way, 32B
Private L1-Instruction Cache	64KB, 4-way, 32B
Memory Ports	2 Read, 1 Write
Branch Predictor	2-Level Predictor
Level1/Level2 Size	1 / 1024 entries
History Register	8 bits
Branch Mis-prediction Latency	6 cycles
Number of Cores	4
Shared L2-Unified Cache	2MB, 8-way, 64B
L1/L2/Memory Latencies	1 / 18 / 150 cycles
Thread Squash/Spawn/Sync Latencies	5 / 5 / 1 cycles

Fig. 2. Sample execution time breakdown of a loop invocation under two execution models: TLS on four core (TLS4) and sequential execution (SEQ). Sequential execution time of the loop predicted from the cycle breakdown of TLS4: a simple prediction pSEQ_s and a detailed prediction pSEQ_d. Bars are normalized to SEQ.

Details of these monitors and how to obtain the breakdowns will be addressed in Section 4.

For our purposes, cycles for TLS execution are broken into six segments:

- Busy*: cycles spent graduating non-TLS instructions;
- ExeStall*: cycles stalled due to lack of instruction-level parallelism;
- iFetch*: cycles stalled due to instruction fetch penalty;
- dCache*: cycles stalled due to data cache misses;
- Squash*: cycles wasted due to speculation failures; and
- Others*: cycles spent on various TLS overheads, including thread spawning, committing, and synchronization; as well as idling cycles due to unbalanced workloads.

Figure 2 shows the execution time breakdown, normalized to the execution time of the sequential execution, of a loop executing in TLS mode on four cores (TLS4) and executing sequentially (SEQ). Each segment in TLS4 is the aggregated cycles scaled down by *four* to show the relative speedup compared to SEQ.

To isolate the performance impact of the speculative threads, we attempt to *predict* the sequential execution time from the execution time breakdown of the TLS execution. A straightforward prediction, shown as the pSEQ_s bar in Figure 2, is to subtract Squash

and Others cycles from the total aggregated cycles of parallel execution. Each segment in $pSEQ_s$ is just four times of its counterpart from TLS4, since the latter is scaled down by four. In this prediction, Busy can be accurately predicted because the amount of useful work done in TLS mode and sequential execution is similar; iFetch is also similar in these two execution modes. However, execution stall and cache behaviors can change dramatically when the sequential program is decomposed into multiple threads. To improve our prediction, a more accurate model is developed to address the inaccuracies in predicting ExeStall and dCache segments.

3.1. Adjustment to Execution Stall

When the original program executes sequentially on one core, instructions from multiple iterations of the same loop are available for scheduling, and thus the core is able to effectively exploit instruction-level parallelism (ILP). However, when the same code is decomposed into multiple threads that are distributed to multiple cores, execution stall may increase since fewer instructions are available for scheduling. This effect correlates with the average number of dynamic instructions per thread (defined as *thread size*). When the thread size is much greater than the reorder buffer (ROB) size, the variances of execution stall between sequential and TLS execution can be negligible. However, when the thread size is smaller than or comparable to the ROB size, execution stall will increase considerably. Thus, to accurately predict execution stall in sequential runs, the execution stalls in parallel runs must be scaled down by a factor that is correlated to the ROB size of the processor and inversely correlated to the thread size. In our experiments, the following estimation is demonstrated to be effective for approximating this relationship:

$$1 + \frac{ROB_Size}{Issue_Width * Thread_Size}. \quad (1)$$

3.2. Adjustment to Data Cache Behaviors

Speculative threads have a significant impact on the first-level data cache (L1D) performance. Because speculative states are buffered at L1D, and L1D cache misses observed in speculative parallel execution may or may not occur during sequential execution; and vice versa. On the other hand, the second level unified cache (L2) is responsible for caching the working set for the entire process. Since the aggregation of all committed instructions on all cores in speculative parallel mode is the same as in the sequential mode, the two modes generate similar access patterns to L2. Thus, the performance of L2 remains mostly unchanged and we do not consider the performance impact of speculation on L2.

Our main approach is to classify L1D cache misses that occur during speculative parallel execution, and predict whether they would occur during sequential execution. We have identified the following scenarios that require special handling: (i) data brought into the L1D cache by a speculative thread that is eventually squashed; (ii) data modified by a speculative thread that is eventually squashed; (iii) data brought into a neighboring L1D cache; and (iv) data evicted from one L1D cache but available in other L1D cache(s). In the rest of this section, we will describe the four scenarios in detail.

Scenario I. Speculative threads that bring data into L1D can be squashed and then reexecuted. When this occurs, a straightforward classifier would count all the cycles, from the point of thread starting to the point of squashing toward Squash. Note that this classifier counts all stalls due to L1D cache misses as part of Squash. However, this simple classification can be inaccurate if the data is used when the thread reexecutes. This is because when the squashed thread reexecutes and accesses the data for a

Fig. 3. Impact of failed speculation on L1 data caches.

second time, it will not incur another cache miss, as shown in Figure 3(a). However, since Squash cycles are discarded in the prediction, the latency associated with this memory access is inadvertently not counted towards the sequential execution time. To rectify this counting inaccuracy, cycles stalled due to L1D miss during the squashed execution must be recorded separately and nevertheless contribute to the predicted sequential execution time.

Scenario II. When a data item is modified by the speculative thread that is eventually squashed, the cycles must be counted differently. As shown in Figure 3(b), the squashing mechanism invalidates the modified copy in the cache to ensure correctness. During reexecution, this data item will be reloaded into the L1D cache, and thus incur another cache miss. In sequential execution, the second cache miss would not occur. This case can be identified in speculative parallel execution, since the second miss would find an invalidated cache line with a matching tag.

Scenario III. Multiple speculative threads concurrently executing on multiple cores can further complicate L1D cache performance. When two threads on different cores load from the same address, they incur two L1D cache misses, as shown in Figure 4(a). However, in sequential execution, it is likely that the logically later miss on the second thread becomes a hit since the threads are executing on the same core. Thus, to correctly predict sequential performance, latency associated with L1D caches misses that return the data item in shared state must be omitted from the prediction.

Scenario IV. Speculative threads run on multiple cores, and thus are able to utilize multiple L1D caches. This can increase the effective associativity of the first level cache compared to the sequential execution. Thus, conflict misses that occur in the sequential execution may or may not occur in the parallel execution. As shown in Figure 4b, the conflict miss served by the second level cache in the sequential execution does not occur

Fig. 4. Impact of speculative parallel execution on L1 data caches.

in parallel execution; rather, it is served by a neighboring L1D cache. This results in a different latency compared to accessing L2. Therefore, we need to compensate the predicted sequential execution time with an L2 access latency each time this scenario occurs. To count the number of such occurrences, we extend each cache line with one additional bit to track whether this cache line is loaded by a logically later thread. We refer to this bit as *ExtraShared*. If a cache line with the *ExtraShared* bit set is evicted, it is *likely* that the load in a logically later thread would result in a conflict miss in sequential execution, so the count is incremented by one. Details regarding this extension will be discussed in Section 4.1.4. This method is simple and with minimal hardware support. It can capture most of the conflict misses, but not all situations. For instance, if the eviction happens in thread 2 (Figure 4(b)), it will not be counted because the *ExtraShared* bit is associated with the cache on which the first thread executes. Moreover, if a cache line loaded by more than one logically later thread is evicted, this method will only increase the count by one, in which case multiple increments are desired.

It is worth pointing out that cache behaviors make it difficult for static analysis to derive the impact of speculative threads accurately. If speculation failure often helps to fetch useful data into the L1 cache, a high failure rate can be benign, but if failed threads often invalidate useful data, even a moderate failure rate can be detrimental. In Section 4.1.4, we will provide a detailed description of the hardware performance monitors needed to classify the misses described above.

3.3. Performance Prediction Accuracy Evaluation

In this section, we evaluate how accurate we are able to predict sequential performance from speculative parallel execution. To simplify our evaluation, we focus only on loops

that are selected for parallelization by the compiler [Wang et al. 2005]. These loops are normally important and representative. For each loop invocation, three sets of execution cycles are obtained: the sequential execution cycle T_{SEQ} , the TLS execution cycle T_{TLS} , and the sequential cycle predicted from parallel execution time breakdown T_{pSEQ} . We use two metrics for evaluation: (i) the percentage of total loop execution we are able to correctly determine whether or not parallel execution outperforms sequential execution (*correctness measure*); and (ii) the difference, in cycles, between the predicted sequential execution and the real sequential execution cycle (*similarity measure*). We evaluated both metrics on three increasingly complex schemes: Base corresponds to the baseline prediction scheme described at the beginning of the section; Base+ExeStall incorporates the execution stall scaling described in Section 3.1; and Base+ExeStall+dCache further incorporates the data cache behavior classification described in Section 3.2. Performance evaluation on how these three schemes work with the rest of the system can be found in Section 6.2.

3.3.1. Evaluation Using the Correctness Measure. For a loop invocation, if T_{pSEQ} and T_{SEQ} are both greater or both smaller than T_{TLS} , the prediction is considered correct, otherwise it is incorrect. We categorize loops into three groups: (i) loops whose parallel and sequential performance differ by less than 5%, the rest of the loops that are (ii) correctly predicted, and (iii) incorrectly predicted. The first loop category is excluded in the correctness measure since whether or not these loops are parallelized contributes little to the final performance.

Figure 5(a) shows the degree of correctness for all benchmarks. Overall, the correctness measure improves as the proposed execution stall scaling and cache miss cycle classification are incorporated. With execution stall scaling, the correctness is significantly improved in MESA, PERLBK and VPR-R. With cache miss cycle classification, the correctness is further improved, especially in BZIP2, CRAFTY, GZIP, MESA, PARSER and PERLBK. There are two exceptions (AMMP and VORTEX), however, that do not follow this trend of improvement, and Base+ExeStall+dCache yields noticeably lower correctness than Base. The reason is that Base in these two benchmarks heavily biases towards one side of prediction that *happens* to match the actual situation. The extensions to the prediction remove such biases but result in lower degree of correctness. Nevertheless, Base+ExeStall+dCache always yields the highest similarity in these two benchmarks (next subsection), showing its effectiveness. Overall, Base+ExeStall+dCache correctly predicts 93.4% of the loops, a 9.7% improvement compared to the Base scheme.

Figure 5(b) shows the dynamic coverage in execution time of the three loop categories: Close represents loops whose performance differs by less than 5% after being parallelized; Correct and Wrong stand for the rest of the loops that is correctly and incorrectly predicted by the last scheme Base+ExeStall+dCache, respectively. Note that the sum of the three categories shows the percentage of speculative parallel execution in each benchmark. Our prediction scheme generally works well: on average, only 2.8% of total execution time suffers incorrect prediction.

3.3.2. Evaluation Using the Similarity Measure. The similarity measure quantitatively evaluates the accuracy of our prediction for *all* loops parallelized. We first define *dissimilarity* as the accumulative difference between T_{pSEQ} and T_{SEQ} over the sum of T_{SEQ} for all loop invocations; that is,

$$dissimilarity = \frac{\sum |T_{pSEQ} - T_{SEQ}|}{\sum T_{SEQ}}. \quad (2)$$

Similarity is defined as $1 - dissimilarity$. The results are shown in Figure 5(c). Generally, similarity improves with the incorporation of execution stall scaling and

(a) Correctness measure: dynamic coverage of parallelized loops that are *correctly* predicted for different prediction techniques.

(b) Dynamic coverage: percentage of execution time spent on parallelized loops over the entire benchmark.

(c) Similarity measure: how close our prediction could approximate the actual sequential execution cycle.

Fig. 5. Prediction accuracy evaluation. Bars correspond to the three prediction schemes described in the text.

data cache miss classification. Overall, the improvement is gradual and significant: 78.5% for Base, 81.6% for Base+ExeStall, and 86.5% for Base+ExeStall+dCache.

There are, however, a few exceptions: EQUAKE, GAP, and VPR-R. For these benchmarks, the predicted stall cycles due to cache misses are less than that of the actual sequential

Fig. 6. Comparing the predicted speedup and the actual speedup for individual loop invocations in benchmark EQUAKE.

execution. This is because there are less conflict misses in the parallel execution, due to the increase in effective associativity when threads are spread across multiple cores. Although we attempt to compensate for this effect, our mechanism does not cover all possible situations, as stated in Section 3.2.

We tap into one of these benchmarks EQUAKE to find evidence. Figure 6 is a 2D scatter graph, where each data point corresponds a loop invocation. The y- and x-axis are the predicted speedup and the actual speedup, respectively. In other words, the y- and x-coordinate of a point represent T_{pSEQ}/T_{TLS} and T_{SEQ}/T_{TLS} . Due to the massive number of loop invocations, each point is in fact an accumulation of 3000 consecutive invocations of the same loop. The dotted line $y = x$ corresponds for perfect prediction.

In EQUAKE, almost all points fall above the dotted line, but form a linear relationship with the same slope as the dotted line. This indicates that our prediction is *systematically* optimistic, that is, predicting less cycles for sequential execution, due to the same reason of not compensating for all conflict misses. We have also observed that the L1 cache replacement rate is very high. This is a strong indication that parallel execution could have reduced conflict misses compared to sequential execution. Also note that these benchmarks achieves near perfect prediction in terms of correctness measure, thus similarity measure is only used as tie breakers. Therefore, as long as the prediction is consistently pessimistic or optimistic across all loops, the dynamic dispatching system is able to make proper decisions.

3.4. Summary

We believe that our performance prediction mechanisms are accurate enough for the runtime system to make parallelization decisions. Our experiments have also shown that being able to accurately understand cache behaviors and the effects on execution stall is crucial for determining the performance impact of speculative threads.

4. RUNTIME SUPPORT

To support dynamic thread dispatching, the execution framework needs to be able to adjust the behaviors of speculative threads. In this section, we describe the execution framework that (i) programs hardware performance counters to collect the necessary information and (ii) store the dispatching decisions about where to spawn speculative threads.

Fig. 7. Each core is extended with hardware performance counters to classify cycle stalls.

4.1. Programming Hardware Performance Counters

Our performance prediction relies on cycle breakdowns of speculative parallel execution. Note that the goal is not to build accurate TLS execution cycle breakdowns, but to generate a set of components to help reconstruct the sequential execution cycle.

Obtaining cycle breakdowns in an out-of-order processor is difficult due to the overlap of multiple on-the-fly instructions. Examining the instructions at the head of reorder buffer (ROB) gives us some clues [Mericas 2006] to the causes for stalls. We propose a cycle classification scheme that works on programmable hardware counters that is common on modern processor chips. Figure 7 depicts our use case that aims to obtain the following breakdown components:

- Useful: cycles spent graduating useful instructions, that is, instructions not related to speculative thread management;
- ILPStall: cycles stalled due to lack of instruction-level parallelism;
- InstFetch: cycles stalled due to instruction fetch penalty;
- DataCache: cycles stalled due to data cache misses; and
- nConflict: the number of conflict misses that need to be compensated when predicting sequential execution time.

Note that all TLS-related overheads, including thread idling and squashing, are not attributed to any counter because they would not be present in sequential execution.

4.1.1. Basic cycle classification. The cycle classification scheme is explained in the following. As a simplification, we describe cycle classification for a processor that commits one instruction per cycle. For processors with higher commit width, multiple counters can be incremented in a single cycle, based on the activity of each graduation slot. At *each* cycle, the classifier examines the head of the ROB. If the ROB is empty, the InstFetch counter is incremented by one. At the head of the ROB, if the instruction is unable to graduate, a hardware *accumulator* is incremented by one to keep track of the number of cycles this instruction stalls. When the instruction finally commits, no counter is incremented if this instruction is related to TLS thread management. Otherwise, the Useful counter is incremented by one, and the value in the accumulator is

added to the `DataCache` counter if this is a memory instruction and results in a cache miss, or added to the `ILPStall` counter otherwise. The accumulator is reset whenever an instruction graduates.

This mechanism is similar to the performance monitors in IBM POWER5 [Mericas 2006]. However, the following situations require special handling for TLS execution: (i) aggregating counters across multiple cores; (ii) redistributing counters when a thread is squashed due to speculation failure; and (iii) categorizing cache misses.

4.1.2. Aggregating Counters across Multiple Cores. Speculative parallel threads are distributed across multiple cores, and thus performance counters must be aggregated to obtain the complete cycle breakdown. There is only one nonspeculative thread and only the nonspeculative thread is allowed to commit. At thread commit time, the nonspeculative thread forwards its performance counters to its successor, and makes the successor nonspeculative. The new nonspeculative thread adds the forwarded values to its own corresponding counters. Therefore, when a speculatively parallelized region completes, the counters on the core that commits the last thread contain the complete breakdown.

4.1.3. Handling Thread Spawning and Squashing. When a speculative thread is spawned to a processing core, all counters on that core are reset. When a speculative thread is squashed, all counters except `DataCache` and `nConflict` on that core are reset. Note that we do not attempt to record the cycles wasted due to squashing, but need to preserve these two counters for the reasons explained in Figure 3(a).

4.1.4. Cache Miss Categorization. When a load instruction that stalls due to a cache miss finally commits, Section 3.2 describes four scenarios each requiring a different classification mechanism based on information returned from the cache.

Upon a first level data (L1D) cache miss, we must check whether the cache access has found an invalidated cache line with a matching tag. If so, it indicates that this cache line has been loaded into L1D, but then speculatively modified by a thread that is eventually squashed. At this point, the original data must be reloaded from the next level cache. This cache miss would not have occurred if this program is executed sequentially (Figure 3.2). Thus, the stall cycles in the accumulator are discarded.

If a cache miss is served by the L1D cache of a different core (feedback from underlying cache coherence protocol), it is possible that this cache miss would not have occurred in sequential execution, as illustrated in Figure 3.2. This cache miss behaves like a coherence miss. Since coherence misses cannot occur during sequential execution, the stall cycles in the accumulator is also discarded.

Figure 4(b) depicts a different scenario, where a cache line brought in by an earlier load is evicted due to replacement. Rereferencing to this cache line will cause a cache miss, and it is possible for this cache miss to be served by a neighboring L1D cache. In this case, the same coherence messages will be exchanged between the caches as before; however, this cache miss is likely to incur a conflict miss in sequential execution. In other words, conflict misses in sequential execution can behave similarly to coherence misses in TLS execution. However, stall cycles that correspond to coherent misses are always discarded for predicting sequential execution time. Thus, a new mechanism must be introduced to compensate this effect. We propose to augment each L1D cache line with one bit and introduce a new counter to count the number of occurrences of such cache misses in each core. We refer this bit as *ExtraShared* and the counter as `nConflict`. The scheme works as follows: (i) *ExtraShared* is initially set 0 for all cache lines when TLS execution starts; (ii) when a cache line is fetched to serve a cache miss from another cache, set the *ExtraShared* bit for that cache line; (iii) when a cache line is replaced, increment the `nConflict` counter if the *ExtraShared* bit is set, and reset

ExtraShared bit; (iv) when the nonspeculative thread commits, the value of `nConflict` is forwarded and accumulated as other cycle counters.

Essentially, the `nConflict` counter predicts how many additional conflict misses might have occurred in sequential execution. During performance evaluation, we compensate the predicted sequential execution cycle with the value in `nConflict` multiplying the second level cache access latency.

Note that this scheme cannot catch all conflict misses that occur in sequential execution but not in speculative parallel execution, as observed in Section 3.3.2. More sophisticated approaches can potentially alleviate this situation, such as replacing the *ExtraShared* bit with a counter in each cache line. However, our simple scheme can provide accurate prediction in most situations, thus we did not pursue alternatives with higher hardware costs.

4.2. Maintaining the Decision Table

The hardware-based decision table is built in each processor core. It is a content-addressable memory (CAM) indexed by a unique identification number associated with each candidate spawning point, namely its instruction address. Each table entry contains two fields: a *saturation counter*, which is incremented if the TLS execution outperforms the predicted sequential execution and decremented otherwise, and a *performance summary*, which contains the cumulative difference in execution time (i.e., cycles) between the TLS execution and the predicted sequential execution. Note that the performance summary can be a negative value.

Before a candidate spawning point is executed, this table is consulted for whether to grant or decline this spawning request. A decline would fail the spawning instruction and serialize the requesting candidate, given precedence to other overlapping candidate threads. When speculative threads complete execution, the processor that commits the last thread has the aggregated information from all other processors and updates the corresponding table entry: (i) increase or decrease the saturation counter, and (ii) add the difference between the TLS execution and the predicted sequential execution cycle to the performance summary. The table update is then broadcast to other processors. This operation is infrequent; it only happens when the entire invocation of speculative threads finishes.

Essentially, the two fields in a table entry store both the qualitative and the quantitative evaluations for each candidate spawning point. They are used by the dynamic dispatching policies described next.

5. DYNAMIC DISPATCHING POLICIES

Our TLS system focuses on parallelizing loops from sequential programs. Many such programs contain multiple nested loops, and thus the dynamic dispatching policy is required not only to identify and parallelize loops that can benefit from TLS, but also to select the right level of loop to maximize the overall performance gain. A straightforward mechanism is to first tentatively parallelize each loop, measure the performance impact, and then serialize the ones for which TLS execution is ineffective. However, there can be various ways to determine the order in which loops in a loop nest are evaluated and to decide the precedence among different loop levels. In this section, we first examine the design issues for the dynamic dispatching system and then build the most effective policy in incremental steps.

5.1. Basic Design Issues

An effective policy should first identify loops that lead to overall speedup. A loop can have different execution times in different invocations across the program. One example is a loop traversing a linked list: the execution time of the loop is dependent on

the length of the linked list, which may vary from one list to another. Another example is a loop with conditional break-out statements. For such loops, performance estimation based on one invocation could be misleading. Consider a loop with 6 invocations, for example; the first one takes a long time to execute and TLS is 2000 cycles faster than sequential execution, while the other 5 invocations have short execution times, and TLS is 100 cycles slower. With the 6 invocations, TLS leads to an overall speedup of $2000 - 5 * 100 = 1500$ cycles. However, if deprived of cycle numbers, this loop could be considered a poor candidate because for 5 out of 6 invocations, TLS yields lower performance. This could lead to *premature serialization* of this profitable loop level. Identifying loops that could lead to overall speedup requires quantitative evaluation of the impact of speculative threads.

Furthermore, an effective mechanism should identify a set of loops that lead to maximum performance benefits. In many programs, it is common to have multiple nesting loop levels that all benefit from TLS. Since the optimal parallel loops are most likely neither the outermost nor innermost loop, finding the right set of loops is essential to maximize performance.

Finally, an effective mechanism should adapt to program phase changes. When a program enters a different phase, loop behaviors can change substantially. Loops that are previously serialized could potentially benefit from TLS in the new phase. Therefore, it may be necessary to reevaluate and reselect the best performing loops when phase changes. We describe our dispatching mechanisms in four incremental policies, each extending the previous one with higher complexity.

5.2. Searching Loop Levels from Inside Out

The first policy follows an *inside-out* search order, that is, from the innermost to the outermost, to evaluate the impact of TLS for each level in a loop nest. Each level runs in TLS mode for several invocations, and the number of times TLS performs worse than sequential execution is recorded in the saturation counter. Similar to branch prediction, once this counter exceeds a certain threshold, the loop is predicted as not suitable for TLS. This decision is stored in the decision table, so this loop will be serialized whenever it is encountered again, yielding resources to its outer loop levels. Otherwise, the current loop is continuously parallelized, and the search for the loop nest may stop at the current level. We name it *InsideOut* since it is the fundamental search order for other policies.

Although this policy could use the opposite *outside-in* search order instead, outer loops are much larger than inner loops and attempting outer loops would significantly prolong the time needed to reach the best level. In extreme cases, the outermost loop covers the entire program execution; by the time the outermost loop level is evaluated, the execution of the program is almost done.

5.3. Using Quantitative Evaluation

InsideOut is prone to serializing a loop prematurely, while the loop could lead to overall speedup. Loop invocations with different execution times should not be treated equally. We devise a new policy named *Quantitative* that quantitatively evaluates every loop invocation and uses the performance summary, that is, cycles saved from sequential execution, as the weights of different invocations. Under this policy, a loop is serialized if both the saturation counter exceeds the threshold *and* the performance summary becomes negative.

5.4. Incorporating Static Analysis through Compiler Annotations

Both InsideOut and Quantitative stop searching once a loop level that could benefit from TLS is reached. Since multiple loop levels can all benefit from TLS, previous

Table II. Summary of the Performance Dispatching Policies

Policy	Search Order	Performance Evaluation	Static Analysis
InsideOut	Inside out	Based on invocations	Not used
Quantitative	Inside out	Quantitative	Not used
Quant+Static	Inside out	Quantitative	Use as decision
Quant+StaticHint	Inside out	Quantitative	Use as hint

searching mechanisms cannot guarantee reaching the level that has the highest performance benefit. Although an exhaustive search could find the best level, it takes too much time and is not cost-effective, as search time is part of runtime. The new policy incorporates static analysis through compiler annotations to prioritize the search order. It starts from the loop level annotated by the compiler. If this level indeed improves performance, the search is over; otherwise, this loop level is serialized and the search begins from the innermost level. We name it *Quant+Static* as it adds static analysis on top of quantitative evaluation.

5.5. Overriding Static Analysis Decisions

Quant+Static can potentially find the best level if static analysis is accurate. However, it can fall into the same sub-optimality if static analysis targets to the wrong loop level. Our final policy strives to protect against the situation where the compiler's choice is wrong. It explores both the compiler-annotated loop level and the neighboring levels and compares them quantitatively to increase the chance of reaching the best loop level. It is named *Quant+StaticHint* since we treat static analysis only as hints and could override them.

The full adaptation to phase change is beyond the scope of this dissertation. In our implementation, a simple mechanism is used for all the policies: the decision tables are reset periodically, so that the impact of speculative threads can be reevaluated. Table II summarizes the features of the four dispatching policies.

6. PERFORMANCE EVALUATION

In this section, we will first compare the performance among the four dispatching policies (Section 5), then evaluate how the different performance prediction schemes (Section 3) impact the performance of the dispatching system. Finally, we will contrast our dynamic dispatching mechanism with the state-of-the-art static thread management.

6.1. Impact of Dynamic Dispatching Policies

Figure 8 compares the speedup with respect to sequential execution among the four increasingly sophisticated dispatching policies. In addition, the scheme that blindly parallelizes the innermost loop level (bar Innermost) is also included in the comparison. All speedup numbers are summarized by geometric mean (G.M.).

6.1.1. Innermost vs. InsideOut. Comparison between the first two bars indicates that InsideOut generally outperforms Innermost, however with the exception of GZIP, MCF, and VPR-R. A close examination of the execution traces reveals that InsideOut missed some profitable inner loop levels in these benchmarks due to premature serialization (Section 5.1), whereas Innermost persistently parallelizes the innermost loop levels regardless of their performance. However, for many other benchmarks, our first dispatching policy greatly outperforms Innermost by a large margin, especially in AMMP, ART, MESA, and VPR-P, showing its usefulness. InsideOut achieves an overall speedup of 1.216x, while Innermost achieves 1.146x.

Fig. 8. Performance comparison of different dynamic dispatching policies. Speedups are normalized to the original sequential execution (SEQ).

6.1.2. Simple vs. Quantitative. Quantitative weights different loop invocations by the cycles saved from sequential execution, and is more accurate than InsideOut in identifying profitable loop levels. It resolves the problem of premature serialization, as the performance is improved in GZIP, MCF, and VPR-R.

Yet ART and MESA experienced significant performance downgrade in this new policy. This is due to the existence of multiple overlapping loop levels that are all profitable. Quantitative stops searching once a profitable level is found, without checking whether an even profitable one lies outside. We refer this problem as *local optimality*. InsideOut and Quantitative both suffer from it in a number of benchmarks. For the cases in ART and MESA, however, InsideOut prematurely serializes the inner loop level by mistake and reaches a more profitable outer level, making it outperform Quantitative. Quantitative has an overall speedup of 1.232x, slightly better than InsideOut.

6.1.3. Quantitative vs. Quant+Static. The Quant+Static policy incorporates compiler annotations to prioritize the search in the loop nest. We found that it selects better loop levels for benchmarks ART, BZIP2, and TWOLF, but it greatly degrades AMMP, GCC, GZIP, and MCF. And the performance downgrade in MESA remains. The problem is that compiler annotation often fails to point to the most profitable loop levels. Quant+Static respects the compiler decision and does not attempt to look at other levels as long as the annotated loop level does not worsen performance. In another word, the problem of local optimality is still at large. Overall, this policy shows 1.224x speedup on average.

6.1.4. Quant+Static vs. Quant+StaticHint. The Quant+StaticHint policy treats compiler annotation only as hints and evaluates both the annotated loop level and its neighboring levels. The compiler-annotated loop will be compared with its inner loops. If an annotated loop level is the innermost, it will be compared with the immediate outer loop level. In both cases, this policy can select a loop level that outperforms the compiler's decision and try to avoid local optimality.

With the Quant+StaticHint policy, almost all benchmarks benefit. Inaccurate static loop selections are overridden in GZIP and MCF, so the performance becomes similar to Quantitative. At the same time, TWOLF enjoys the benefit of accurate compiler annotations. More importantly, for AMMP, ART, GAP, GCC, MESA, and VPR-R, loop levels with higher performance are discovered and parallelized, which leads to better results than both

Fig. 9. Performance comparison of different prediction schemes. Speedups are normalized to the original sequential execution (SEQ). Note that bar Base+ExeStall+dCache is also the last bar in Figure 8.

Quantitative and Quant+Static. In Section 6.4, we will look into a few benchmarks as case studies to illustrate how this dispatching policy searches the loop nest and to offer insights into why it outperforms static analysis.

Across all the benchmarks, Quant+StaticHint generally yields the best performance among all dispatching policies, while negligible overhead is observed in some benchmarks due to tentatively trying out nonoptimal loop levels for comparison. The average speedup to sequential execution is 1.371x.

6.1.5. Summary. Performance gradually improves as we build up the desired dispatching policy, rising from the most basic that achieves 1.216x to the most sophisticated that achieves 1.371x.

6.2. Impact of Performance Prediction Schemes

Section 3 has proposed three increasingly complex schemes to predict sequential execution and evaluated their accuracy. In this section, we will evaluate their performance impact for the dispatching system. For the purpose of this comparison, we will use the most successful dispatching policy Quant+StaticHint.

Figure 9 contrasts the performance of these prediction schemes. Base corresponds to the baseline prediction scheme described in Section 3; Base+ExeStall incorporates the execution stall scaling described in Section 3.1; and Base+ExeStall+dCache further incorporates the data cache behavior classification described in Section 3.2.

A general trend of increasing performance is observed as the prediction scheme is amended to model TLS execution in more details. Base yields the lowest performance because the inaccuracy in prediction causes the runtime system to often target on wrong loops. Execution stall scaling and data cache behavior classification prove to be useful in correcting this inaccuracy. For example, in TWOLF incorporating execution stall scaling recovers most of the performance loss; while in ART, data cache behavior classification is key to prediction accuracy; nevertheless, GCC benefits from both amendments. MCF is a memory-bound benchmark. Only applying execution stall scaling happens to expose more inaccuracy; however, further incorporating data cache behavior classification ultimately corrects this error.

Benchmark VPR-R exhibits some abnormality as incorporating data cache behavior classification results in lower performance. This is due to a simplification when

predicting conflict misses in sequential execution. We will discuss this problem and its possible remedy in further detail in Section 6.3.

Overall, Base and Base+ExeStall show the speedup of 1.265x and 1.290x, a 7.7% and 5.9% loss compared to Base+ExeStall+dCache (1.371x), respectively.

6.3. Performance Comparison with the Static Approach

This section contrasts our most successful dispatching policy (Quant+StaticHint) with the state-of-the-art static analysis. Before presenting a detailed comparison, we first explain the overhead introduced to TLS. To parallelize a loop, necessary special instructions (thread spawning, synchronization instructions such as signal and wait, and thread committing) are generated. These extra instructions are TLS overheads. For static approach, the compiler selects only a subset of loops to insert these instructions. But for dynamic approach, all the loops are instrumented so that any loop may be eligible for parallel execution. When the code is running in sequential mode, TLS-specific instructions are executed as NOPs, but still incur extra performance penalty. If a program has a large number of small loops that are instrumented but eventually serialized, the slowdown due to this overhead can be significant. We referred to this as *parallel code overhead* for TLS.

One way to completely eliminate this overhead is to generate two versions of code for each loop, a parallel version and a sequential version, and switch between these versions as needed. However, we are unable to experiment with this scheme due to limitations in our infrastructure. To estimate the performance after eliminating this overhead, we attempt to normalize the TLS runs to the sequential execution of their respective parallelized code.

Parallel code overhead can also be mitigated in several ways. Using simple heuristics, the compiler or programmer can filter out some loops first. For example, loops with tight dependencies (such as pointer chasing or short reduction) are unlikely to benefit from TLS. A runtime reoptimization system can also eliminate such overhead for all the loops not selected for TLS execution. We do not explore these optimizations and leave their integration as future work.

In Figure 10, Static and Dynamic are both normalized to the execution time of the original sequential executable where no instrumentation is made. Over all the benchmarks, dynamic dispatching (1.371x) outperforms static analysis (1.253x) by 9.4%.

There are a number of cases where dynamic dispatching is worse compared to static analysis, such as in BZIP2, CRAFTY, PERLBMK, TWOLF, VORTEX, and VPR-R. We find that it is the parallel code overhead that causes most of the performance downgrade. We run the static and dynamic executables sequentially, and their execution time is shown in Figure 11(a) as bars StaticSEQ and DynamicSEQ, respectively. The higher the bar is, the greater the overhead is. In Figure 11(b), Static/StaticSEQ and Dynamic/DynamicSEQ are normalized to their corresponding sequential baselines. In this rescaled comparison, most previously under-performing benchmarks has shown comparable or even better performance under dynamic dispatching. Overall, dynamic dispatching (1.475x) could potentially outperform static analysis (1.271x) by 16.0%.

After the rescaling, VPR-R is the only benchmark in which dynamic dispatching performs noticeably worse than static analysis. The reason is due to inaccurate performance prediction discussed in Section 3.3.2. For a profitable loop in VPR-R, our model does not predict the full extent of conflict misses that would have occurred in sequential execution. Therefore, sequential execution cycle is predicted less than what it should be, and also less than the TLS execution cycle, leading to an incorrect conclusion that this loop does not benefit from TLS and consequently, this profitable loop is serialized. Static approach benefits from parallelizing this loop. We have observed significant count of conflict cache misses when this loop is serialized and run sequentially. This

Fig. 10. Performance comparison against the state-of-the-art static approach. Speedups are normalized to the original sequential execution (SEQ). Note that bar Dynamic is also the last bar in Figure 8.

(a) Execution time of the parallelized binary running sequentially. All instrumented loops will incur overhead. The higher the bar, the worse the overhead.

(b) Performance comparison between static analysis and dynamic dispatching when normalized to their respective sequential versions.

Fig. 11. Understanding the impact of parallel code overhead.

```

562:  for( inode = 0; inode < nx*ny*nz; inode ++ )
    /* loop over all mm nodes */
    ...
995:  for( i=0; i< imax; i++)
    {
        a2 = (*atomall)[i];
998:  for( j=0; j< a1->dontuse; j++)
        { if( a2 == a1->excluded[j]) goto SKIPNEW; }
        ...
SKIPNEW: j = j;
    }
}

```

(a) AMMP code snippet in rectmm.c

(b) Performance characteristics of loops inside the snippet

Fig. 12. Source code and execution time breakdown for loops in AMMP. The bars are labeled with loop name and its execution mode.

observation could lead to a potential remedy to this problem: if a serialized loop incurs high rate of conflict misses, we should give it another chance to be parallelized, taking into account the benefit of parallel execution to reduce conflict misses from sequential execution.

6.4. Case Studies

In this section, we examine in more detail on benchmarks where dynamic approach performs much better than static approach and explain why this can only be achieved through runtime decision making. Many cases have similar behaviors, so we highlight on the cases for AMMP, ART and MESA in the next Sections.

6.4.1. AMMP. The performance improvement of AMMP from Static to Dynamic in Figure 10 comes from the different loop levels selected by them. One instance of such difference is located in source file rectmm.c. The outer loop starts at line 562 and the inner loop starts at line 995. We named loops by their starting line number. Their code snippets and execution time breakdowns are shown in Figure 12(a) and Figure 12(b). The bars are labeled with loop name and its execution mode. For example, 995_SEQ and 995_TLS4 correspond to loop 995 running sequentially and in parallel, respectively. Bars are normalized to sequential executions with respect to the same loop. The static analysis believes the outer loop 562 has a greater performance benefit than the inner loop 995. However, loop 562 incurs frequent speculation failures (i.e., is squashed), which cannot be predicted by the compiler. Although the inner loop 995 would also incur speculation failure if selected for TLS, it achieves a better speedup than parallelizing loop 562.

```

584: for (ti=0;ti<numfls;ti++) {
      tsum = 0;
      ttemp = fl_layer[ti].P;
589:   for (tj=0;tj<numf2s;tj++) {
      if ((tj == winner)&&(Y[tj].y > 0))
          tsum += tds[ti][tj] * d;
      }
      fl_layer[ti].P = fl_layer[ti].U + tsum;
      tnorm += fl_layer[ti].P * fl_layer[ti].P;
      if (ttemp != fl_layer[ti].P)
          tresult=0;
  }

```

(a) ART code snippet in scanner.c

(b) Performance characteristics of loops inside the snippet

Fig. 13. Source code and execution time breakdown for loops in ART. The bars are labeled with loop name and its execution mode.

Our dynamic dispatching policy Quant+StaticHint uses cycle-saving as the measurement. In this case, it works as follows: since the outer loop 562 is selected by compiler, Quant+StaticHint policy first parallelizes inner loop levels within loop 562 at its first invocation, and innermost level loop 998 is parallelized for comparison. Since loop 998 degrades performance, it is quickly serialized. In the next step, loop 995 is parallelized and the cycle-saving is recorded. At the second invocation of loop 562, this compiler-selected loop is parallelized while all of its inner levels are tentatively serialized and the cycle-saving is also recorded. From the third invocation on, loop 562 and loop 995 are compared and the one with greater cycle-saving is selected for TLS execution. In this case, loop 995 wins the competition.

6.4.2. ART. Differences in the performance of ART, as shown in Figure 10, partly come from source code in scanner.c: static analysis chooses the inner loop starting at line 589, whereas dynamic mechanism favors the outer loop starting at line 584. Figure 13(a) and Figure 13(b) show the code snippets of these two loops and contrast their performance. Bars are labeled as in the previous section.

Judged by the execution time breakdown of 584.TLS4 alone, this loop should not be parallelized since the cost of speculation failure (the Squash segment) is high. However, when compared against the sequential execution 584_SEQ, the failed speculative threads help to fetch useful data into the L2 data cache and reduce the data cache stalling (dCache) to a large extent. While this cache effect (discussed in Section 3.2) benefits parallel execution, its impact is hard to accurately estimate at compile time. Our compiler uses dependence profiles to estimate speculation failures, so loop 584 is determined not ideal for TLS due to possible speculation failures from runtime aliasing.


```

897: for (i=2;i<VB->Count;i++) { /* vrender.c */
    START_PROFILE
    (*ctx->Driver.TriangleFunc)( ctx, i-2, i-1, i, i );
    END_PROFILE( ctx->PolygonTime, ctx->PolygonCount, 1 )
}

471: static void general_textured_triangle /* triangle.c */
( GLcontext *ctx, GLuint v0, GLuint v1, GLuint v2, GLuint pv )
{
    ...
730: while (lines>0) { /* tritemp.h */
    #if INTERP_Z
        GLfixed ffz = fz;
        /*GLdepth *zp = zRow;*/
    #endif
    #if INTERP_RGB
        GLfixed ffr = fr, ffg = fg, ffb = fb;
    #endif
    ...
}
    ...
}

```

(a) MESA code snippet in vrender.c, triangle.c and tritemp.h

(b) Performance characteristics of loops inside the snippet

Fig. 14. Source code and execution time breakdown for loops in MESA. The bars are labeled with loop name and its execution mode.

This is why the static analysis chooses the inner loop 589 for TLS. Unfortunately, the performance of inner loop 589 is not up to the expectation due to the smaller coverage and *insufficient thread count* (part of Others in the breakdown). Insufficient thread count means the total number of thread is less than the number of available processing cores (four in our experiment), so some cores are left idling when this loop is being executed. This contributes most of the Others segment in the breakdown. Dynamic dispatching policy is able to compare the compiler-selected inner loop 589 with the outer loop 584 and ends up selecting loop 584 for better performance gain.

6.4.3. MESA. The performance difference between static and dynamic approach mainly comes from the code snippet shown in Figure 14(a). In vrender.c, an outer loop at line 897 calls a function that contains an inner loop at tritemp.h:730. The execution breakdowns of these two loops are shown in Figure 14(b).

Static analysis chooses the inner loop 730, which turned out to have a low thread count per loop invocation. Thus, TLS execution is unable to utilize all the cores efficiently. It is shown as a significant Others segment in the breakdown. This also

correlates with the loop coverage in Figure 5(b) that most loops selected by the static approach perform no better than 5% compared to their sequential execution. Dynamic dispatching approach attempted to parallelize both loops, and found it more beneficial to parallelize the outer loop. We noticed that dCache segment increases from sequential to parallel execution for loop 897. This is because parallel execution needs to load the same data item to multiple L1D caches while sequential execution for the same code segment may need to load only once. Nevertheless, this minor overhead does not overcome the benefit of speculative parallelization in this case.

7. RELATED WORK

This research used hardware-performance-counter based techniques to help determine where to create parallel speculative threads. In this section, we will discuss related work in two areas of research: dynamic optimization and determining where to parallelize.

Dynamically detecting performance bottlenecks for program optimization has been demonstrated to be effective [Luk et al. 2005; Bruening et al. 2003; Lu et al. 2003; Sullivan et al. 2003; Lu et al. 2005; Lu et al. 2004] for performance enhancing. For example, DynamoRIO [Bruening et al. 2003; Sullivan et al. 2003] uses a combination of a native *Just-In-Time* compiler and partial evaluation techniques. On the other hand, Lu et al. [2004, 2005] generate helper thread prefetches using information obtained from the hardware monitors on the Sun UltraSPARC®. The optimization framework proposed in ADORE [Lu et al. 2004, 2005] is similar to the speculative thread optimization framework proposed in this article, with the following differences: (i) our work uses hardware-based performance counters that generate cycle breakdowns [Eyerman et al. 2006; Mericas 2006], while ADORE uses event-based hardware performance counters; (ii) in ADORE, a dynamic compiler is responsible for generating and patching reoptimized code at runtime, while our scheme does not require dynamic code generation; and (iii) we carefully evaluate the performance impact of speculative threads before optimization, while ADORE does not evaluate the effectiveness of the prefetching threads.

There is also a large body of previous work on runtime performance optimization for parallel applications (such as OpenMP [Dagum and Menon 1998]) [Suleman et al. 2008; Burcea 2005; Lee et al. 2008; Zhang et al. 2005]. stOMP [Burcea 2005] selects among multiple specialized versions of parallel regions based on parameters; Zhang et al. [2005] experiment with different OpenMP scheduler configurations at different parallel regions. Lee et al. [2008] peel parallel loops and collect performance profiles using the first few iterations of the loop to reoptimize the program dynamically. However, performance optimization for OpenMP is very different from that for speculative threads. First, the optimization goals are different: for OpenMP system, the tuning knobs are the number of threads [Suleman et al. 2008], shared variables [Burcea 2005], etc., while our system optimizes TLS performance by redeciding where to speculate. Second, the performance models for OpenMP and TLS differ significantly: in OpenMP, all threads perform useful work, while in TLS, work done by speculative threads can be wasted when speculation fails. To summarize, the knowledge learned from OpenMP performance optimization cannot be directly applied to TLS. However, we believe that our performance analysis technique (cache behavior classification, execution stall scaling, etc.) and dynamic dispatching policy can be applied to OpenMP threads to understand their performance and select which loop level to parallelize.

In the context of TLS, one key issue is deciding how to extract parallel threads. Existing work fall into two categories: compiler-based [Sohi et al. 1995; Vijaykumar and Sohi 1998; Marcuello and González 2002; Du et al. 2004; Johnson et al. 2004; Quiñones et al. 2005; Wang et al. 2005; Liu et al. 2006; Johnson et al. 2007] and hardware-based [Tubella and González 1998; Marcuello et al. 1998]. Among compiler-based approaches,

the POSH [Liu et al. 2006] TLS compiler partitions the program into tasks based on code structures like loops and subroutines. It uses a simple profiling pass for weeding out the ineffective tasks. The Mitosis compiler [Quinones et al. 2005] inserts spawning pairs to transform a sequential program into a number of speculative threads. The selection of spawning pairs is also dependent on static analysis and profiling information. Wang et al. [2005] and Du et al. [2004] use extensive profiling information to statically estimate loop performance and select a set of loops for parallelization to maximize overall program performance. Johnson et al. [2004] propose the balanced min-cut approach to decompose the program at compile time. They use a graph-theoretic framework to try and take into account overheads of data dependence, load imbalance, and thread prediction. Vijaykumar and Sohi [1998] deal with the issue of selecting the right tasks for speculation in the Multiscalar architecture [Sohi et al. 1995] by relying on compiler heuristics. These heuristics choose tasks based on characteristics like task-size, inter-task control flow, and data dependence. These compiler-based approaches, by virtue of being static, cannot predict program behavior accurately. Program behavior depends on numerous factors like micro-architecture-specific features, inputs applied, and memory access patterns, none of which can be projected beforehand. Also, some of them require detailed profiling to get better accuracy, which can be very expensive and cannot adapt to different input sets. Johnson et al. [2007] also propose using a compiler to instrument a profile run to search through candidate threads and pick the best threads as the profile run executes. The key difference between these profile-directed compiler-based thread partition techniques and our proposed technique is that the former searches for speculative threads using profile information or through profile runs and then compile the choices into the binary, whereas the latter does the search at runtime. Thus, only the proposed technique can adapt to performance characteristics variations associated with different input sets and phase changes.

Hardware-based approaches often extract parallelism dynamically with appropriate hardware support. Marcuello et al. [1998] and Tubella et al. [1998] propose hardware to dynamically detect loops and gather characterization information. This information is then used to speculatively parallelize threads from a program. However, hardware cannot benefit from high-level information such as program structure. Therefore, hardware-based mechanisms tend to parallelize the inner loops. Thus, our hybrid approach, where the compiler instruments the application with performance annotations and the hardware makes the final decision, has been proposed.

Renau et al. [2005, 2005] use the frequency of squashes and resource availability to determine whether to stall a thread from respawning, a simple metric that is effective in preventing the system from useless respawning. But in our case, we need to determine whether speculation improves performance compared to sequential execution, and more importantly, how much the improvement is. Therefore, more detailed information needs to be collected and more sophisticated evaluation is proposed.

Kim et al. [2010] proposed a software runtime system that enables speculative parallelization techniques on nonshared memory clusters. The large speedup reported (49x on average) is primarily due to the large number of available cores (128 in total) and the use of speculation techniques that efficiently address internode communication costs. In contrast, our research is built on a TLS-enabled 4-core CMP system with cache-coherent shared memory. Our focus is not how to efficiently support TLS on the given hardware, but to select better speculation candidates through dynamic monitoring and dispatching.

8. CONCLUSIONS AND FUTURE WORK

Execution models that spawn assistant threads to enhance the performance of single-threaded applications are emerging, targeting multithreaded architectures.

Their complex runtime behaviors greatly increase the difficulty of estimating their performance impact using existing static analysis. Managing threads at runtime is a natural alternative. This article presents an execution framework that monitors and evaluates the performance of speculative threads under the context of TLS, and then adjusts their creation accordingly. We utilize hardware performance monitors to generate an accurate execution cycle breakdown for TLS and dynamically analyze the cycle breakdown to determine the efficiency of TLS. The proposed analyses are capable of correctly determining whether TLS is able to improve the performance for loops that correspond to over 93% of total loop execution time across all benchmarks. This article also proposes, implements and evaluates various dynamic dispatching policies to adjust the exploitation of speculative threads according to their performance profile. The best policy achieves an overall speedup of 1.37X compared to sequential execution and outperforms static thread management by 9.4%.

Therefore, our dynamic optimization system proves to be effective in exploiting the performance potential of TLS and enhancing their capabilities. The performance counters that are used in this article are available on some high-performance processors, and the additional hardware overhead is moderate. With proposed hardware performance monitoring capability and a set of dynamic dispatching policies, TLS may reach a performance level not previously attainable, making it more attractive for multicore architectures. The future work is discussed in the following.

8.1. Dynamic Optimization of Other Assistant Threads

Numerous execution models involving the creation of *assistant threads* have been proposed to utilize the emerging multicore to satisfy diverse performance or nonperformance requirements. For many such threads, their execution does not necessarily affect the correctness of the application, but introduces significant performance variations and resource competition. Thus, judicious utilization of these threads is key to application performance. Such threads not only include performance-enhancing speculative threads and helper threads [Kim and Yeung 2002; Collins et al. 2001; Luk 2001; Purser et al. 2000; Sundaramoorthy et al. 2000; Lu et al. 2003; Lu et al. 2005], and other workload sharing parallel threads [Suleman et al. 2008; Burcea 2005; Lee et al. 2008], but also monitoring threads and verification threads that aim to improve nonperformance metrics of a system.

This article focuses on one class of assistant threads that aims to enhance program performance by creating speculative parallel threads. However, the proposed techniques can be deployed to evaluate the performance impact of a large variety of assistant threads and make decisions on how to best deploy them. Let us take helper threads as an example. A helper thread improves the performance of an application by bringing data into the shared cache before they are needed. However, if deployed improperly, prefetching threads can also degrade application performance by polluting the cache or saturating shared resources, such as the off-chip pin bandwidth or the bus, to create a performance bottleneck. We can apply the following steps to dynamically optimize the performance of a helper thread: (i) executing the program with a helper thread and configuring the hardware performance monitors to dynamically collect a performance profile that contains information regarding how prefetched lines are used by the main thread, whether prefetched data displaces useful data, and whether the helper thread contributes to reduced data cache stalls; (ii) isolating the performance impact of the helper thread using the dynamically collected performance profile, estimating the performance of the main thread in the absence of the helper thread using the profile information; and (iii) enabling/disabling or reoptimizing the helper-thread based on its performance impact.

8.2. Adaptation to Phase Changes

Programs exhibit phase behaviors. During different phases of execution, speculative threads may behave differently. Thus, the dynamic optimization system must adapt to these changes and potentially reoptimize the assistant threads accordingly. In this research, a simple mechanism is used to adapt to phase changes. The performance and decision tables are reset periodically, and the performance impact of speculative threads is reevaluated. This mechanism can potentially introduce unnecessary overhead if the performance characteristics of the speculative threads remain unchanged. Many researchers have been working on detecting phase changes. Our system can adopt such phase change detection mechanisms so that speculation effectiveness is reevaluated only when a phase change is observed.

REFERENCES

- AKKARY, H. AND DRISCOLL, M. A. 1998. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 226–236.
- BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed And Runtime Optimization (CGO '03)*. IEEE, 265–275.
- BURCEA, M. 2005. stOMP: A specializing thread library for OpenMP. M.S. thesis, University of Toronto.
- CINTRA, M. AND TORRELLAS, J. 2002. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*. IEEE, 43.
- COLLINS, J. D., TULLSEN, D. M., WANG, H., AND SHEN, J. P. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 306–317.
- DAGUM, L. AND MENON, R. 1998. OpenMP: An industry-standard API for shared-memory programming. *Comput. Sci. Engin.* 5, 46–55.
- DU, Z.-H., LIM, C.-C., LI, X.-F., YANG, C., ZHAO, Q., AND NGAI, T.-F. 2004. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, 71–81.
- DUBEY, P. K., O'BRIEN, K., O'BRIEN, K. M., AND BARTON, C. 1995. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT '95)*. IFIP Working Group on Algol, Manchester, UK, 109–121.
- EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 175–184.
- FRANKLIN, M. AND SOHI, G. S. 1992. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. ACM, New York, NY, 58–67.
- GUPTA, M. AND NIM, R. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing '98)*. IEEE, 1–12.
- HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 58–69.
- JOHNSON, T. A., EIGENMANN, R., AND VIJAYKUMAR, T. N. 2004. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, 59–70.
- JOHNSON, T. A., EIGENMANN, R., AND VIJAYKUMAR, T. N. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, NY, 205–214.
- KIM, D. AND YEUNG, D. 2002. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 159–170.

- KIM, H., RAMAN, A., LIU, F., LEE, J. W., AND AUGUST, D. I. 2010. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 3–14.
- KNIGHT, T. 1986. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, 105–112.
- LEE, J. W., NG, M. C., AND ASANOVIC, K. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE, 89–100.
- LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. ACM, New York, NY, 158–167.
- LU, J., CHEN, H., FU, R., HSU, W.-C., OTHMER, B., YEW, P.-C., AND CHEN, D.-Y. 2003. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 180.
- LU, J., CHEN, H., YEW, P.-C., AND CHUNG HSU, W. 2004. Design and implementation of a lightweight dynamic optimization system. *J. Instruct.-Level Parallel*. 6.
- LU, J., DAS, A., HSU, W.-C., NGUYEN, K., AND ABRAHAM, S. G. 2005. Dynamic helper threaded prefetching on the Sun UltraSPARC CMP processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE, 93–104.
- LUK, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. ACM, New York, NY, 40–51.
- LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, 190–200.
- MARCUELLO, P. AND GONZÁLEZ, A. 1999. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99)*. ACM, New York, NY, 365–372.
- MARCUELLO, P. AND GONZÁLEZ, A. 2002. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*. IEEE, 55.
- MARCUELLO, P., GONZÁLEZ, A., AND TUBELLA, J. 1998. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing (ICS '98)*. ACM, New York, NY, 77–84.
- MERICAS, A. 2006. Performance monitoring on the POWER5 microprocessor. In *Performance Evaluation and Benchmarking*, L. K. John and L. Eeckhout, Eds. CRC Press, 247–266.
- NAGPURKAR, P., KRINTZ, C., HIND, M., SWEENEY, P. F., AND RAJAN, V. T. 2006. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE, 111–123.
- OPEN64 DEVELOPERS. 2001. Open64 compiler and tools. <http://www.open64.net>.
- OPLINGER, J. T., HEINE, D. L., AND LAM, M. S. 1999. In search of speculative thread-level parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE, 303.
- PERELMAN, E., POLITO, M., BOUGUET, J.-Y., SAMPSON, J., CALDER, B., AND DULONG, C. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE, 88–88.
- PURSER, Z., SUNDARAMOORTHY, K., AND ROTENBERG, E. 2000. A study of slipstream processors. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, New York, NY, 269–280.
- QUÍÑONES, C. G., MADRILES, C., SÁNCHEZ, J., MARCUELLO, P., GONZÁLEZ, A., AND TULLSEN, D. M. 2005. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, 269–279.
- RENAU, J., STRAUSS, K., CEZE, L., LIU, W., SARANGI, S., TUCK, J., AND TORRELLAS, J. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, 219–228.
- RENAU, J., TUCK, J., LIU, W., CEZE, L., STRAUSS, K., AND TORRELLAS, J. 2005. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, 179–188.

- SIMPLESCALAR LLC. 2004. The SimpleScalar tool set. <http://www.simplescalar.com/>.
- SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 414–425.
- STEFFAN, J. G., COLOHAN, C., ZHAI, A., AND MOWRY, T. C. 2005. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst.* 23, 253–300.
- STEFFAN, J. G., COLOHAN, C., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, 1–12.
- SULEMAN, M. A., QURESHI, M. K., AND PATT, Y. N. 2008. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 277–286.
- SULLIVAN, G. T., BRUENING, D. L., BARON, I., GARNETT, T., AND AMARASINGHE, S. 2003. Dynamic native optimization of interpreters. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*. ACM, New York, NY, 50–57.
- SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBURG, E. 2000. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 257–268.
- TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D. J., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Trans. Comput.* 48, 881–902.
- TUBELLA, J. AND GONZÁLEZ, A. 1998. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. IEEE, 14.
- VIJAYKUMAR, T. N. AND SOHI, G. S. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 81–92.
- WANG, S. 2007. Compiler techniques for thread-level speculation. Ph.D. thesis, University of Minnesota - Twin Cities.
- WANG, S., DAI, X., YELLAJYOSULA, K. S., ZHAI, A., AND CHUNG YEW, P. 2005. Loop selection for thread-level speculation. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*.
- ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2002. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 171–183.
- ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE, 39.
- ZHANG, W., CALDER, B., AND TULLEN, D. M. 2005. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. IEEE, 87–98.

Received August 2010; revised January 2012; accepted May 2012