Performance Scalability of Decoupled Software Pipelining

RAM RANGAN
IBM Austin Research Laboratory
and
NEIL VACHHARAJANI, GUILHERME OTTONI, and DAVID I. AUGUST
Princeton University

Any successful solution to using multicore processors to scale general-purpose program performance will have to contend with rising intercore communication costs while exposing coarse-grained parallelism. Recently proposed pipelined multithreading (PMT) techniques have been demonstrated to have general-purpose applicability and are also able to effectively tolerate intercore latencies through pipelined interthread communication. These desirable properties make PMT techniques strong candidates for program parallelization on current and future multicore processors and understanding their performance characteristics is critical to their deployment. To that end, this paper evaluates the performance scalability of a general-purpose PMT technique called decoupled software pipelining (DSWP) and presents a thorough analysis of the communication bottlenecks that must be overcome for optimal DSWP scalability.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design Studies, Performance Attributes; D.1.2 [**Automatic Programming**]: Program Transformation; D.1.3 [**Concurrent Programming**]: Parallel Programming; C.1.4 [**Parallel Architectures**]

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Decoupled software pipelining, performance analysis

ACM Reference Format:

Rangan, R., Vachharajani, N., Ottoni, G., and August, D. I. 2008. Performance scalability of decoupled software pipelining. ACM. Trans. Architec. Code Optim. 5, 2, Article 8 (August 2008), 25 pages. DOI = 10.1145/1400112.1400113 http://doi.acm.org/10.1145/1400112.1400113

This work has been graciously supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of Intel Corporation.

This work done when Ram Rangan was a graduate student at Princeton University.

Authors' addresses: Ram Rangan, IBM Austin Research Laboratory, 11501 Burnet Road, Austin TX 78758; email: rrangan@us.ibm.com; Neil Vachharajani, Guilherme Ottoni, and David I. August, Department of Computer Science, 35 Olden Street, Princeton, NJ 08540; email: {nvachhar, ottoni, august}@princeton.edu.

1. INTRODUCTION

Multicore processors or chip multiprocessors (CMPs) have become the predominant organization for current microprocessors. Besides greatly simplifying design and verification tasks, such designs overcome the clock speed, power, thermal, and scalability problems plaguing aggressive uniprocessor designs while continuing to provide additional computing power using additional transistors provided by technology scaling. While additional processors on the chip improve the throughput of many independent tasks, they, by themselves, do nothing to improve the performance of individual tasks. Worse still for task performance, processor manufacturers are considering using simpler cores in CMPs to improve performance/power. This trend implies that single task performance will *not* improve and may actually degrade. Thus, performance improvement on a CMP requires programmers or compilers to parallelize individual tasks into multiple threads and expose thread-level parallelism (TLP).

The major challenge in extracting multiple threads from sequential code is handling dependences between threads. Parallelization techniques must insert synchronization between threads to communicate these dependences. Unfortunately, if not handled carefully, synchronization for interthread dependences can eliminate parallelism by serializing execution across threads.

Depending on how they handle interthread dependences, TLP techniques can be categorized into three principle paradigms: independent multithreading (IMT), cyclic multithreading (CMT), and pipelined multithreading (PMT). IMT techniques like DOALL loop parallelization [Lundstorm and Barnes 1980] attempt to parallelize programs into fully independent threads. While they work very well for scientific codes, which do not have difficult-to-break recurrences, they have poor applicability in general-purpose codes. CMT techniques like DOACROSS parallelization [Cytron 1986], on the other hand, can handle codes with recurrences. However, these techniques map recurrences as interthread dependences, thereby exposing them to intercore delays at runtime. As more and more cores are integrated on the same die, increased wire delays and contention in the shared memory subsystem may cause intercore operand communication latencies to vary from few tens of cycles to even few hundreds of cycles, leading to poor CMT performance.

It is in this context that the relatively unheard of DOPIPE parallelization technique [Padua 1979][1] or more generally, PMT, assumes significance. PMT, like CMT, handles codes with recurrences, but differs in that the resulting interthread dependences do not form a cycle, but rather a pipeline (more precisely an arbitrary directed-acyclic graph). By parallelizing codes with recurrences, PMT achieves the wide applicability of CMT, and by avoiding cyclic cross-thread

[1]DOPIPE was originally proposed as a multithreading technique alongside DOACROSS to handle scientific codes with recurrences. Since the loop body of threads produced by DOACROSS remained identical across all threads, it enabled DOACROSS'ed codes to spawn enough threads to match the available processor count. DOPIPE, on the other hand, split the original loop's body among multiple threads and, as a result, the number of threads was essentially fixed at compile time. The runtime scalability requirements of the scientific computing community resulted in DOACROSS becoming the preferred parallelization strategy.

dependences, PMT-parallelized codes can tolerate long interthread communication latencies. Communication latency in PMT programs only affects the pipeline "fill time," which is only a one-time cost.

The recent years have seen a revival of PMT techniques. For example, StreamIt [Thies et al. 2002; Gordon et al. 2002], a language-level technique, provides first-class streaming constructs to enable programmers to write PMT programs. Our early work on a nonspeculative PMT transformation called decoupled software pipelining (DSWP) showed that PMT techniques are very effective in tolerating variable latency stalls [Rangan et al. 2004]. This was demonstrated on hand-parallelized recursive data structure codes. Subsequently, we presented a generic algorithm to automatically apply DSWP to general-purpose codes [Ottoni et al. 2005]. Dai et al. [2005] showed that pipelined threaded execution can greatly improve packet processing performance.

The promise shown by the above techniques and the compelling advantages of PMT make it a strong candidate for use in program parallelization for current and future multicore architectures. Therefore, an understanding of the performance characteristics of PMT programs is crucial to their deployment.

In this paper, we shall focus on DSWP, a nonspeculative automatic PMT transformation. While prior work on DSWP provided insights into understanding its ability to effectively tolerate variable latency stalls [Rangan et al. 2004] and the importance of thread balance to achieving optimal speedup [Ottoni et al. 2005], they restricted their evaluation to only two threads and did not delve into the performance scalability aspects of DSWP. In the current paper, we present a thorough analysis of the performance scalability of automatically generated DSWP codes.

Based on our analysis, we identify interesting communication bottlenecks that prevent DSWP from yielding theoretically predicted performance with increasing thread count. We observe that thread pipelines are of two types: *linear* and *nonlinear*. Linear pipelines are a chain of threads. They are characterized by strict pairwise interactions among threads, i.e., each thread in the pipeline consumes from, at most, one upstream thread and produces to, at most, one downstream thread. Nonlinear thread pipelines, on the other hand, are directed acyclic graphs (DAG). Even though, in principle, there are no cyclic interthread dependences in a PMT transformation, such as DSWP, the use of finite-sized queues creates cyclic interthread dependences called *synchronization cycles*. Synchronization cycles require a producer of a queue item to block until the queue is nonempty. For reasonably sized queues (for example, eight entries and above) and linear pipelines, synchronization cycles never lead to performance bottlenecks, since the slack provided by the queue sizing is sufficient to tolerate them. However, it will be shown that, for similarly sized queues, the synchronization cycles for nonlinear pipelines have such high latencies that the slack provided by queue sizing is not sufficient to tolerate the long delays.

The paper shows that while good thread balance is important for optimal DSWP performance, superpartitioning of an application loop among multiple threads can create complex communication patterns, which can interact pathologically with the underlying communication substrate. Consequently, it

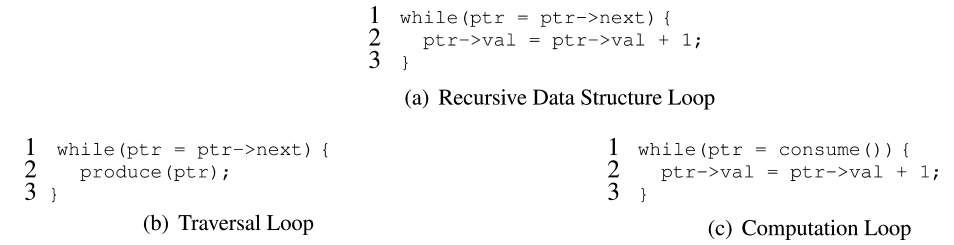


Fig. 1. Splitting RDS loops.

is important for the compiler's partitioning heuristic to be aware of and avoid such communication pathologies. This paper shows how to gauge the minimum communication buffering requirements for a given partitioning, which can then be used by a compiler's partitioning heuristic to generate code so as to avoid any runtime communication bottlenecks.

The remainder of the paper is organized as follows. Section 2 presents background information on DSWP, including a brief description of the automatic DSWP algorithm [Ottoni et al. 2005], and compares the scalability issues of DSWP with other multithreading techniques. Information about the evaluation infrastructure, the benchmarks used, analysis, and performance measurement methodology is provided in Section 3. The performance scalability study of DSWP is presented in Section 4. Section 5 summarizes key results from this work and provides pointers to future research directions.

2. DECOUPLED SOFTWARE PIPELINING

This section presents background material on decoupled software pipelining (DSWP) and briefly describes the automatic DSWP algorithm to parallelize generic application loops. It discusses how DSWP differs from other multicore/multithreading strategies aimed at improving single program performance.

2.1 Background

DSWP, a nonspeculative PMT transformation, started as a mechanism to effectively tolerate variable latency stalls imposed by delinquent memory loads, without resorting to complex speculative issue processors [Rangan et al. 2004]. DSWP was used to parallelize recursive data structure (RDS) loops to execute as two concurrent nonspeculative threads: a critical path (CP) thread comprising the traversal slice and an off-critical path (off-CP) thread comprising the computation slice. For example, consider the loop shown in Figure 1(a). The traversal slice consists of the critical path code, `ptr=ptr->next`, and the computation slice consists of `ptr->val=ptr->val+1`. A DSWP parallelization of this loop yields a traversal and computation thread as shown in Figures 1(b) and (c), respectively. In the figure, the `produce` function enqueues the pointer onto a queue and the `consume` function dequeues the pointer. If the queue is full, the `produce` function will block waiting for a slot in the queue. The `consume` function will block waiting for data, if the queue is empty. In this way, the traversal

and computation threads behave as a traditional decoupled producer–consumer pair.

The above parallelization effectively decouples the execution of the two code slices and allows useful code to be overlapped with long variable-latency instructions without resorting to speculation or extremely large instruction windows. Unidirectional interthread communication enables the use of a decoupling buffer. The decoupling buffer insulates each thread from stalls in the other thread. Further, the reduced size loop of the individual threads allows the program to take better advantage of traversal cache hits to initiate traversal cache misses early. From an ILP standpoint, this allows for an overlap between traversal and computation instructions from distant iterations. Simulation studies have shown that DSWP-parallelized recursive data structure codes running on a dual-core processor achieve latency tolerance not just through a prefetching effect, but also due to DSWP's ability to expose useful nonspeculative parallelism to take advantage of the extra functional units provided by an additional processing core [Rangan et al. 2004].

In its general form, DSWP tolerates variable latency resulting from not only memory loads, but also variable trip count inner loops, imbalanced if-then-else hammocks, and floating-point operations. It is also an effective means to parallelize resource-constrained applications loops with recurrences across multiple cores. The next subsection describes an automatic DSWP technique to parallelize generic program loops across two or more threads.

2.2 Automatic DSWP

While the above technique based on identifying critical and off-critical path threads can only yield two threads (for example, traversal and computation threads for RDS codes), Ottoni et al.'s [2005] general-purpose algorithm departs from the notion of identifying critical and off-critical paths of the regions targeted for DSWP. Instead, it focuses on identifying more generic program recurrences and achieves acyclic dependence flow among threads by ensuring that no single recurrence crosses thread boundaries. This approach enables the automatic DSWP algorithm to be a truly general-purpose multithreading technique.

The automatic DSWP technique (*autoDSWP* for short) takes a loop's dependence graph, which contains all register, memory, and control dependences, as input. In order to create an acyclic thread dependence graph for pipelined parallelism, it first identifies strongly connected components (SCCs) in the input dependence graph. The graph formed by these SCCs, by definition, will be a directed acyclic graph (DAG_{SCC}). The algorithm then partitions the DAG_{SCC} into the requisite number of threads while making sure that no cyclic interthread dependences are created. The original loop body is now partitioned among the DSWP threads. In this manner, *autoDSWP* parallelizes program loops into pipelined threads. For this paper, we use a load-balancing heuristic in our *autoDSWP* compiler to appropriately schedule the SCCs among the requisite number of threads.

DSWP delivers improved performance through coarse-grained overlap among pipelined threads and improved variable latency tolerance. The amount of overlap is limited by the performance of the slowest thread. Since the granularity of scheduling in *autoDSWP* is a single SCC, the maximum theoretical speedup attainable is 1/Normalized weight of heaviest (slowest) SCC. Thus, there is an upper bound to the performance obtainable from merely scheduling and balancing the SCCs across available threads. A discussion of other optimizations that may be needed to further break or parallelize the individual SCCs to expose more parallelism is beyond the scope of this paper.

The current thread model for DSWP is as follows. Execution begins as a single thread, called *primary thread*. It spawns all necessary *auxiliary threads* at the beginning of a program. When the primary thread reaches a DSWPed loop, auxiliary threads are set up with necessary loop live-in values. Similarly, upon loop termination, loop live-outs from auxiliary threads have to be communicated back to the primary thread. While this creates a cycle in the thread dependence graph, any dynamic cost because of this once-per-loop-invocation event, will be rendered negligible by long-running pipelined multithreaded loop bodies. However, this cycle can become a significant overhead for short-running loops, which are not good candidates for DSWP in the first place.

The interthread decoupling queues can be implemented with any of the designs presented in Rangan et al. [2006]. However, all performance evaluation in this paper is done with the high-performance synchronization array (SA) communication support [Rangan et al. 2004] to achieve fast low-overhead interthread communication. The software interface of the SA comprises of blocking `produce` and `consume` instructions, which enqueue and dequeue 64-bit operands, respectively.

The next section discusses the scalability potential and limitations of DSWP in relation to other nonspeculative as well as speculative multithreading techniques.

2.3 Related Work

In contrast to DSWP, which partitions a given loop body across threads so as to keep all recurrences local to a thread, DOACROSS parallelization [Cytron 1986] keeps the original loop body unchanged and communicates recurrences across threads. Figure 2 illustrates the space–time behavior of DSWP and DOACROSS parallelization techniques. The dependence graph of a single-threaded loop body is shown in Figure 2(a). This is also the DAG_{SCC} for DSWP. The single-threaded DOACROSS and DSWP schedules of this loop are given in Figures 2(b), (c), and (d), respectively. The y axis represents time and the x axis represents the threads across which the original loop is parallelized. The dashed dark box in the DOACROSS and DSWP schedules highlights how one loop iteration executes in each case. Notice that even though the loop bodies of the individual DSWP threads are different from the original single-threaded loop body, both the DSWP and DOACROSS parallelizations provide identical performance improvement (i.e., 4X) over single-threaded execution.

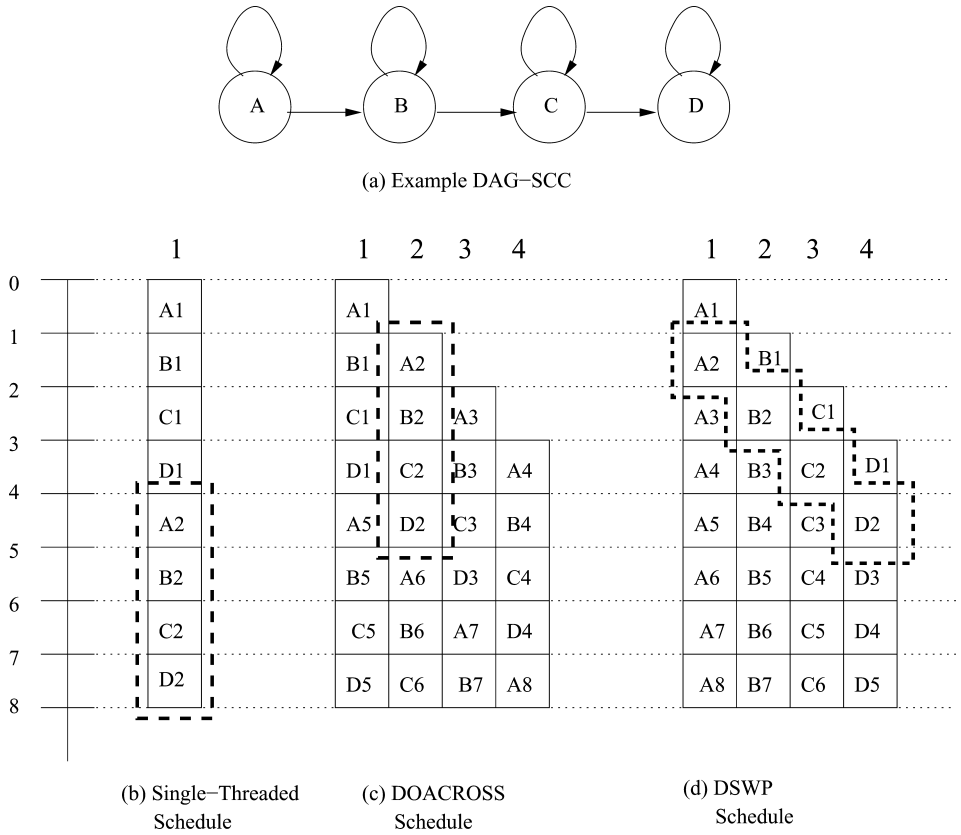


Fig. 2. Comparison of space-time behavior of DSWP and DOACROSS parallelization.

Therefore, theoretically speaking, the scalability potential is similar in both DSWP and DOACROSS; their speedup being bound by the performance of the slowest SCC or recurrence, respectively. However, their dynamic communication behavior is vastly different, leading to unique performance scalability issues in these techniques. The difference arises from the nature of interthread dependences in the two techniques. Since recurrences are communicated across DOACROSS threads, they are exposed to communication latencies and, hence, are very sensitive to communication costs. For the same reason, DOACROSS is also not effective in dealing with variable latency stalls, since there is no opportunity to build sufficient decoupling between communicating DOACROSS threads. Communication of recurrences between threads in DOACROSS results in lock-step behavior. Consequently, efficient synchronization is critical to scalable DOACROSS performance. On the other hand, DSWP does not suffer from the lock-step synchronization problem because of acyclic interthread dependences. However, it has to contend with performance bottlenecks arising from inefficient interthread operand queueing, as will be shown in Section 4, in order to achieve optimal scalability.

Unlike DSWP and DOACROSS, the applicability of DOALL parallelization [Lundstorm and Barnes 1980] is limited to loops without recurrences. In

DOALL parallelized programs, each thread executes one or more loop iterations independent of other threads. No interthread dependences need to be communicated in this style of parallelization. Consequently, its performance potential is limited only by the available processor count and the number of iterations in a given loop. Thus, given a loop that does not have any recurrences, DOALL is the most preferred parallelization strategy. If, however, the loop has recurrences, then depending on performance or flexibility requirements, one may use DSWP or DOACROSS parallelization, respectively.

Decoupled access-execute (DAX) architectures [Smith 1982] statically partition the code into a memory-access stream and an execute stream comprising ALU operations. Dynamically, the two streams execute in separate processing cores decoupled by FIFOs. This setup proved useful for tolerating memory latencies in applications with high memory-level parallelism (MLP) (i.e., many memory loads can be initiated before any of their results are used). However, the cyclic dependence of the execute stream on the access stream causes the performance of DAX architectures to degenerate to or become worse than the performance of traditional processing cores when handling codes with poor MLP, such as RDS loops, with rising intercore communication costs. DSWP avoids this problem by enforcing acyclic communication among its threads. Since DAX partitions instructions based on their type, it cannot scale to more than two streams with an optional prefetch stream to improve access stream performance [Ro et al. 2006]. DSWP, on the other hand, exposes inherent pipelined parallelism available in program loops by partitioning based on program recurrences and can scale to as many threads as there are SCCs in a given loop.

Techniques like flea-flicker [Barnes et al. 2003, 2005] and dual-core execution [Zhou 2005] tolerate load miss latencies by holding missed loads and their dependent instructions in intra- or intercore queues for deferred processing by a second core. When the load miss is resolved, all instructions are committed in sequential order by the second core. This deferred handling enables the first core to continue execution past a load miss. The similarity of these techniques with DSWP is limited to the use of a decoupling queue between cores. These techniques are microarchitectural optimizations to improve load miss handling in single-threaded programs and can complement DSWP.

In addition to the aforementioned techniques, the literature abounds in several speculative multithreading techniques aimed at improving single-threaded performance. While some of these techniques seek to extract more ILP by increasing the effective logical instruction window size, others improve performance by using helper threads to warm up microarchitectural structures of the main program thread. Thread-level speculation [Hammond et al. 2000; Steffan et al. 2005], multiscalar processors [Sohi et al. 1995], superthreading [Tsai et al. 1999], and trace processors [Vajapeyam and Mitra 1997] are representative techniques of the first kind that speculate input dependences of dynamically far-apart regions and spawn them as speculative threads on available cores. The threads are all committed in program order upon validation of their speculative inputs. Subordinate microthreading [Chappel et al. 1999] spawns microcoded threads at strategic points from the main program thread to warm

up caches and branch predictors. Speculative pre-computation [Collins et al. 2001; Roth and Sohi 2001] and helper-thread prefetching [Wang et al. 2002] spawn execution-driven speculative threads to warm up cache structures. Instead of throwing away all the work of speculative threads, some techniques optionally register-integrate the computation results of the speculative threads with the main thread.

A main characteristic of these speculative multithreading techniques is that they were evolved to operate under the constraints of the traditional single-threaded execution model. Instruction fetch and commit are *still* serialized. In order to overcome the fundamental restrictions imposed by the execution model, architects have gone to great lengths to build copious amounts of buffering in the processing core to hold unissued or uncommitted instructions or to hold speculative architectural state updates of executed-but-uncommitted instructions. Scalability of these speculative techniques depends not only on the inherent parallelism available, but also on the accuracy of speculation as well as the execution efficiency (performance per watt, performance per transistor, etc.) of these techniques.

In contrast, DSWP avoids heavy hardware usage by attacking the fundamental problem of working within a single-threaded execution model and moving to a concurrent multithreaded execution model. Since DSWP is an entirely *nonspeculative* technique, each DSWP thread performs useful work toward program completion. DSWP threads are typically long-running and do not require frequent thread spawns. Individual, concurrent threads commit register and memory state independently of other threads. Consequently, the amount of storage required to communicate true interthread dependences is insignificant compared to the speculative storage in techniques like TLS and other single-threaded multicore techniques. These characteristics argue well for DSWP's execution efficiency and scalability.

Next, in Section 3, we discuss details of our evaluation methodology. We present experimental results in Section 4.

3. EVALUATION METHODOLOGY

This section provides details about the benchmark applications, compiler, simulator, and sampling methodology used in this paper.

3.1 Benchmarks and Tools

All quantitative evaluation presented in this paper uses code produced by the VELOCITY compiler framework [Triantafyllis et al. 2006]. An *autoDSWP* implementation in the VELOCITY framework produced DSWPed codes. A diverse set of applications drawn from several publicly available benchmark suites is used for evaluation. The benchmarks studied include `art`, `mcf`, `equake`, `ammp`, and `bzip2` from the SPEC-CPU2000 benchmark suite, `epicdec` and `adpcmdec` from the MediaBench [Lee et al. 1997] suite, `mst`, `treeadd`, `em3d`, `perimeter`, and `bh` from the Olden suite, `ks` from the Pointer-Intensive benchmark suite, and the Unix utility `wc`. A key loop in each of these applications is targeted

Table I. Loop Information

Benchmark	Function	Exec. Time (%)	Benchmark Description
mst	BlueRule	100	Minimal spanning tree
treeadd	TreeAdd	100	Binary tree addition
perimeter	perimeter	100	Quad tree addition
bh	walksub	100	Barnes-Hut N-body simulation
em3d	traverse_nodes	100	3D electromagnetic problem solver
wc	cnt	100	Word count utility
ks	FindMaxGpAndSwap	99	Kernighan-Lin graph partitioning
adpcmdec	adpcm_decoder	98	Adaptive differential PCM sound decoder
equake	smvp	68	Earthquake simulation
ammp	mm_fv_update_nonbon	57	Molecular mechanics simulation
mcf	refresh_potential	30	Combinatorial optimization
epicdec	read_and_huffman_decode	21	Image decoder using wavelet transforms and Huffman tree based compression
art	match	20	Neural networks based image recognition
bzip2	getAndMoveToFrontDecode	17	Burrows-Wheeler compression

for DSWP. A short description of each application and details about the loop chosen from each benchmark are provided in Table I. All the Olden benchmarks, except for em3d, which were originally recursive implementations, were rewritten to be iterative procedures, since the DSWP compiler can handle only regular loops at this time. The different code versions for all benchmarks were generated with all the classic optimizations turned on. In all cases, instruction scheduling for control blocks was done both before and after register allocation.

The generated codes were then run on a multicore performance simulator constructed with the liberty simulation environment [Vachharajani et al. 2002, 2004]. The multicore simulator was derived from a validated core model, which was shown to be within 6% of the performance of native Itanium 2 hardware [Penry et al. 2005]. This framework does not model a hardware or a software thread scheduler. Therefore, an N-core configuration can run, at most, N threads.

Details of the baseline in-order model are given in Table II. The synchronization array [Rangan et al. 2004] was integrated into this model such that produce instructions stalled at the *REG* stage of the Itanium 2 pipeline [Intel Corporation 2002] on queue full conditions and consume instructions stalled in the *EXE* stage on queue empty conditions. Each core can initiate, at most, four produce or consume operations. We model a centralized synchronization array to which cores connect via a dedicated interconnect. The compiler ensures that each pair of communicating threads uses a globally unique queue identifier to send and receive operands. The effective instructions per cycle (IPC) of the baseline single-threaded code for the above benchmark loops on the baseline in-order simulator model is given in Figure 3. Note, the effective IPC calculation excludes no-ops and predicated-off instructions.

Table II. Baseline Simulator

Core	Functional Units - 6-issue, 6 ALU, 4 Memory, 2 FP, 3 Branch Misprediction pipeline - 7 stages L1I Cache - 1 cycle, 16 KB, 4-way, 64-B lines L1D Cache - 1 cycle, 16 KB, 4-way, 64-B lines, Write-through L2 Cache - 5,7,9 cycles, 256KB, 8-way, 128-B lines, Write-back Maximum Outstanding Loads - 16
Shared L3 Cache	>12 cycles, 1.5 MB, 12-way, 128-B lines, Write-back
Main Memory latency	141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration
Synchronization Array (SA)	Single centralized 4-ported structure, 1-cycle access, 32-entry queues
SA interconnect	4 independent 1-cycle, 8-byte buses connect the cores to the 4 SA ports, Arbiter always favors upstream threads

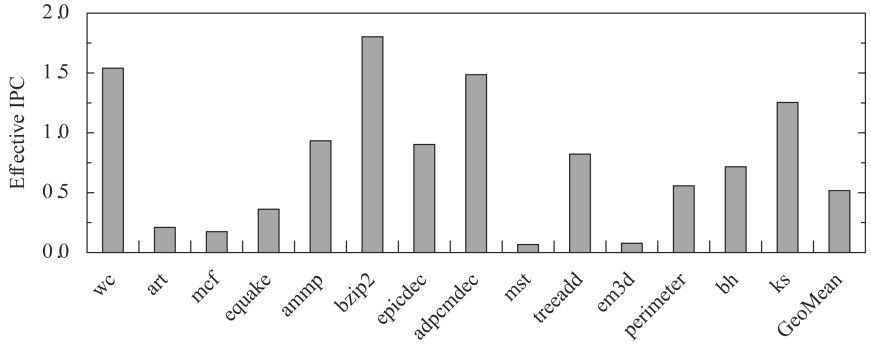


Fig. 3. Raw performance of single-threaded code on baseline model.

3.2 Performance Measurement

Since the number of instructions executed is not constant across different code versions (single-threaded, two-thread DSWP, four-thread DSWP, etc.), aggregate statistics like instructions per cycle (IPC) or clocks per instruction (CPI) will not suffice to capture performance improvements across code changes. Instead, execution time has to be used to compare performance over different configurations (across code and microarchitecture changes). However, pure execution time does not yield any insight into the run-time behavior of codes and analysis becomes difficult. Therefore, we use a cycle accounting methodology to breakdown overall execution time to aid in performance bottleneck analysis.

3.3 Sampling Methodology

Application of DSWP to a program loop leaves the pre- and postloop code almost[2] untouched. As a result, their performance is the same across all single- and multithreaded executions for a given hardware configuration (modulo minor

[2]There are minor changes to propagate loop live-ins and outs from and to the primary thread, respectively.

differences because of cache state differences). Therefore, detailed simulation and performance measurement is done only for DSWPed loops across various threading versions. However, the highly detailed modeling of core, as well as memory architectures, and the large input set sizes of benchmarks preclude the possibility of simulating all iterations of each and every invocation of a given loop in a reasonable time.

In order to drive down simulation time, we collect TurboSMARTS-style [Wenisch et al. 2004] checkpoints of architectural and microarchitectural state at the beginning of randomly chosen loop invocations. Loop iteration granular SMARTS [Wunderlich et al. 2003] sampling is initiated from these checkpoints. We simulate 10,000 loop iterations for all benchmarks, except `188.ammp` and `ks`, whose outer loops are parallelized resulting in smaller sample sets. The 10,000 figure is not large enough to estimate the performance of an individual code-model configuration (for example, single-thread execution on baseline machine model) to a desired confidence level with a narrow enough confidence interval. However, performance *comparison* metrics (e.g., speedup of one technique relative to another) for a given application across code/architecture changes tend to demonstrate strong correlation during various phases of program execution and have been shown to require far fewer samples to yield tight confidence intervals [Luo and John 2004]. The performance comparisons in this paper are given at a 95% confidence level and the accompanying error bars are shown in all speedup graphs. The accompanying error bars confirm that the sample set size chosen is sufficient to yield narrow enough confidence intervals.

The next section presents DSWP performance scalability results.

4. PERFORMANCE SCALABILITY OF DSWP

Experimental results from simulating two-, four-, six-, and eight-thread DSWP codes are presented in this section and their performance analyzed in detail. For each benchmark, the compiler created the requested number of threads, N, only if it could heuristically determine that it was profitable to partition the loop's SCCs among N threads, taking into account the relative balance of computation and communication costs. These experiments used N-way multicore simulator models with in-order Itanium 2-like cores, where N equaled the number of application threads.

Figure 4 shows the speedup provided by automatically generated DSWP threads relative to single-threaded in-order execution, when moving from two threads to four, six, and eight threads. This performance graph will be referred to as $DSWP_{Q_{32}}$. Note that the graph shows two geometric means: a plain geometric mean (denoted "GeoMean" in the graph) and a best geometric mean (denoted "Best-GeoMean" in the graph). When calculating the plain geometric mean for N-thread code versions, if a certain benchmark did not have an N-thread code version (for example, `equake` does not have 6- and 8-thread versions), then for that benchmark, the speedup of the version with the next highest number of threads is used. For example, when calculating the *plain geometric mean* across all eight-thread versions, the speedup of `equake` from the

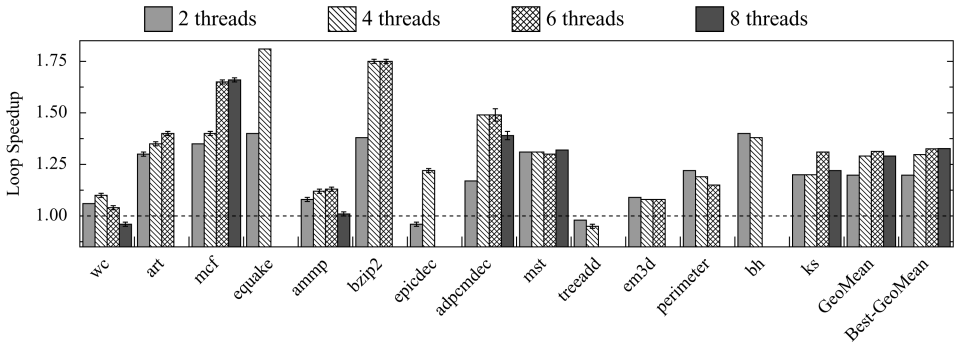


Fig. 4. $DSWP_{Q_{32}}$: DSWP performance when moving from two to four, six, and eight threads with 32-entry interthread queues and a bus interconnect. Note, a missing bar for a particular number of threads for a benchmark means the compiler was not able to partition the chosen loop for that benchmark into that many number of threads.

four-thread version is used, since it does not have a code version with more than four threads. On the other hand, the *best geometric mean* for an N-thread version represents the mean of the best speedups across all benchmarks for all code versions with number of threads fewer than or equal to N. It represents the speedup that can be achieved with code generated by an intelligent compiler that will generate the best multithreaded version for each benchmark for a given number of cores, even if it means generating fewer threads than available cores. The analysis presented here primarily uses the plain geometric mean to compare DSWP's performance in the presence and absence of bottlenecks. The best geometric mean is also provided to highlight the maximum speedup achievable through careful selection of multithreaded code versions.

To understand the performance of $DSWP_{Q_{32}}$, recall that the *autoDSWP* technique partitions the DAG_{SCC} such that there are no backward dependences among partitions. Since the performance of pipelined multithreading is limited by the performance of the slowest running thread, optimal DSWP performance can be achieved by placing the "heaviest" SCC in a partition of its own and by making sure that no other partition is heavier than the partition with the heaviest SCC. This can be done by either load-balancing the remaining partitions in such a way so as to not exceed the weight of the heaviest SCC or, if that is not possible, then, each SCC can be placed in its own thread. The heaviest thread is called the *bottleneck thread*. Ofttimes, application loops contain a few large SCCs and many small, mostly single-instruction, SCCs. Once the heaviest SCC has been placed in a thread of its own and no other partition is heavier (including ones with more than one SCC), the heaviest SCC thread becomes the bottleneck thread and it is no longer possible to obtain any more performance improvement by partitioning the remaining SCCs among more threads. This trend is clearly seen in Figure 4, which shows that even for benchmarks that yield more than two threads, no performance improvement is seen beyond six threads.

On the contrary, a performance *slowdown* is seen for some application loops when moving to more threads, which is somewhat counterintuitive. The

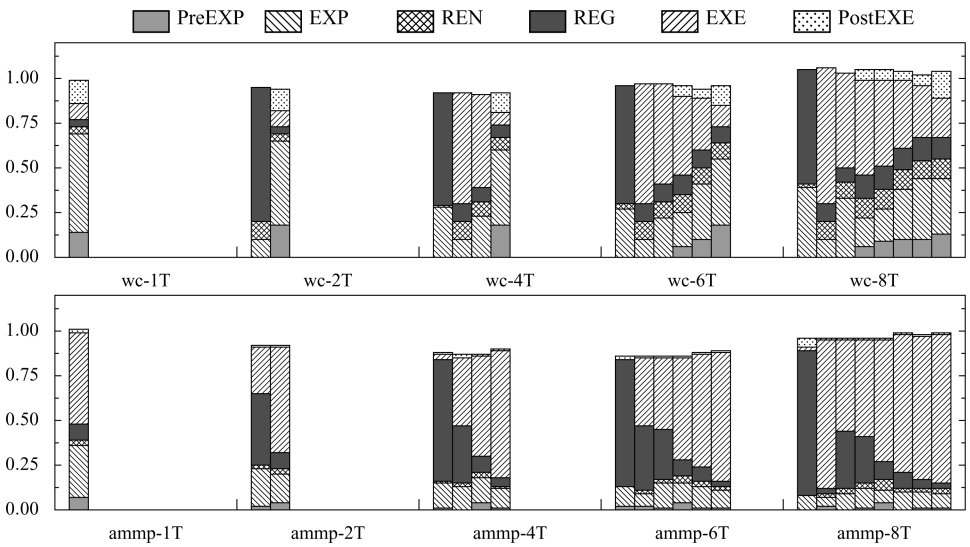


Fig. 5. Normalized execution time breakdown of `wc` and `ammp` when moving to more threads with 32-entry queues and bus interconnect.

theoretical performance improvement expected when moving to more threads no longer holds. Since *autoDSWP* virtually does no code duplication, the above slowdown, when moving to more threads, cannot be because of differences in the amount of computation. While the total computation remains constant across the four different multithreaded partitionings, the amount of communication varies. Figure 5 provides the normalized execution time breakdown of each thread for the different multithreaded partitionings for two representative benchmarks, `wc` and `ammp`. The figure shows how the execution time of each benchmark for each thread configuration is spent in different stages of the processor pipeline. We use our detailed cycle accounting methodology to break down the total time spent into six aggregate stall groups: *PreEXP* (comprises stalls in the instruction fetch stages of the pipeline), *EXP* (stalls in the decode stage), *REN* (stalls in the register renaming stage), *REG* (stalls in the scoreboarding and register access stage and synchronization array renamer), *EXE* (stalls in the execution stage, which accounts for all execution time, including memory access and synchronization array access), and *PostEXE* (cycles spent by the leading critical instruction in the exception-detection and write-back stages of the Itanium 2 pipeline). The x axis of each graph in Figure 5 represents the various code partitionings: *bench*-1T, *bench*-2T, *bench*-4T, *bench*-6T and *bench*-8T, of each benchmark *bench*. Within a cluster, for example, *bench*-6T, the normalized execution time breakdown of each thread of that configuration is shown. Within a cluster, the bars corresponding to "upstream" threads appear to the left and the bars corresponding to "downstream" threads appear to the right.

The breakdowns show that when moving to more threads, the *EXE* component increases dramatically compared to the one or two thread configurations. A fine-grained breakdown of the stalls on a per instruction basis revealed that `consume` instructions were the main reason that led to the increased *EXE*

component. For example, in the graph for `wc`, note that while the first thread has a large *REG* component, because of frequent stalls by `produce` instructions on queue full conditions, the second and third threads have large *EXE* components, as a result of frequent stalls by `consume` instructions on queue empty conditions. Given the classical notion of pipelined execution, this is very counterintuitive. To explain this performance anomaly when moving to more threads, it is important to understand the actual communication pattern among threads and their runtime behavior.

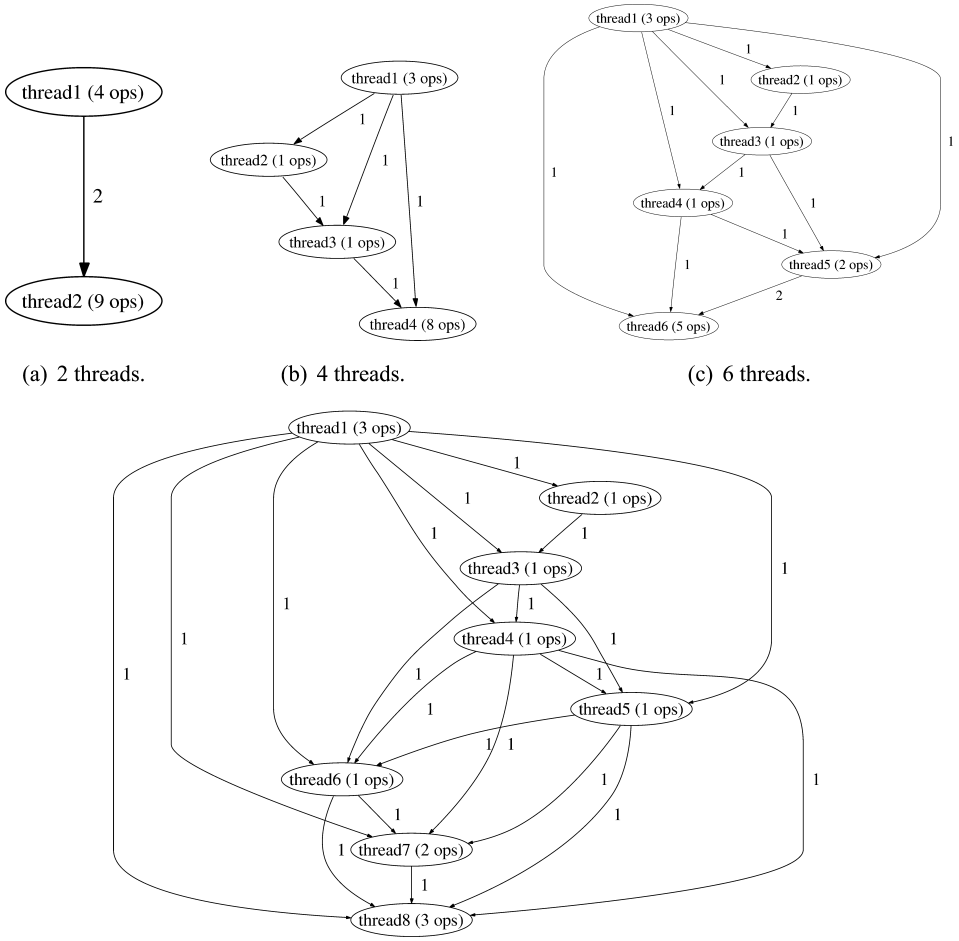
4.1 Linear and Nonlinear Thread Pipelines

Given a linear chain of producer-consumer threads (i.e., thread 2 consuming from thread 1, thread 3 consuming from thread 2, and so on), the communication rate in the chain will be determined by the slowest thread and all threads will produce and consume at the exact same rate as the slowest thread. Such thread pipelines will be called *linear pipelines*. As mentioned before, the maximum performance attainable by such thread pipelines is S/D_H, where S is the single-threaded execution time, D_i is the execution time of thread i of the pipeline, and H is the slowest thread in the pipeline. This expression does not say anything about the communication requirements of such pipelines. In particular, if a linear pipeline had insufficient queue buffering, then the factor D_H will increase to include the time the slowest thread spends waiting for data arrival, thereby adding intercore communication delays to the overall thread execution time. However, such a situation can be avoided if interthread queues are sized appropriately. In particular, if the time taken to communicate a data item or a synchronization token from one thread to another is C cycles, it takes a total of $2 \times C$ cycles for a producer thread to communicate a value to a consumer thread and for the consumer to communicate its acknowledgment to the producer. This round-trip communication will be called a *synchronization cycle*. Since all threads in the pipeline need only communicate at the same rate as the slowest thread H, i.e., once every D_H cycles, all interthread queues need only be as big as the queues leading into and out of thread H. If the synchronization cycle delay, $2 \times C$, is less than D_H, then D_H is the limiting factor and only one entry is needed in all interthread queues (no buffering is needed if communication happens instantaneously, i.e., C equals 0). On the other hand, if the round-trip time is greater than D_H, then the number of loop iterations the slowest thread can execute in that time is $(2 \times C)/D_H$. Consequently, there needs to be at least these many queue slots to keep the slowest thread continuously busy. Thus, the minimum[3] queue size necessary to prevent intercore communication delays from being added to thread execution times is given by $\lceil (2 \times C)/D_H \rceil$.[4]

As long as the intercore communication latency is less than or equal to the computation time, queue sizes of 1 or 2 will suffice to provide peak throughput.

[3]This is the minimum queue size without accounting for variability in data production and consumption rates.

[4]This can be easily augmented to account for different communication costs between different pairs of threads.



(a) 2 threads. (b) 4 threads. (c) 6 threads.

(d) 8 threads.

Fig. 6. Thread dependence graphs for loop from `wc`.

Increase in computation time will only reduce the demand for more queue entries. This is a very desirable property of linear pipelines as it helps place reasonable bounds on interthread queue sizes, enabling optimal communication support design.

However, in practice, general-purpose applications often do not yield linear pipelines. As SCCs are partitioned among more threads, more interthread dependences are created among threads, since previously local (*inter*-SCC, but *intra*thread) dependences may now need to be communicated between threads. The communication pattern among constituent threads is quite varied and the partitioner ends up creating dependences between *almost* every pair of up- and downstream threads. Such thread pipelines will be referred to as *nonlinear pipelines*. Consider the example of `wc` shown in Figure 6. The figure shows the

thread dependence graphs of different partitionings of the `wc` loop. It also shows the number of operations (compiler intermediate representation operations, not machine operations) in each thread and the label on each edge indicates the number of queues running between a pair of threads. Except for the two-threads case (Figure 6a), the dependence graphs for the other cases (four-, six- and eight-thread cases in Figure 6(b), (c), and (d), respectively) do not turn out to be linear pipelines. Such nonlinear thread pipelines, while still providing PMT parallelism, experience certain communication bottlenecks that lead to below par performance.

To understand the communication bottlenecks in such pipelines, consider the thread dependence graphs and the execution schedules of a four-thread linear pipeline, *ABCD*, and a four-thread nonlinear pipeline, $A'B'C'D'$, in Figures 7(a) and (b), respectively.

The dashed arcs in the backward direction in the thread dependence graphs represent synchronization dependences from consumer threads back to their producers. These arcs indicate to the producer when it is permissible to write to a particular queue location. When a consumer thread is slower than the corresponding producer thread, the latter has to block after filling up the queue, until the consumer thread frees up a queue slot, to produce the next data item. These arcs become relevant when interthread queue buffering is not sufficient to tolerate intercore communication delays and the difference in data production and consumption rates.

For illustration purposes, suppose one iteration of the original single-threaded loop takes 120 cycles and the individual threads each take 40 cycles, in both *ABCD*, as well as $A'B'C'D'$, for executing one iteration of the loop in question. Let the interthread communication latency be ten cycles. Even though, in practice, the compiler is free to schedule the communication instructions anywhere in a thread, for this example, assume that all `produce` instructions are executed at the end of thread's loop iteration and all `consume` instructions, at the beginning. Finally, for simplicity, all `produce` instructions in a thread will be blocked if any one `produce` blocks. A similar all-or-none behavior will be assumed for `consume` instructions as well. In the execution schedules shown in the above figure, a solid interthread arrow means a data value communication from the producer thread to its consumer. A dashed interthread arrow in the reverse direction denotes an acknowledgment signal from a consumer to a producer, indicating a queue entry is free to be reused. Dotted straight lines in a thread's schedule indicate periods of no activity in the thread, because it is blocked on a produce or a consume operation.

As the execution schedule in Figure 7(a) shows, the linear pipeline *ABCD* is able to finish a loop iteration once every 40 cycles. Notice that because the per-iteration time is 40 cycles and the synchronization cycle delay (round-trip time) is only 20 cycles, the acknowledgment for a queue entry arrives well before a producer thread is ready to produce the next data item. Therefore, a producer will never block on a queue full condition and the performance of the linear pipeline attains the theoretical maximum speedup of $S(120cycles)/D_H(40cycles)$ i.e., 3X. The important point to note here is that the linearity of the pipeline enables it to achieve this speedup with just 1 entry per interthread queue.

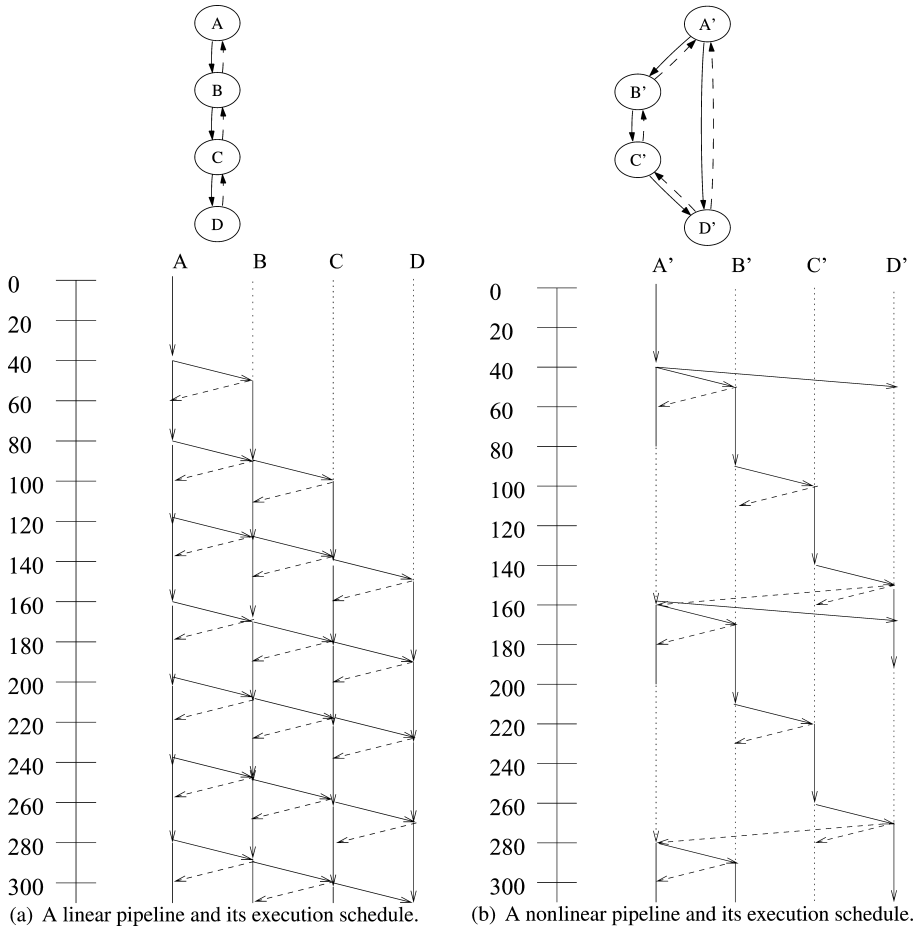


Fig. 7. Linear and nonlinear thread pipeline execution with intercore delay of 10 and per-thread iteration computation time of 40 cycles.

Now, consider the nonlinear pipeline $A'B'C'D'$'s execution schedule in Figure 7 (b). This schedule has been drawn assuming one-entry interthread queues to contrast its performance with the linear pipeline from above. Notice that $A'B'C'D'$ is able to complete only 1 loop iteration every 120 cycles (the first iteration of thread D' completes in cycle 190 and the second iteration in cycle 310) resulting in no speedup at all over single-threaded execution. The reason for this abysmal performance is because of inadequate queue sizing that leads to prolonged stalls, as can be seen by the long dotted lines in all threads in Figure 7(b). So, why do single-entry queues, which were adequate to deliver peak throughput in the linear pipeline above, create performance bottlenecks here?

To answer the question, observe that in Figure 7(b), thread A' sends values to both threads B' and D'. Consequently, in any given iteration, before producing a data item, thread A' has to ensure that it can produce into the queues

leading into both thread B' and thread D' before initiating both data sends (per assumptions stated above). In other words, whenever it is blocked on queue full condition, thread A' has to wait for acknowledgments from both threads. This requirement leads to communication bottlenecks, thereby slowing down multithreaded performance. In the execution schedule, notice that even though thread A' is ready to produce data after its second iteration as early as cycle 80, it has, by that point in time, received acknowledgment only from thread B'. It has to wait a further 80 cycles before it receives acknowledgment from thread D', at which point, it proceeds to produce the data to both threads B' and D'. Since thread A' is at the head of the pipeline, the rest of the pipeline also stalls, waiting for data from upstream threads. The fundamental problem here is that queue sizes for executing such nonlinear pipelines cannot be determined solely from the interthread communication delay and the per-iteration computation time of the slowest thread.

For nonlinear pipelines, the synchronization cycle expands to include the computation time of all intermediate threads, as well as the one-way communication delays between the intermediate threads. For example, for thread A' above, the synchronization cycle comprises the communication delay from thread A' to B', the computation time in thread B', the communication delay from thread B' to C', the computation time of thread C', the communication delay from thread C' to D', and, finally, the delay for the acknowledgment to go from thread D' to A'. More generically, the round-trip delay of the new longer synchronization cycle can be expressed as $2 \times C + \sum_{i=1}^{N_s}(D_i + C)$, where N_s is the number of threads in the synchronization cycle s. By a similar reasoning as from before, the queues should be large enough to tolerate this round-trip delay, but only to the point of sustaining the maximum throughput. Thus, for nonlinear pipelines, the minimum queue size needed to provide peak PMT performance is

$$\left\lceil \frac{2 \times C + \max_{s,\forall s \in S}\left(\sum_{i=1}^{N_s}(D_i + C)\right)}{D_H} \right\rceil$$

where S is the set of all synchronization cycles in a given thread dependence graph. The second term in the numerator causes nonlinear pipelines to require longer queues to deliver peak throughput. This term also makes nonlinear pipelines unwieldy for communication support design, since it is impossible to place an upper bound on the size of the interthread queues. Synchronization cycles can be made arbitrarily long because of the computation costs of intermediate threads, making it very difficult to design bottleneck-free communication support. This explains the anomaly in Figure 5, wherein threads experienced increased *EXE* stalls, as a result of the creation of nonlinear thread pipelines, when moving to more threads.

4.2 Relaxing Queue-Size Constraints

The 32-entry queue sizing was insufficient to tolerate longer synchronization cycles created by nonlinear thread pipelines. This phenomenon is particularly acute when moving to eight threads. To remedy the situation and evaluate the performance scalability potential of DSWP when moving to more threads, a

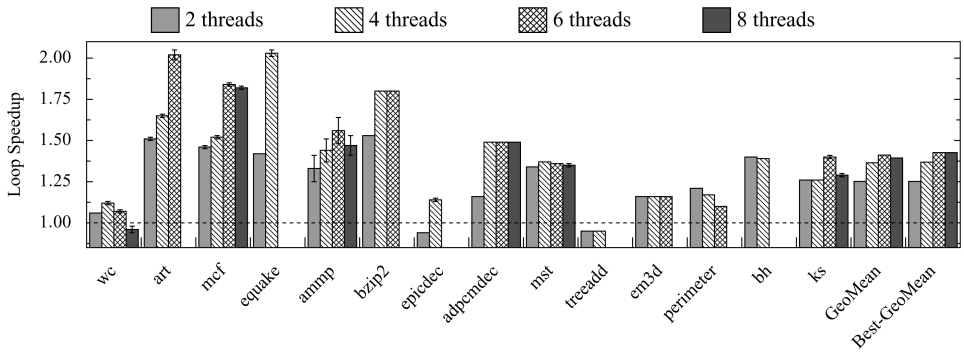


Fig. 8. $DSWP_{Q_\infty}$: Performance of DSWP with two, four, six, and eight threads with infinitely long queues and a bus interconnect relative to single-threaded in-order execution.

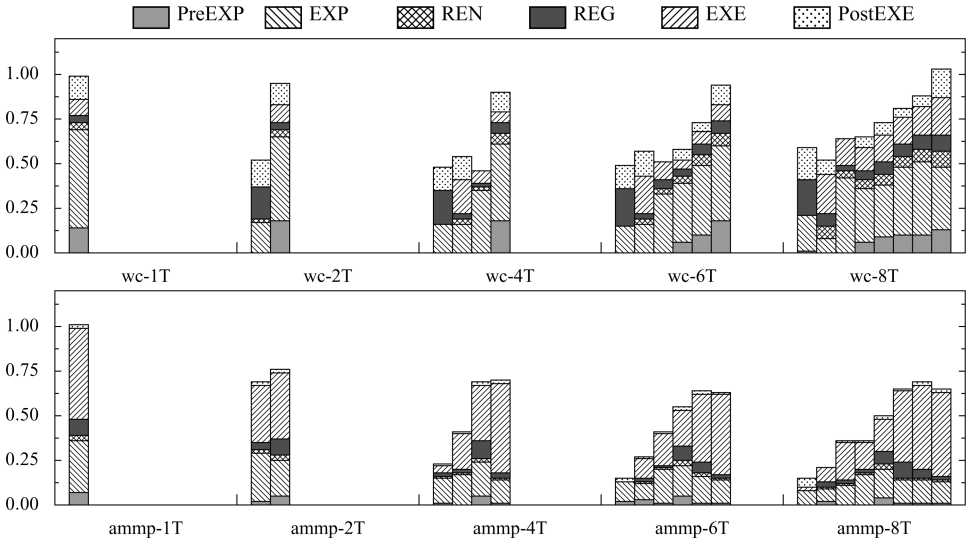


Fig. 9. Normalized execution time breakdown of `wc` and `ammp` when moving to more threads with infinitely long queues and bus interconnect.

second set of simulations (labeled $DSWP_{Q_\infty}$) were run, once again on different multicore configurations, with enough cores to match the number of application threads. The only difference was that the queue sizes were set to infinity.[5] Figure 8 presents the speedup obtained from the different multithreaded configurations relative to single-threaded in-order execution. Figure 9 shows the execution time breakdown of each thread in the different multithreaded code versions with infinite queue sizes normalized to single-threaded in-order execution for benchmarks `wc` and `ammp`. Notice that the *EXE* component of downstream threads has reduced significantly in Figure 9 compared to the breakdown presented in Figure 5.

[5]A size of 10000 was sufficient to ensure that no queue full/empty stalls occurred.

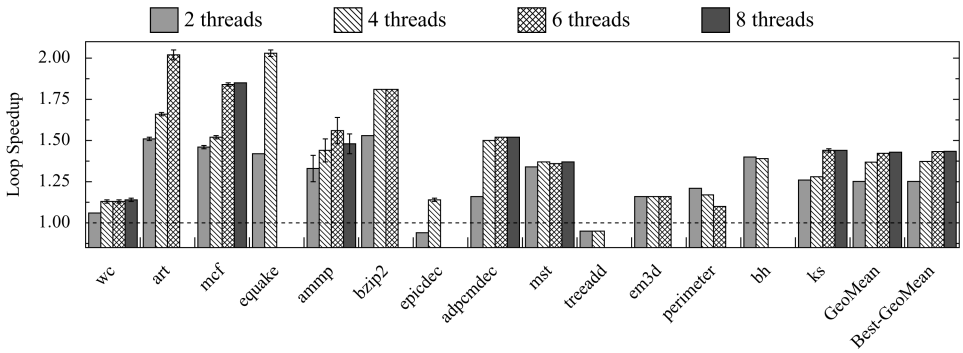


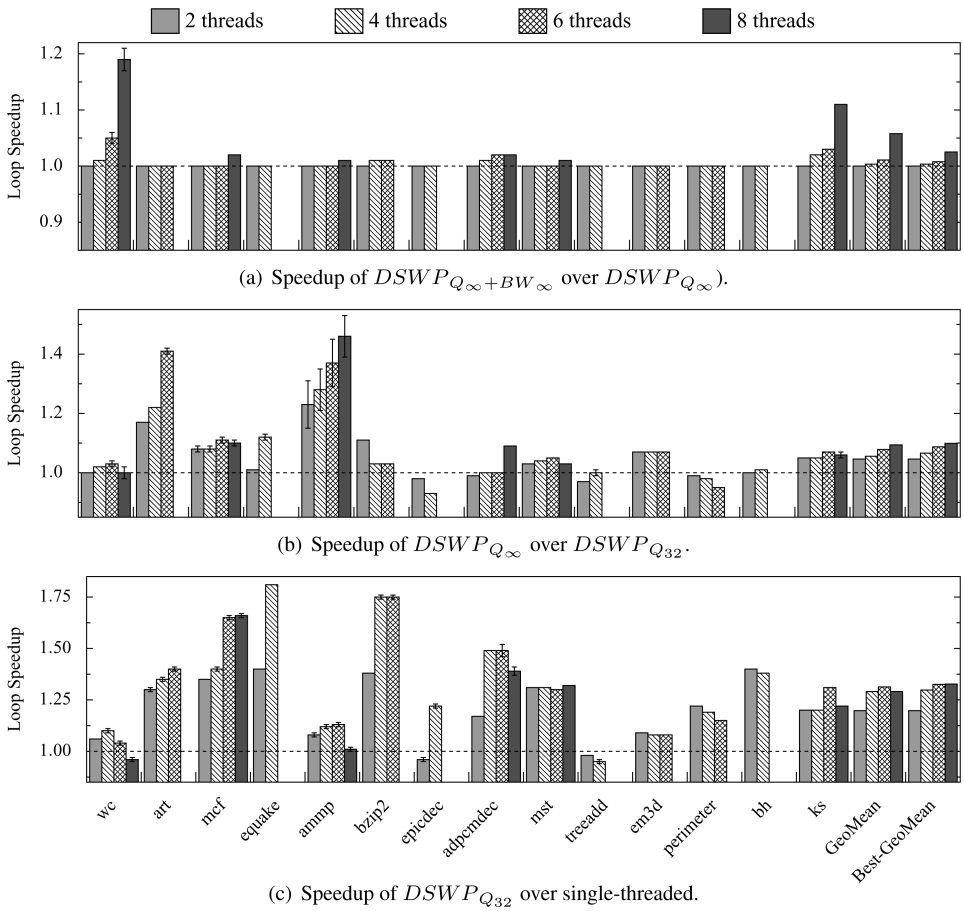
Fig. 10. $DSWP_{Q_\infty+BW_\infty}$: Performance of DSWP with two, four, six, and eight threads with infinitely long queues and an infinite communication bandwidth relative to single-threaded in-order execution.

Several observations are in order. As expected, easing the queue-size limitation does alleviate the communication bottleneck imposed by nonlinear pipelines and improves DSWP performance for most benchmarks when moving to more threads. The geometric mean speedups of $DSWP_{Q_\infty}$ with two, four, six, and eight threads are 1.25X, 1.36X, 1.41X, and 1.39X, respectively, whereas the geometric mean speedups of $DSWP_{Q_{32}}$ with two, four, six, and eight threads from Figure 4 were 1.20X, 1.29X, 1.31X, and 1.29X, respectively.

Despite the overall improvement, there are several notable exceptions. Benchmarks `wc`, `mcf`, `ammp`, `perimeter`, and `ks` continue to see a performance degradation when moving to more threads. A closer look at the execution revealed that the arbitration policy of the bus interconnect carrying synchronization array traffic, always favored earlier threads. This caused threads later in the pipeline to suffer arbitration stalls in six- and eight-thread scenarios. The removal of the bottleneck because of pipeline nonlinearity with infinitely long queues resulted in producer threads earlier in the pipeline being greatly sped up, causing instructions from threads later in the pipeline to suffer interconnect contention stalls. In particular, when these stalls hit `consume` instructions, which were at the head of dependence chains, of downstream threads, performance slowdown was inevitable.

4.3 Relaxing Queue-Size *and* Bandwidth Constraints

In order to alleviate the finite bandwidth limitations to the synchronization array, the bus interconnect was replaced with a crossbar interconnect and the synchronization array was allowed to have as many ports as required to cater to requests from all cores every cycle. This configuration with infinite queue sizes and infinite communication bandwidth is labeled as $DSWP_{Q_\infty+BW_\infty}$. This idealization made the eight-thread versions of benchmarks `wc`, `mcf` and `ks` perform no worse than the six-thread versions. From Figure 10, which presents the overall speedup obtained by $DSWP_{Q_\infty+BW_\infty}$ relative to single-threaded in-order execution, the geometric mean speedups of $DSWP_{Q_\infty+BW_\infty}$ across two-, four-, six-, and eight-thread versions can be seen to be 1.25X, 1.37X, 1.42X, and 1.43X, respectively. The infinite interthread communication bandwidth remedies the



(a) Speedup of $DSWP_{Q_\infty+BW_\infty}$ over $DSWP_{Q_\infty}$).

(b) Speedup of $DSWP_{Q_\infty}$ over $DSWP_{Q_{32}}$.

(c) Speedup of $DSWP_{Q_{32}}$ over single-threaded.

Fig. 11. Relative performance of DSWP with two, four, six, and eight threads under different communication scenarios.

performance degradation for `wc` seen in Figure 9 and causes `wc` to fall in line with theoretical expectations. Similar improvements can also be seen for benchmarks `mcf`, `mst`, and `ks`.

Figure 11 presents the relative speedup graphs for all three communication scenarios—$DSWP_{Q_{32}}$, $DSWP_{Q_\infty}$, and $DSWP_{Q_\infty+BW_\infty}$. It highlights the incremental improvement obtained by easing just the queue size bottleneck relative to $DSWP_{Q_{32}}$ (Figure 11b) and by easing both the queue size, as well as the interconnect contention bottlenecks (Figure 11a) relative to $DSWP_{Q_\infty}$.

Notice that `ammp` and `perimeter` continue to experience performance degradation even after elimination of interconnect contention stalls. Detailed analysis revealed that in `perimeter`, the best balance is achieved with two threads, with thread 1 being the bottleneck thread. Since thread 1 contained only 1 SCC, further parallelization is made possible by only by spreading thin the remaining SCCs among more threads. However, this leads to loss of locality in data accesses leading to increased coherence activity, thereby increasing the number of coherence-induced misses in the L2 caches. False sharing occurs due

to different threads accessing different fields of the same structure. A similar problem is observed in `ammp` as well when moving from six to eight threads. The best balance is obtained with six threads. Moving to eight threads only results in the creation of more coherence traffic leading to performance slowdown. Performance loss resulting from false sharing can be avoided by using clever, DSWP-aware data layout schemes. We leave such optimizations for future work.

Finally, note that the best geometric mean speedup, which represents the maximum speedup possible with N cores (with number of threads fewer than or equal to N), for two, four, six, and eight threads improves from 1.20X, 1.30X, 1.32X, and 1.33X in the $DSWP_{Q_{32}}$ configuration to 1.25X, 1.37X, 1.43X, and 1.43X, respectively, in the $DSWP_{Q_\infty + BW_\infty}$ scenario. The overall improvement from elimination of communication bottlenecks clearly demonstrates that it is not enough for the compiler to be intelligent enough to pick the best multithreaded code version, but that it must also strive to eliminate nonlinear pipelines during partitioning, in order to achieve optimal performance.

5. CONCLUSIONS AND FUTURE WORK

This paper analyzed the performance scalability of automatically generated DSWP codes. It demonstrated that superpartitioning of application loops leads to complex inter-thread communication DAGs and that such DAGs interact pathologically with the underlying communication substrate, adversely affecting performance. Analytical expressions derived in this paper can be incorporated into an *autoDSWP* compiler's partitioning heuristic to statically determine whether a given DSWP partition would lead to these communication pathologies and repartition the code, if needed. Without any communication bottlenecks, DSWP delivers a geometric mean speedup of 1.25X to 1.43X when going from two to eight threads across a variety of benchmarks.

While the speedup achieved by DSWP in innermost and second innermost loops were presented in this paper, initial results from work in progress indicate that there is good potential for pipelined parallelism at outer loop nest levels. Future work will focus on evaluating DSWP's scalability at coarser granularities and improving our *autoDSWP* compiler's partitioning heuristic based on results from this paper.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. Their critical comments helped improve the focus and the quality of the presentation. Special thanks to Matthew Bridges for help with the Velocity compiler infrastructure.

REFERENCES

BARNES, R. D., NYSTROM, E. M., SIAS, J. W., PATEL, S. J., NAVARRO, N., AND HWU, W. W. 2003. Beating in-order stalls with "Flea-Flicker" two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture*.

BARNES, R. D., RYOO, S., AND HWU, W. W. 2005. "Flea-Flicker" multipass pipelining: An alternative to the high-powered out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture*. 319–330.

CHAPPEL, R. S., STARK, J., KIM, S. P., .REINHARDT, S. K., AND PATT, Y. N. 1999. Simultaneous subordinate microthreading. In *Proceedings of the 26th International Symposium on Computer Architecture*. 186–195.

COLLINS, J. D., WANG, H., TULLSEN, D. M., HUGHES, C., LEE, Y.-F., LAVERY, D., AND SHEN, J. P. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*.

CYTRON, R. 1986. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*. 836–884.

DAI, J., HUANG, B., LI, L., AND HARRISON, L. 2005. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 237–248.

GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. 2002. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 291–303.

HAMMOND, L., HUBBERT, B. A., SIU, M., PRABHU, M. K., CHEN, M., AND OLUKOTUN, K. 2000. The Stanford Hydra CMP. *IEEE Micro 20*, 2, 71–84.

INTEL CORPORATION. 2002. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. 330–335.

LUNDSTORM, S. F. AND BARNES, G. H. 1980. A controllable MIMD architecture. In *Proceedings of the International Conference on Parallel Processing*. 19–27.

LUO, Y. AND JOHN, L. K. 2004. Efficiently evaluating speedup using sampled processor simulation. *Comput. Architect. Lett.*

OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*.

PADUA, D. A. 1979. Multiprocessors: Discussion of some theoretical and practical problems. Tech. Rep. UIUCDCS-R-79-990 (Nov.). Department of Computer Science, University of Illinois, Urbana, IL.

PENRY, D. A., VACHHARAJANI, M., AND AUGUST, D. I. 2005. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*.

RANGAN, R., VACHHARAJANI, N., VACHHARAJANI, M., AND AUGUST, D. I. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 177–188.

RANGAN, R., VACHHARAJANI, N., STOLER, A., OTTONI, G., AUGUST, D. I., AND CAI, G. Z. N. 2006. Support for high-frequency streaming in CMPs. In *Proceedings of the 39th International Symposium on Microarchitecture*. 259–269.

RO, W. W., CRAGO, S. P., DESPAIN, A. M., AND GAUDIOT, J.-L. 2006. Design and evaluation of a hierarchical decoupled architecture. *J. Supercomput. 38*, 3 (Dec.), 237–259.

ROTH, A. AND SOHI, G. S. 2001. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Achitecture*.

SMITH, J. E. 1982. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*. 112–119.

SOHI, G. S., BREACH, S., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*.

STEFFAN, J. G., COLOHAN, C., ZHAI, A., AND MOWRY, T. C. 2005. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst. 23*, 3, 253–300.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*.

TRIANTAFYLLIS, S., BRIDGES, M. J., RAMAN, E., OTTONI, G., AND AUGUST, D. I. 2006. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. 61–71.

TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D. J., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Trans. Comput. 48*, 9, 881–902.

VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. 2002. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture.* 271–282.

VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2004. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI).* 195–206.

VAJAPEYAM, S. AND MITRA, T. 1997. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ACM Press, New York. 1–12.

WANG, P. H., WANG, H., COLLINS, J. D., GROCHOWSKI, E., KLING, R. M., AND SHEN, J. P. 2002. Memory latency-tolerance approaches for Itanium processors: Out-of-order execution vs speculative precomputation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. 187–196.

WENISCH, T. F., WUNDERLICH, R. E., FALSAFI, B., AND HOE, J. C. 2004. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Tech. Rep. 2004-003 (Nov.). Computer Architecture Lab at Carnegie Mellon.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. 84–97.

ZHOU, H. 2005. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*.

Received March 2007; revised August 2007; accepted January 2008