Performance Evaluation of Dynamic Speculative Multithreading with the Cascadia Architecture

David A. Zier, *Member*, *IEEE*, and Ben Lee, *Member*, *IEEE Computer Society*

Abstract—Thread-level parallelism (TLP) has been extensively studied in order to overcome the limitations of exploiting instruction-level parallelism (ILP) on high-performance superscalar processors. One promising method of exploiting TLP is Dynamic Speculative Multithreading (D-SpMT), which extracts multiple threads from a sequential program without compiler support or instruction set extensions. This paper introduces Cascadia, a D-SpMT multicore architecture that provides multigrain thread-level support and is used to evaluate the performance of several benchmarks. Cascadia applies a unique sustainable IPC (sIPC) metric on a comprehensive loop tree to select the best performing nested loop level to multithread. This paper also discusses the relationships that loops have on one another, in particular, how loop nesting levels can be extended through procedures. In addition, a detailed study is provided on the effects that thread granularity and interthread dependencies have on the entire system.

Index Terms—Multithreading processors, multicore processors, simulation, speculative multithreading.

1 Introduction

Superscalar techniques that exploit *instruction-level parallelism* (ILP) have been proven to be a cornerstone in advancing the state of the art in high-performance processors. However, the performance of monolithic processors has been limited by true and control flow dependencies inherent in single-threaded execution, along with the complexities of supporting large instruction windows and many in-flight instructions. In order to overcome these limitations and effectively utilize the increasing transistor count and die space, chip makers have focused on *Multicores* or *Chip Multiprocessors* (CMPs) that combine several processor cores on a single die to exploit *Thread-Level Parallelism* (TLP). However, properly utilizing the increasing processing power has been a major challenge since programmers have to reason about parallelism and deal with communication and synchronization issues.

Thread-Level Speculation or *Speculative Multithreading* (SpMT) is a promising technique for speeding up single-threaded applications without the need for a programmer to explicitly define TLP. SpMT achieves this using software and/or hardware methods to exploit TLP that exists in single-threaded programs. TLP can be extracted by creating threads from loop boundaries [1], [2], [3], function continuations [4], code reconvergence [5], and/or acyclic software pipelining [6], [7]. SpMT can be done either statically or dynamically. In static SpMT, programs are processed by either a parallelizing compiler or a binary annotator to find optimal code segments for threads [4], [8], [9]. Then, new instructions are inserted into the code that spawn/join threads and handle interthread data dependencies using hardware support. Unfortunately, static methods have difficulty taking advantage of the dynamic aspects of programs. As such, they may disregard large segments of a program, where dependencies cannot be resolved statically. In contrast, *dynamic* SpMT (D-SpMT) relies only on hardware mechanisms to handle the thread management and data dependencies at runtime [1], [2], [3], [10], [11], [12]. Because D-SpMT relies only on hardware to detect, spawn, and manage threads, it does not require any special instruction set extensions or intervention from the Operating System.

There have been several D-SpMT architectures proposed in the literature, but the rationale for their performance requires a more detailed study. For example, what is the proportion of time a program spends speculatively multithreading? What typical thread characteristics provide the best performance gains? What are the effects of interthread dependency misspeculation? Therefore, this paper presents a detailed analysis to answer these questions. The main contributions of the paper are as follows:

- First, this paper introduces a multigranular D-SpMT architecture called *Cascadia*. Cascadia exploits threads from loop segments. Loop selection is performed with a unique loop tree structure that not only reflects the relationship among loops, but also the relationship between procedure calls and loops, and uses a heuristic based on past performance indicators. This allows Cascadia's multigrain approach to provide a more accurate loop-level selection of multinested loops and achieve greater performance than either fine-grain or coarse-grain approaches.

- *D.A. Zier is with the NVIDIA Corporation, 20400 NW Amberwood Dr. #100, Beaverton, OR 97006. E-mail: dzier@nvidia.com.*
- *B. Lee is with the School of Electrical Engineering and Computer Science, Oregon State University, 3117 Kelley Engineering Center, Corvallis, OR 97331. E-mail: benl@eecs.oregonstate.edu.*

Manuscript received 15 Sept. 2008; revised 7 Jan. 2009; accepted 2 Mar. 2009; published online 12 Mar 2009.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-09-0353.
Digital Object Identifier no. 10.1109/TPDS.2009.47.

- Second, a comprehensive analysis is performed to better understand the dynamic relationships of loops. There already have been several studies on loop behavior and characteristics for the purpose of extracting TLP [13], [14], [15], [16], [17]. However, these studies only consider control flow through a single loop in order to develop heuristics that determine the best spawning point. To the best of our knowledge, this paper is the first to examine the relationships that loops have on each other in order to dynamically select the most appropriate nesting level.
- Third, the effect that interthread dependency misspeculations have on performance is presented. This study also shows the potential performance gains by ideally predicting interthread dependencies. The purpose is to show the effects of specific dependencies on the entire system and how relationships between loops change under ideal conditions.

Our simulation studies of Cascadia show significant performance improvement on multimedia applications with an average speedup of 1.5 with one benchmark achieving a speedup of over 5 with 16 cores. Programs with large, well-structured loops will obviously perform well, but the results show that Cascadia can also handle very dynamic loops for positive gains. The results also show that some integer and floating-point benchmarks contain an insufficient number of loops necessary to achieve the same type of gains from multimedia benchmarks.

The remainder of this paper is organized as follows: Section 2 discusses how loops are multithreaded and includes a discussion on loop structure and the multigrained loop selection heuristic. A brief discussion on the implementation of the Cascadia architecture is provided in Section 3. This is followed by a comprehensive analysis of the simulation results in Section 4. Related work is discussed in Section 5 and future work in Section 6. Finally, Section 7 concludes the paper.

2 Multithreading Loops

Unlike static SpMT techniques, D-SpMT must determine loops and their dependencies during the execution of a single instruction stream or program flow. In addition, a multigrain D-SpMT system must dynamically determine the nested relationships of those loops in order to select an appropriate nested loop level.

2.1 Loop Structure

Within an instruction stream, a loop is defined as the instructions contained within a backward-branch instruction, *BPC*, and the target PC of the branch, *TPC*. This definition of a loop is sufficiently general for D-SpMT to detect all the executed loops within an application. In addition, loop A is an inner loop of loop B *iff* $TPC_B \leq BPC_A < BPC_B$. This definition covers all nested loops, including the case of multiple backward edges for a single loop. Moreover, a function is defined by the TPC of the function call and the return PC, *RPC*. Nested loops through function calls are also recognized in order to create an accurate representation of the loop structure.

The structure of loops and functions are inherently hierarchical in nature and they are often intertwined. Therefore, the *Loop Tree* is used to represent the relationship among multiple nested loops and function calls, and the TPC:RPC pair is pushed onto the *Function Stack* to indicate the current instance of a function call within the Loop Tree.

Fig. 1 demonstrates how the Loop Tree can dynamically be created from an instruction stream. Fig. 1a shows a fairly complex instruction stream with multiple nested loops and function calls that may exist in many integer and scientific programs. The instruction stream consists of four nested function calls **F1-F4** and seven unique loops **L1-L7**.

L6 is the first encountered loop as the instruction stream propagates down to function $F4$. After **L6**, **L7** is discovered as a sibling to **L6** since neither loop is within the loop boundaries of one another. Eventually, **L5** is found to encompass both **L6** and **L7**. The state of the Loop Tree after $F4$ completes is illustrated in Fig. 1b. Upon the return of $F4$ to the RPC $R4$, the instruction stream encounters loop **L3**. Since the TPC of **L3** is greater than $R4$, **L3** becomes a sibling to the **L5** Loop Tree. Afterward, another function call is made to $F4$, and again, the same Loop Tree in Fig. 1b is found. It is essential to note that this is a new instance of the **L5** Loop Tree since $R5$ is different from $R4$ and is a sibling to the prior **L5** Loop Tree and **L3** loop. By the time the BPC of loop **L2** is discovered, the Loop Tree contains the three instances of the **L5** Loop Tree, the **L3** loop, and the **L4** loop as equal siblings. **L2** will then encompass all the siblings, since every BPC and RPC for each of the siblings are within the loop boundaries of **L2**. The state of the Loop Tree at the end of function $F3$ is shown in Fig. 1c. After the instruction stream returns to function $F2$ at $R2$, another call to $F3$ is encountered, and consequently, a new instance of the **L2** Loop Tree will be found. Finally, the loop **L1** is found to encompass the RPC values of both **L2** sibling Loop Trees and ultimately becomes the outermost loop within the whole tree structure. Fig. 1d illustrates the entire, static view of the loop hierarchy upon the completion of the function $F2$.

The above example shows the importance of considering both loops and function calls for detecting threads. If function calls were not considered, the system would see three separate Loop Trees that have no correlation with each other and the loop selection mechanism would fail to find the best nested level, and ultimately, the best performance gain. The final tree structure of the above example consists of 7 unique loops with 25 loop nodes and a maximum depth of 4 levels.

2.2 Loop-Level Selection

A loop must meet the following two requirements to maximize TLP performance: 1) a loop's multithreaded performance must be greater than or equal to its single-thread performance and 2) a loop's multithreaded performance must be greater than or equal to the performance of any of its nested inner loops. In order to determine if these two requirements are met, the selection heuristic uses three IPC performance indicators: 1) a *Pre-Multithreading IPC* (*pIPC*) derived from the sequential run of an iteration as a single thread, 2) a *Multithreaded IPC* (*mIPC*) derived from a loop while it is being multithreaded, and 3) a *Sustainabe IPC* (*sIPC*), which is an *approximated IPC* for the current loop if

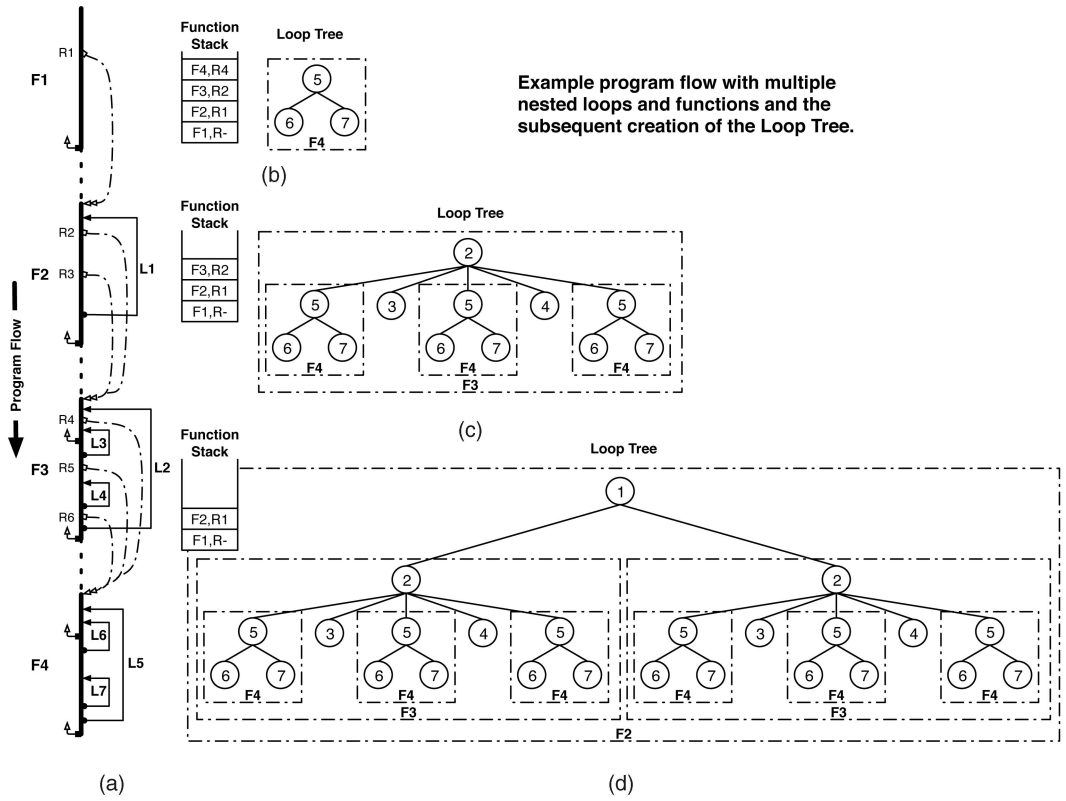

Fig. 1. (a) Example of a program flow with multiple nested loops embedded within functions, (b) the loop tree after function **F4** completes, (c) the loop tree after function **F3** completes, and (d) the loop tree after function **F2** completes.

its inner loops were multithreaded. Ultimately, $mIPC$ of a loop must be better than $sIPC$, i.e., $mIPC \geq sIPC$, in order to *sustain* a beneficial gain in performance.

In order to estimate $sIPC$ for the current loop, $pIPC$ and $mIPC$ values from all the inner loops are weighted relative to the amount of time spent in a single iteration of the current loop. The $sIPC$ metric has the general form

$$sIPC = \sum_i \alpha_i \cdot mIPC_i + \left(1 - \sum_i \alpha_i\right) \cdot pIPC_o, \quad (1)$$

where α_i represents the fraction of a single iteration of the current loop code segment that is covered by the ith nested inner loop, $mIPC_i$ is the $mIPC$ value of the ith nested inner loop, and $pIPC_o$ is the $pIPC$ value of the current loop.

Equation (1) holds true if all nested inner loops performed well while multithreading. However, this equation does not account for the situation, where a nested inner loop fails the $mIPC \geq sIPC$ requirement. In this case, the $sIPC$ value of the inner loop will be used instead of the $mIPC$ value. Therefore, a more advanced form of the $sIPC$ metric is shown below:

$$sIPC = \sum_i \alpha_i \cdot \Phi_i + \left(1 - \sum_i \alpha_i\right) \cdot pIPC_o, \quad (2)$$

where

$$\Phi_i = \begin{cases} mIPC_i, & \text{for } mIPC_i \geq pIPC, \\ sIPC_i, & \text{for } mIPC_i < pIPC. \end{cases}$$

As can be seen, (2) exhibits a recursive pattern when an inner loop needs its own $sIPC$ value. Fortunately, based on the definition of a loop presented in Section 2.1, inner loops are always found and processed before any outer loops, which means that the $sIPC_i$ value has already been calculated before being needed in the outer loop's $sIPC$ evaluation.

3 The Cascadia Architecture

This section presents *Cascadia*, a multicore processor capable of multigranular D-SpMT. Fig. 2a shows the architectural diagram of Cascadia, which consists of a *Central Processing Element* (CPE) and multiple *Processing Elements* (PEs). Cascadia utilizes four major communication buses as follows:

1. Control Bus,
2. Interthread data bus (ITBus),
3. Program Memory bus, and
4. Data Memory bus.

The bidirectional *Control Bus* facilitates communication between the CPE and the individual PEs. Its main role is to transmit commands and interthread register dependency information (see Section 3.3.2).

The *IT Bus* facilitates communication between neighboring PEs in a ring network. The bulk of all register data is transferred through this bus, one register element at a time, and requires a 32-bit unidirectional data bus. In addition, the IT Bus consists of a small bidirectional bus used to transmit the requests and commands from neighboring PEs along with the register address.

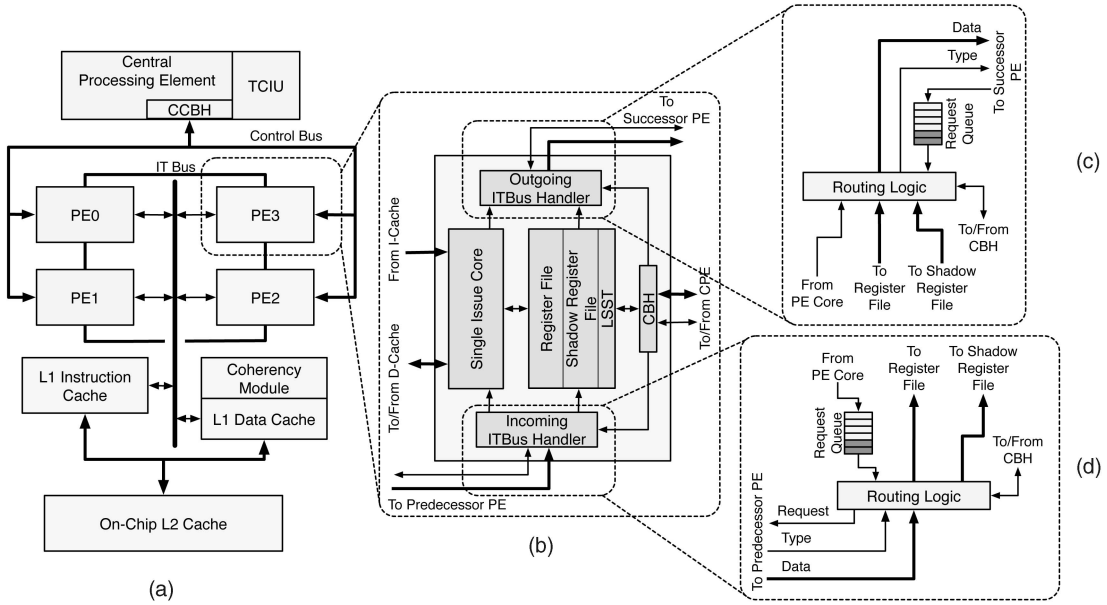

Fig. 2. Block diagram of the Cascadia architecture: (a) 4-PE system overview, (b) datapath for a single PE, (c) details of the outgoing ITBus handler, and (d) details of the incoming ITBus handler.

The *Program Memory* and *Data Memory* buses connect each PE to the instruction and data cache, respectively. Each cache is multiported and every PE has a direct connection to a single port on the instruction and data caches. This shared L1 cache scheme has been used on the SPARC Processor [18] and allows each PE to have equal access to the caches without conflict or complex cache prioritization mechanisms. The data cache contains a cache coherency module that keeps track of speculative loads and alerts the PEs to any interthread memory dependency violations. Cascadia does not employ a mechanism to directly transfer memory data from one PE to another, nor does Cascadia allow stores from speculative threads to be written to memory.

3.1 Operating Modes

Cascadia has three modes of operation: 1) *Normal*, 2) *Pre-SpMT*, and 3) *SpMT*. In the Normal mode, the program runs sequentially on a single PE while the CPE monitors all branch instructions to identify loop candidates for D-SpMT.

Once a loop is found and selected as a candidate, Cascadia transitions to the Pre-SpMT mode. In this mode, the program continues to execute on a single PE, but now the TCIU and CPE gather information necessary for SpMT, including interthread register dependency speculation values, $pIPC$ values, and initial calculations for the $sIPC$ value. The system stays in the Pre-SpMT mode for the first three iterations of a loop so that loop strides can be observed and confirmed, as well as gathers a more accurate $pIPC$ value for the loop. Our research shows that three iterations are sufficient to gather the loop stride information. Fewer iterations result in less accurate register stride and $pIPC$ values. Additional iterations provide better $pIPC$ values, but also increase the delay from the time a loop is found to when it is multithreaded, which reduces the amount of improvement from SpMT.

After Cascadia completes the prerequisite number of iterations, it transitions to the SpMT mode where subsequent iterations of the loop are spawned and executed across all the PEs. While in the SpMT mode, only the first, *head PE* is nonspeculative and holds the true state of the system, while all other PEs are speculative and will be squashed on a register value misprediction or an interthread dependency violation. When this occurs, the PE that caused the violation and all of its successor PEs are squashed and respawned.

3.2 Processing Elements

An overview of a single PE is shown in Fig. 2b. A PE consists of a single-issue processor, a pair of register files, and various communication bus handlers. Register data are transferred from one PE to another over the ITBus network. The *Outgoing ITBus Handler* and *Incoming ITBus Handler* handle all register data traffic to and from the PE as illustrated in Figs. 2c and 2d, respectively. The role of the ITBus is twofold. First, during all modes of operation, all register writes within the nonspeculative head thread propagate through the ITBus and are written to the shadow register file on each PE. Also, during Normal mode, branch data are propagated through the ITBus so that each PE starts with the correct branch table history. Second, a *request* command can be sent on the ITBus to read a register value form a predecessor PE.

The *CommandBus Handler* (CBH) provides the necessary routing logic and queues to communicate with the CPE. The CBH handles squash and spawn requests, transmits and receives register file utility bit information, and sends branch information to the CPE.

Each PE has a pair of register files, one to maintain the current state of the PE and another *shadow register file* that maintains the current nonspeculative state. This pair of register files is connected in a three-dimensional fashion [19] so that when a thread is spawned on the PE, the register data in the shadow register file can be copied to the register

file in a single cycle. This allows each speculative PE to start with the most up-to-date, nonspeculative state.

A *Loop Stride Speculation Table* (LSST) is connected to both register files and calculates the strides for all integer registers. A stride is calculated by finding the difference from the current and previous register data. A stride is considered good if two calculations of a stride match and any future references to that register will have the stride value added to the current register value. During the SpMT mode, the strides are calculated from the register file as the PE executes its thread, while during the Pre-SpMT mode, the strides are calculated from the shadow register file. This allows the LSST to match the nonspeculative state of the head PE prior to the spawning of a thread.

Interthread register prediction occurs at two levels. In the first level, a PE will request a register data through the ITBus by sending a request to the Incoming ITBus Handler. The predecessor PE will then receive the request in the Outgoing ITBus and return the register data to the successor PE. This register transfer requires a minimum of two cycles to send the request and receive the data, but could take up to 10 cycles or more depending on how full the respective request queues are. In addition, the predecessor PE will wait up to 30 cycles for the data to be written. If the data have not been written within this time period, it will send an invalid command back to the successor PE. At this point, the PE will perform the second level interthread register prediction by getting the immediate register data of the nonspeculative head PE. Since the shadow register file always has the current state of the nonspeculative head PE, the register data are simply obtained from the shadow register file.

3.3 Central Processing Element

The CPE consists of a multiple-issue, superscalar processor core, the *Thread Control and Initialization Unit* (TCIU), and the *Central Control Bus Handler* (CCBH). The processor core runs a helper thread to manage the Loop Tree and contains its own on-chip cache to store this information. The TCIU is responsible for managing threads and handling interthread register dependencies. Furthermore, the CCBH facilitates all communication between the CPE and the PEs over the Control Bus.

3.3.1 Helper Thread

The CPE executes a helper thread [20] that creates and maintains the *Loop Tree*, stores loop performance indicators, calculates the $sIPC$ value, and ultimately decides whether or not a loop will be multithreaded. When Cascadia is in the Normal mode, the CPE passively listens for a TPC:BPC pair from a recently executed backward branching instruction or a TPC:RPC pair from a function call. When a TPC:RPC pair is detected, it is pushed onto to the Function Stack and used to index into a particular node within the Loop Tree. When a TPC:BPC pair is detected, one of the two following events occurs: 1) if the pair is from a newly discovered loop, a new entry is created and inserted in the Loop Tree or 2) if an entry already exists for the loop, then the performance indicators for the loop are retrieved and used to determine whether it should be multithreaded again. In either case, the CPE alerts the TCIU to begin the multithreading process and the system transitions to the Pre-SpMT mode, where the CPE monitors the number of instructions and cycles on the Head PE in order to calculate the $pIPC$ and $sIPC$ for the loop.

Once Cascadia enters the SpMT mode, the CPE gathers all committed instructions from both the nonspeculative and speculative PEs to calculate the $mIPC$ for the loop. After a thread completes, the final $sIPC$ value is calculated using the heuristic discussed in Section 2.2 and compared against the current $mIPC$ value. If $mIPC$ falls below $sIPC$, then the CPE stops multithreading the loop and Cascadia transitions back to the Normal mode.

3.3.2 Thread Control and Initialization Unit

The TCIU is responsible for squashing and spawning threads, managing the modes of operation, and handling interthread register dependencies. In order to start a thread, a PE needs the following:

1. nonspeculative register data,
2. branch history data,
3. TPC:BPC pair, and
4. interthread register dependency information.

The register and branch history data are propagated to the PEs from the nonspeculative head PE along the ITBus (see Section 3.2). On the other hand, the TCIU sends both the loop boundaries and dependency information to each PE through the Control Bus. The interthread register dependency information is represented as *utility bits* within the TCIU and PE register files [2].

The TCIU will simultaneously send the TPC:BPC loop data to all PEs when entering the Pre-SpMT mode. When spawning a thread in the SpMT mode, the TCIU sends dependency utility bits as a single vector to a PE requiring a 1-cycle delay. Threads are spawned in consecutive cycles to allow for the nonspeculative register data to propagate through the ITBus. When a thread completes, the PE will send a *retire* signal to the TCIU along with the current instruction and cycle counts. When the PE becomes the nonspeculative head, the thread is retired and these counts are sent to the CPE to calculate the $sIPC$ value. Squashing a thread requires the TCIU to send a squash command to the appropriate threads.

4 Performance Evaluation

The performance evaluation of D-SpMT was done on a cycle-accurate, execution-based simulator built on the NetSim Architectural Simulator Suite [21]. The simulator replicates the behavior of Cascadia as described in Section 3 using SimpleScalar's PISA instruction set [22]. The configuration parameters used in the simulation study are listed in Table 1. The parameters associated with the PEs and cache structure were chosen to provide a comparative analysis with closely related works [2], [9], [12], [15], [23].

A selection of benchmarks from SPEC CPU 2000 [24] and MediaBench [25] were used in this evaluation. In order to reduce the simulation time, the SPEC benchmark results were based on the reduced data sets from MinneSPEC [26]. All benchmarks were compiled with SimpleScalar's [22] GCC PISA cross-compiler with -O2 optimizations and run in their entirety to properly reflect their performance.

TABLE 1
Cascadia Simulation Parameters

Architectural Parameters	
IT Bus Latency	1 cycle
Control Bus Latency	1 cycle
Spawn Latency	1 cycle propagated
1st-Level Prediction	2–10 cycles
2nd-Level Prediction	1 cycles
Individual Processing Elements	
Fetch Width	1 inst./cyc.
Issue Width	1 inst./cyc.
Inst. Queue Size	4
ROB Size	16
Exec. Units	2 IALU, 1 Mul/Div, 1 FPALU, 1 Branch
Load/Store Unit	Load Forwarded, 32-entry Store Queue
Memory	
L1 ICache	32KB, 4-way, 1 cycle hit latency
L1 DCache	32KB, 4-way, 1 cycle hit latency
L2 Unified Cache	128KB, 8-way, 8 cycle hit latency
Memory	60 cycle hit latency

Benchmarks from SPEC CPU 2000 (SPEC2000) and MediaBench that have a variety of parallelism and dynamic behaviors and that could be compiled with the GCC PISA cross-compiler were selected. Out of the SPEC2000 integer benchmarks, `176.gcc` was chosen because it exhibited very dynamic behavior with little inherent parallelism. `181.mcf` and `164.vpr` are integer benchmarks with more predictable parallelism, while `183.equake` and `188.ammp` were selected as representatives of the two extremes in TLP (parallel and dynamic, respectively) for the SPEC2000 floating-point benchmarks. Out of the MediaBench benchmarks, `MPEG2`, `DJPEG`, and `UNEPIC` were used since they represent the most popular types of multimedia programs. `EPIC` exhibited a lot of TLP while both `MESA` and `Pegwit` are two multimedia benchmarks that had little TLP.

Fig. 3 illustrates the overall performance for 2, 4, 8, and 16 PEs. The SPEC2000 results were rather mixed with an average speedup of 1.01 with 2 PEs to 1.04 with 16 PEs. On the other hand, the MediaBench benchmarks performed well with an average speedup of 1.2 with 2 PEs to 1.75 with 16 PEs, with `EPIC` providing a speedup of more than 5 with 16 PEs. Removing the bias from `EPIC`, the MediaBench benchmarks still fared well with an average speedup of 1.09 with 2 PEs to 1.2 with 16 PEs.

In general, Fig. 3 shows that there is a linear increase in performance as the number of PEs increases. Yet these trends have diminishing returns since the efficiency of each PE decreases as the number of PEs increases. This is due to the limited parallelism within the loops caused by inter-thread register and memory dependencies rather than Cascadia's ability to perform multithreading. This is illustrated by `181.mcf` and `EPIC`, whose performance scales well with the number of PEs.

4.1 Thread Characteristics

Fig. 4 shows a breakdown of all the unique loops and their characteristics. A *Good* loop indicates a loop that was speculatively multithreaded and resulted in a performance gain. The remaining loops failed to provide performance benefit for one of the following reasons: 1) an inner loop showed to have better performance, *Bad Nested*, 2) the loop failed to have a performance gain, *Bad sIPC*, or 3) the loop lacked sufficient iterations to complete the Pre-SpMT phase of Cascadia, *Pre-SpMT*. On average, about 25-30 percent of loops from SPEC2000 and 35-42 percent of loops from MediaBench proved to be beneficial for D-SpMT. The percentage of *Pre-SpMT* loops remains the same for all configurations, since this depends only on the compiled code and data set. The overall performance is ultimately limited by the percentage of *Bad Nested* and *Bad sIPC*. On average, 5 percent of all loops was marked as being *Bad Nested*. The average percentage of loops with *Bad IPC* starts around 28 percent for SPEC2000 and 32 percent for MediaBench and decreases as the number of PEs increases.

Fig. 4 shows that the MediaBench benchmarks have a higher percentage of *Good* loops. However, the high percentage of good loops does not necessarily guarantee an overall performance gain. For example, approximately 30 percent of all loops within `188.ammp` were good, yet the overall performance was negligible. This was also the case for `MESA`, even though more than half of its loops were considered favorable.

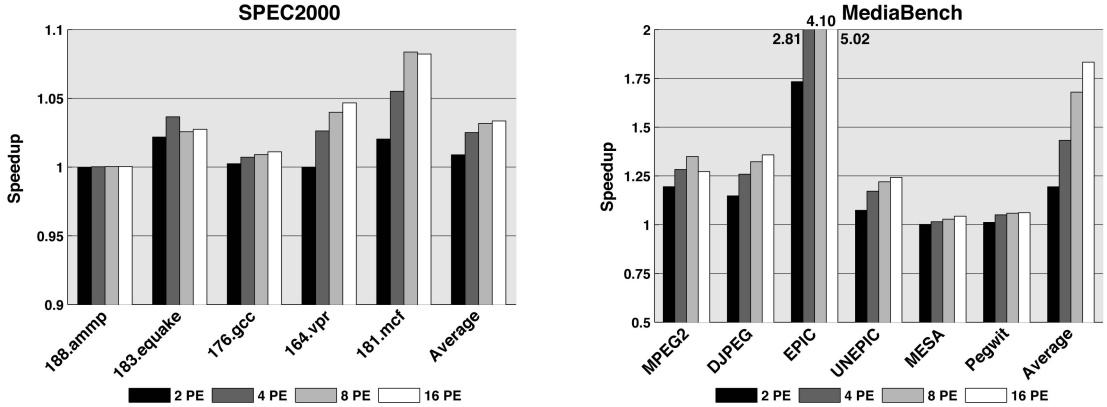

Fig. 3. Overall performance gains as function of number of PEs.

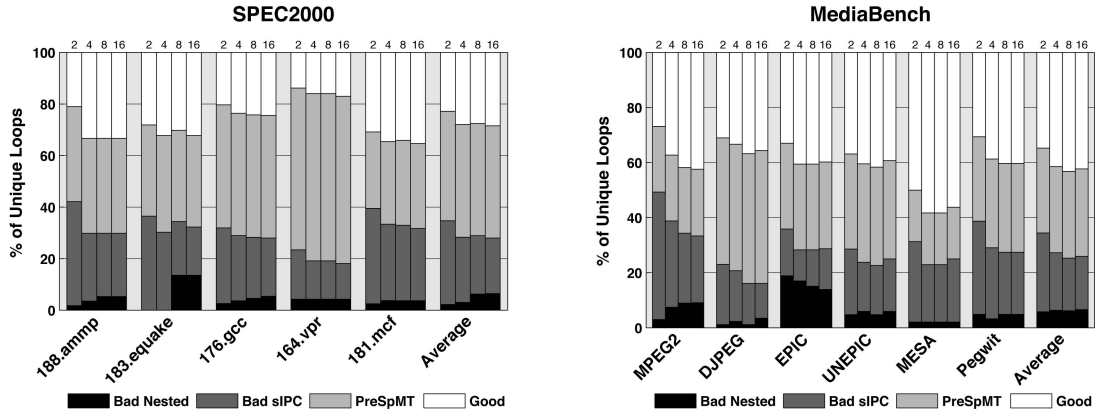

Fig. 4. A breakdown of unique loops.

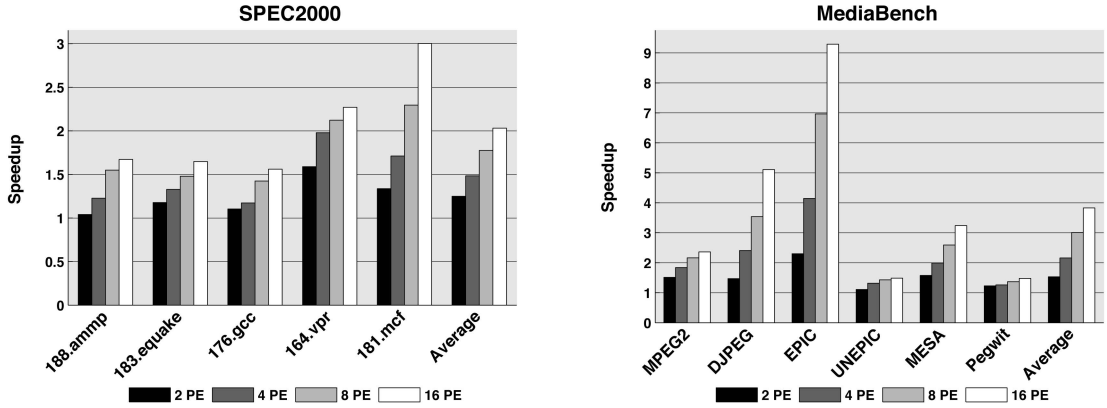

Fig. 5. Performance of Cascadia only in SpMT mode.

The low performance exhibited by many of the benchmarks in Fig. 3 is due to the fact that they have far too many code segments that cannot be multithreaded by D-SpMT. This subsequently increases the sequential portion of a benchmark and does not properly illustrate Cascadia's ability to perform D-SpMT. Therefore, Fig. 5 shows the speedup achieved only in the *SpMT* mode. These values were obtained by weighing the $pIPC$ and $mIPC$ values for each loop against the total number of instructions executed for that loop while in the *SpMT* mode. This resulted in an average speedup of 1.25 with 2 PEs to 2.0 with 16 PEs for the SPEC2000 benchmarks, while MediaBench had speedup of 1.5 with 2 PEs to 3.9 with 16 PEs. Even very hard to multithread programs such as `176.gcc` achieved a good performance with an average speedup of 1.1 with 2 PEs and 1.6 with 16 PEs.

Fig. 5 shows that Cascadia does a good job performing D-SpMT and it is the sequential portions of the benchmarks that limit the overall performance gains. Therefore, given more parallel applications and larger data sets, the overall performance of Cascadia would reflect the potential gains shown in Fig. 5. `EPIC` is an excellent example that shows the performance potential of Cascadia from large parallel programs.

Table 2 shows the number of unique loops found (#*Lps*), the average thread size (*Thread Size*), and the average number of threads per loop execution (#*Threads/Loop*). The average thread size corresponds to the average number of instructions per iteration, whereas the average number of threads per loop corresponds to the number of threads spawned from each instance of a loop. Programs such as `EPIC` and `UNEPIC` have both large thread sizes and a large number of threads per loop. Some benchmarks, such as `MESA`, contain loops with many iterations, but have small thread sizes. In contrast, `176.gcc` has large sized threads, but simply not enough threads per loop.

Table 2 also includes the number of nodes in the dynamically generated Loop Tree (#*Nds*), the average

TABLE 2
Loop Statistics

Benchmark	#Lps	Thread Size	#Threads /Loop	#Nds	Ave. NL	Max. NL
188.ammp	57	25	146.9	825	4	11
183.equake	96	301.4	4925.6	3,373	2.6	7
176.gcc	1024	168.9	19.2	5.6M	6.8	32
164.vpr	94	24.4	37.8	6,763	13.6	27
181.mcf	81	621.5	1590.5	2,777	2.6	6
MPEG2	67	156.3	73.8	2,283	3.5	9
DJPEG	87	44.8	39.5	767	2.6	8
EPIC	106	3475.9	5214.3	3,742	2.5	5
UNEPIC	84	28265.3	4434.1	3,251	2.6	5
MESA	48	37.7	7073.3	710	1.4	3
Pegwit	62	59.8	1011.4	1095	1.9	3

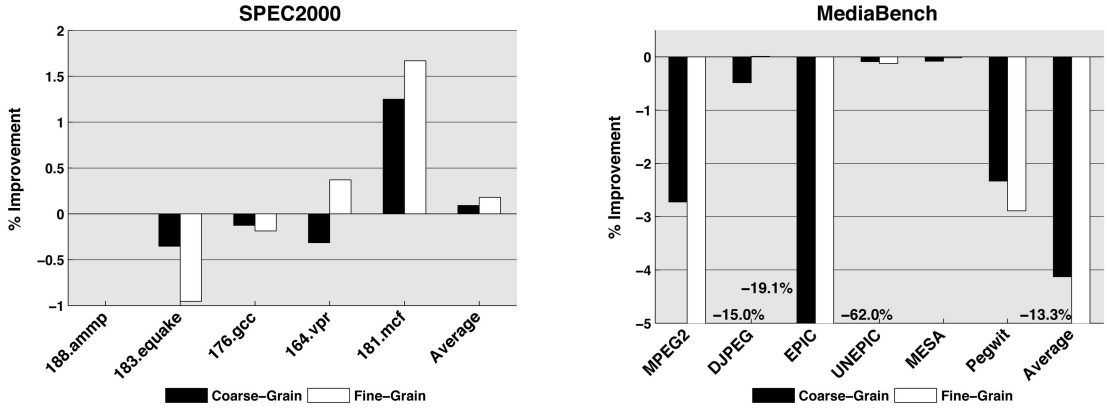

Fig. 6. Granularity performance relative to the multigrain speedup on a 4-PE Cascadia system.

nested level (*Ave. NL*), and the maximum nested level (*Max. NL*). These loop trees are larger compared to loop trees presented in [3] and [27], which were constructed statically from the source codes. Several reasons can be attributed to a larger loop tree including compiling to a different ISA or using different data sets, but the main reason has to do with the way Cascadia dynamically detects loops. Cascadia propagates nested loop status through function calls, which leads to a loop tree containing more loop nodes and deeper nested levels. This allows Cascadia to more accurately predict the correct loop level to multithread resulting in an optimal performance.

4.2 Thread-Level Granularity Comparison

Choosing the proper granularity of threads is crucial for maintaining an optimal performance, not only in D-SpMT but also parallel processing in general. As such, both *coarse-grain* threads extracted from outermost loops and *fine-grain* threads extracted from inner loops have advantages and disadvantages. Many SpMT methods either exploit only coarse-grain [1], [4], [10], [11] or fine-grain threads [3], [8], [12] and ignore the performance potential of the other. The main advantage of Cascadia over other SpMT architectures is the ability to exploit *multigrain* threads. Thus, this section compares the multigrained performance against both coarse- and fine-grain methods. Fig. 6 shows the relative performance of the multigrain approach versus fine-grain and coarse-grain methods on a 4-PE Cascadia system.

In general, coarse-grain threads result in better performance compared to fine-grain threads. Disregarding `181.mcf`, the multigrain method improves performance of SPEC2000 by only an average of 0.2 percent over both coarse-grain and fine-grain. In fact, the difference in all three methods varies by no more than 1.5 percent, meaning that all three methods are comparable when multithreading benchmarks from SPEC2000. `181.mcf` performs better with both fine- and coarse-grain methods as it contains several doubly nested loops, where both inner- and outer loops perform well. In contrast, Cascadia went from an inner loop back to an outer loop and lost performance from reentering the Pre-SpMT phase on the outer loop.

On the other hand, the multigrained method is clearly superior when benchmarks are well structured with lots of good loops. On average, the multigrained method increases performance of MediaBench by 4.2 percent over coarse-grain and 14.3 percent over fine-grain. `MPEG2`, `EPIC`, and `Pegwit` all have loop structures with multiple levels of good performing loops, resulting in a combined average performance increase of 8 percent over coarse-grain and 26.6 percent over fine-grain. These results clearly demonstrate that different loop levels, at different times, perform well, thus, illustrating the importance of choosing the correct nested loop level.

4.3 Thread Performance

Fig. 7 illustrates the percentage of cycles spent in each of the three multithreading modes. No increase in performance is possible when the system is in the *Normal* mode. The time spent in the *Pre-SpMT* mode also does not provide any performance increase since the system is gathering the necessary information to perform multithreading. *SpMT* is the only mode that provides performance improvement. On average, the SPEC2000 benchmarks spend approximately 10 percent of the time in the *SpMT* mode, while the MediaBench benchmarks fared much better at approximately 35 percent.

Fig. 8 illustrates the quality of the loops being multithreaded by showing a breakdown of all spawned threads while in the *SpMT* mode. Fig. 8 labels all successfully completed threads as *Good* and categorizes the remainder of the threads into the reasons why they were misspeculated. The first group of misspeculations is due to interthread register dependency violations, which have been further separated into three categories: 1) first-level dependency misspeculation (*ITDep1L*), 2) second-level dependency misspeculation (*ITDep2L*), and 3) failing to initially detect a dependency at all (*ITDep*). Other squashes are caused by threads that either fail the sIPC test (*sIPC*), which forces the system to spawn an inner loop, violate a memory dependency (*Load*), or misspeculate a register stride value (*Stride*).

Fig. 8 shows that misspeculations from both *Stride* and *sIPC* account for only 0.7 and 0.2 percent of the threads within SPEC2000 and MediaBench, respectively, which means that Cascadia properly predicts interthread register strides and nested loop levels. The majority of squashed threads are attributed to misspeculated interthread register and memory dependencies with an average squash rate of 48.8 and 26.4 percent for SPEC2000 and MediaBench,

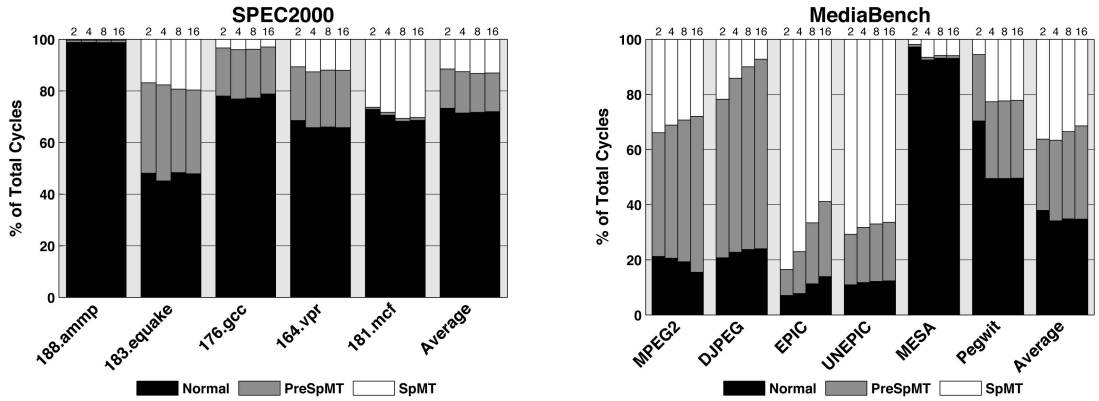

Fig. 7. A breakdown of the amount of cycles spent in each of the Cascadia modes.

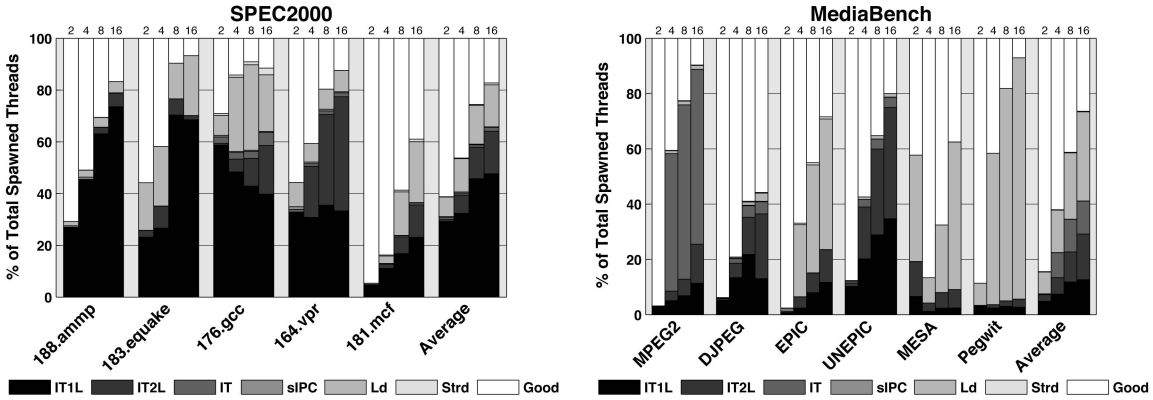

Fig. 8. A breakdown of all spawned threads, including whether the thread completed or squashed.

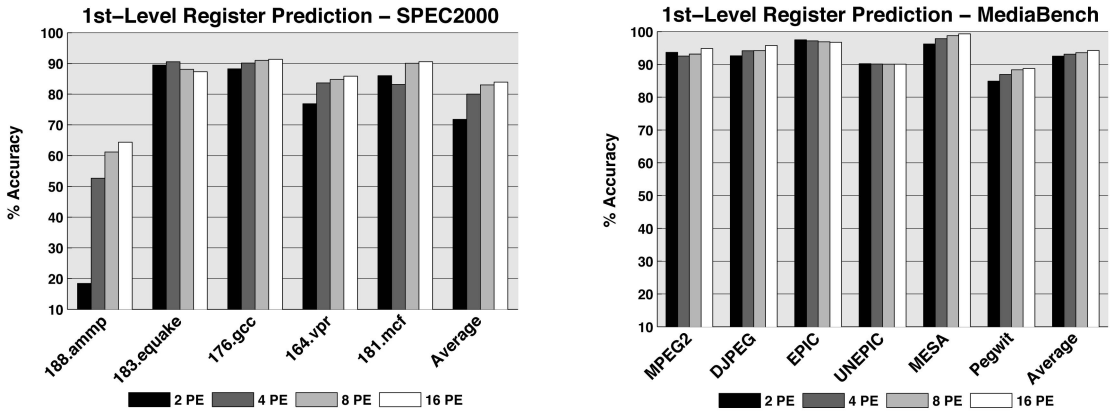

Fig. 9. First-level interthread register prediction accuracy.

respectively. First-level interthread register dependency misspeculations represent the largest share with an average miss rate of 38.7 and 9.2 percent for SPEC2000 and MediaBench, respectively. The higher rate of misspeculation for SPEC2000 is expected since the majority of their loops are very dynamic and hard to predict.

MediaBench benchmarks suffered more from interthread memory dependency violations with 19.9 percent of threads squashed from misspeculated loads, while SPEC2000 only had an average rate of 12.9 percent. Currently, Cascadia does not forward or predict memory loads between threads and relies on the Coherency Module to detect interthread memory dependency violations. As such, programs that are memory intensive, including `183.equake`, `176.gcc`, `EPIC`, `MESA`, and `Pegwit`, account for over 20 percent of misspeculations, with `Pegwit` having the highest average rate of 56.7 percent.

Figs. 9 and 10 further examine the accuracy of first- and second-level interthread register dependency predictions, respectively. First-level predictions provide an average accuracy of 72 percent with 2 PEs to 83 percent with 16 PEs for SPEC2000 and 92.4 percent with 2 PEs to 94.2 percent with 16 PEs for MediaBench. The SPEC2000 benchmarks are less accurate since most of the loops contain very hard to predict data dependencies. In contrast, second-level predictions are highly accurate with an average rate of 99.5 percent

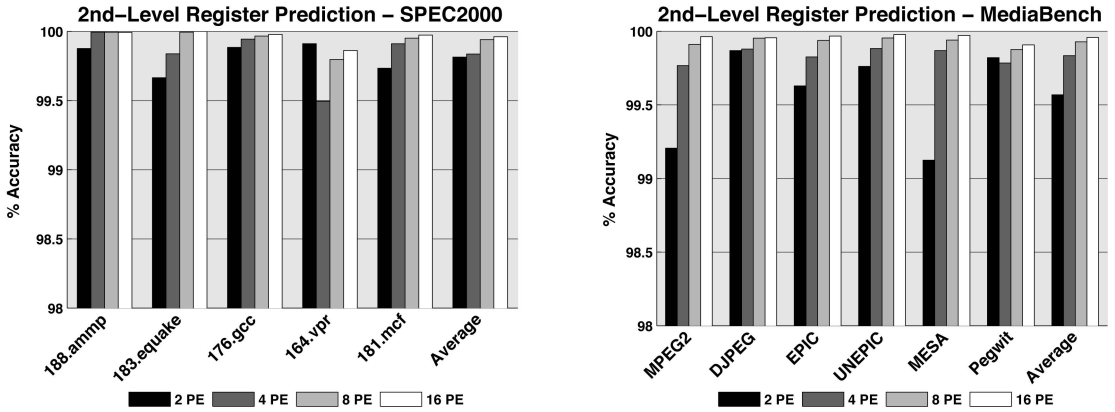

Fig. 10. Second-level interthread register prediction accuracy.

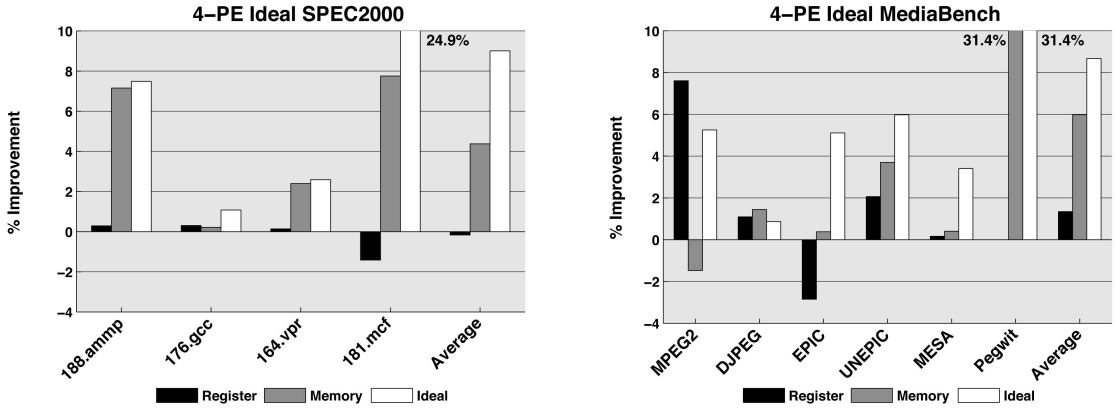

Fig. 11. Performance gains with ideal interthread dependencies on a 4-PE system.

with 2 PEs to 99.9 percent with 16 PEs. The reason that second-level prediction accuracy is so high is because the data are read from the nonspeculative head PE. In general, this PE is much farther along in the iteration than any successor PE, so the register data are more likely to be available and accurate.

4.4 Ideal Interthread Dependency Predictions

This section explores the performance potential of Cascadia through various idealized implementations that eliminate interthread dependency misspeculations. Fig. 11 shows the performance gains on a 4-PE system based on the following ideal prediction mechanisms: 1) Ideally predict all interthread register dependencies, *Register*, 2) ideally predict all interthread memory load dependencies while allowing stores to write to memory from speculative PEs, *Memory*, and 3) ideally predict both register and memory dependencies, *Ideal*. The gains in Fig. 11 are normalized to a nonideal, 4-PE configuration.

The ideal interthread register dependency prediction (*Register*) had the least impact with an average performance improvement of only 1.5 percent for MediaBench, while SPEC2000 lost about 0.2 percent performance. This shows that Cascadia's two-level interthread register dependency prediction scheme (see Section 3.2) is quite effective. The loss in performance from the ideal *Register* case in `181.mcf` and `EPIC` due to the overhead in the loop selection algorithm and the way the $sIPC$ metric is calculated. This is because when the ideal interthread register dependency prediction is applied, a speculative thread knows exactly when to read interthread dependent register data and will stall until the dependency is met. This, in turn, reduces the total number of speculatively committed instructions, decreasing the $mIPC$ value for that loop, ultimately marking the loop as *Bad sIPC*. If the loop is an inner loop, for example, the CPE will select the outer loop for multithreading on the next iteration of the outer loop. This sudden shift back to an outer loop forces Cascadia back into the *Pre-SpMT* mode for three iterations of the outer loop during which there is no performance gain.

In contrast, correctly handling interthread memory dependencies (*Memory*) has a greater potential to increase performance with an average improvement of 4.3 percent for SPEC2000 and 6 percent for MediaBench. The greatest potential performance gain occurs when threads no longer have to stall from speculative store instructions coupled with the ability to predict interthread memory loads. In this manner, memory operations for each PE can occur fairly independently from each other, regardless of whether the PE is speculative or not.

The combined set of ideal predictions (*Ideal*) increases the performance by an average of 9 percent for SPEC2000 and 8.8 percent for MediaBench. It is interesting to note that the overall *Ideal* performance for SPEC2000 and MediaBench is actually greater than the sums of the *Register* and *Memory*. The reason for this is that although the interthread

dependency prediction is ideal, the loop selection mechanism is not. Cascadia still relies on the $sIPC$ metric to determine the correct loop to multithread, and by combining both *Register* and *Memory* conditions, Cascadia finds loops that would otherwise have been ignored. These new loops achieve a much greater performance with *Ideal* than with just *Register* or *Memory* alone.

5 Related Work

The earliest study on the characteristics of dynamic loops was presented in [14]. Since then, there have been a number of studies that specifically exploit TLP through loops [3], [27], [28], [29], [30]. There also have been studies that build *control flow graphs* (CFG) or similar data structures to organize unpredictable loops into simple code segments [4], [5], [10], [16]. Other studies have looked into TLS-driven, parallelizing compilers that define several novel ways to statically select loops [31], [32], [33]. In contrast to CFG, which can only be created and analyzed statically, Cascadia constructs a loop tree dynamically with regards to function calls and uses an $sIPC$ metric to determine thread spawning points at runtime.

The majority of multithreaded architectures are static in nature and require ISA extensions and/or compiler support, including *Multiscalar* [10], *Inthreads* [8], and *Implicitly Multithreaded Processors* [23]. Recent contributions include *Multiple Instruction Stream Processor* [34], which uses a combination of OS and user-level sequencers to maximize multithreaded performance on an asymmetric multicore, and *Global Multithreaded Scheduling* [16], which simultaneously schedules different code blocks instead of the traditional linear multithreaded scheduling scheme. Other works extract TLP using an acyclic software pipelining method and include *Decoupled Software Pipelining* [35], *Source Code Annotations* [6], and *CorD* [7]. Although each of these methods demonstrates improvements through exploiting TLP, only the code segments chosen prior to running the program can be multithreaded. This eliminates the possibility of extracting TLP from other code segments that may produce good performance gains for certain data sets. In contrast, Cascadia adapts to a program by dynamically speculating threads in order to maximize the TLP performance for any given data set. In addition, Cascadia is completely backward compatible since it does not require any changes to the existing ISA or help from the OS.

The two earliest representative D-SpMT architectures are *Dynamic Multithreading* (DMT) [11] and *Speculative Multithreaded Processors* (SMPs) [36]. DMT used several innovative techniques to exploit TLP from loop continuations and procedures including the use of dataflow and data value predictions to ease the limitations of register and memory dependencies. DMT used a modified SMT architecture and was an early foundation for our work. Unfortunately, the methodology used to exploit threads limited multithreading to large, coarse-grain threads, which excluded exploitation of a potentially high degree of parallelism that exists across inner loop iterations. In contrast, SMP used a chip multiprocessor and relied on smaller threads to minimize the effect of interthread register dependencies. SMP introduced novel methods to detect loops and dependencies through special caches and lookup tables. *Dynamic Simultaneous Multithreading* (DSMT) [2] provided the capability to spawn threads at multiple levels of granularity by expanding on DMT and SMP. DSMT used an architecture similar to DMT and further refined SMP loop detection and dependency resolution to simpler tables and utility bits. In addition, DSMT introduced the concept of using IPC values to select nested loop values. Cascadia further extends the work of DMT, SMP, and DSMT by incorporating a more thorough loop structure for a wider range of comparison and an sIPC metric that handles multiple multinested loops based on loop sizes, IPC, and prior loop status. Cascadia also moves beyond a simple SMT architecture by modeling a multicore design. Although a direct comparison of Cascadia to the previous three works would be interesting, the differences are sufficient enough that making a direct comparison is impractical.

Other D-SpMT techniques include *Clustered Speculative Multithreaded Processor* (CSMP) [12], [15] and the *Polyflow Processor* [37]. CSMP divided the multithreading workload over multiple, self-contained threading units and extracted fine-grain threads from small, inner loop iterations. This requires very little hardware to handle interthread register dependencies, but also limits the amount of TLP since small fine-grain threads seldom contain enough speculative instructions for a large performance gain. In addition, the simulation study of CSMP was performed using truncated benchmarks and ignored large code segments that are difficult to multithread, which lead to idealized performance results. In contrast, Cascadia ran every benchmark, from start to finish, in order to demonstrate its performance over the whole program space.

The work that is closest to Cascadia, in the sense that it has the ability to perform multigranular threading on a multicore architecture, is *Pinot* [9]. Threads are generated from loop boundaries, function returns, and code segments and require only a minimal amount of additional hardware. Interthread register data are transferred through a *unidirectional ring network* and are similar to our proposed ITBus. Although Pinot has shown to improve the overall performance of many benchmarks, it does so at the cost of extending the ISA and requiring several modifications to the Operating System in order to support *Versioning Cache* [38]. In addition, Pinot does not exploit the dynamic characteristics of loops. Cascadia substitutes ISA extensions and compiler support with additional hardware support to dynamically speculate loop iterations for multithreading.

6 Future Work

Although results for Cascadia show a lot of promise, there are still some areas that can be improved upon. The two-level interthread register dependency predictors have shown to be fairly accurate, but mispredictions account for as much as 7.5 percent loss in performance for some benchmarks. New techniques for predicting interthread register dependencies, such as critical path detection or a branch prediction table, can reduce this loss.

Currently, Cascadia does not exploit any *Memory Level Parallelism* (MLP). Section 4.4 showed that performance can be improved by predicting interthread memory dependencies and speculatively writing to memory. Therefore, a technique needs to be developed that supports MLP similar to *Versioning Cache* [38], *Transactional Memory* [39], and/or

ARB [40]. It may also be possible to use a two-level prediction mechanism similar to the interthread register dependency predictors to predict interthread memory dependencies.

Another area that would increase the practicality of Cascadia is to eliminate altogether the need for the CPE and TCIU. The idea would be to attach *helper threads* [20] to a program to facilitate the thread creation and management. This would reduce the amount of additional hardware required, but will increase the thread spawning overhead.

7 Conclusion

This paper introduced Cascadia, a multicore architecture capable of multigranular D-SpMT. Cascadia has the unique ability to dynamically exploit multigrain threads by utilizing a loop tree that properly reflects the relationships among nested loops through procedure calls. This allows the most beneficial loops within the loop tree to be selected for multithreading based on a sustainable IPC (sIPC) metric. Our studies show that Cascadia improves the performance with average speedups of 1.02 with 2 PEs to 1.03 with 16 PEs on selected benchmarks from SPEC CPU 2000, while selected benchmarks from MediaBench show average speedups of 1.2 with 2 PEs to 2.5 with 16 PEs. The MediaBench benchmark, `EPIC`, demonstrated the best performance with a speedup of over 5 on 16 PEs. Our study also shows that the performance of multigrained multithreading is comparable to both coarse- and fine-grain multithreading for programs in SPEC CPU 2000, but for the large, well-structured programs in MediaBench, multigrain out-performed coarse-grain methods by 8 percent and over 14 percent for fine-grain methods. Finally, Cascadia's performance can be increased by as much 9 percent on a 4-PE system by improving both interthread register and memory *dependency* speculations.

References

[1] J.T. Oplinger, D.L. Heine, and M.S. Lam, "In Search of Speculative Thread-Level Parallelism," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 303-313, Oct. 1999.

[2] D. Ortiz-Arroyo and B. Lee, "Dynamic Simultaneous Multithreaded Architecture," *Proc. 16th Int'l Conf. Parallel and Distributed Computing Systems,* Aug. 2003.

[3] J. Tubella and A. González, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture,* Feb. 1998.

[4] S. Balakrishnan and G.S. Sohi, "Program Demultiplexing: Data-Flow Based Speculative Parallelization of Methods in Sequential Programs," *Proc. 33rd Ann. Int'l Symp. Computer Architecture,* pp. 302-313, June 2006.

[5] J.D. Collins, D.M. Tullsen, and H. Wang, "Control Flow Optimization via Dynamic Reconvergence Prediction," *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 129-140, Dec. 2004.

[6] M.J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D.I. August, "Revisiting the Sequential Programming Model for Multi-Core," *Proc. 40th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* pp. 69-84, Dec. 2007.

[7] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or Discard Execution Model for Speculative Parallelization on Multicores," *Proc. 41st Int'l IEEE/ACM Symp. Microarchitecture,* pp. 300-341, Nov. 2008.

[8] A. Gontmakher, A. Mendelson, A. Schuster, and G. Shklover, "Speculative Synchronization and Thread Management for Fine Granularity Threads," *Proc. 12th Int'l Symp. High-Performance Computer Architecture,* pp. 278-287, Feb. 2006.

[9] T. Ohsawa et al. "Pinot: Speculative Multi-Threading Processor Architecture Exploiting Parallelism over a Wide Range of Granularities," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 81-92, Nov. 2005.

[10] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 414-425, June 1995.

[11] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proc. 31st Ann. Int'l Symp. Microarchitecture,* pp. 226-236, Dec. 1998.

[12] P. Marcuello and A. González, "Clustered Speculative Multithreaded Processors," *Proc. 13th Int'l Conf. Supercomputing,* pp. 365-372, 1999.

[13] A. Kejariwal et al. "On the Performance Potential of Different Types of Speculative Thread-Level Parallelism," *Proc. 20th Ann. Int'l Conf. Supercomputing,* 2006.

[14] M. Kobayashi, "Dynamic Characteristics of Loops," *IEEE Trans. Computers,* vol. 33, no. 2, pp. 125-132, Feb. 1984.

[15] P. Marcuello and A. González, "Thread-Spawning Schemes for Speculative Multithreading," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture,* Feb. 2002.

[16] G. Ottoni and D.I. August, "Global Multi-Threaded Instructions Scheduling," *Proc. 40th IEEE/ACM Int'l Symp. Microarchitecture,* Dec. 2007.

[17] C.G. Quiñones et al. "Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices," *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 269-279, 2005.

[18] M. Tremblay and S. Chaudhry, "A Third-Generation 65 nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor," *Proc. IEEE Int'l Solid-State Circuits Conf.,* vol. 51, pp. 82-83, Feb. 2008.

[19] M. Tremblay, B. Joy, and K. Shin, "A Three Dimensional Register File for Superscalar Processors," *Proc. 28th Hawaii Int'l Conf. System Sciences,* 1995.

[20] J. Lu et al. "Dynamic Helper Threaded Prefetching on the Sun UltraSPARC CMP Processor," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 93-104, Nov. 2005.

[21] D.A. Zier, J.A. Nelson, and B. Lee, "NetSim: An Object-Oriented Architectural Simulator Suite," *Proc. Int'l Conf. Computer Design,* June 2005.

[22] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, no. 2, pp. 59-67, Feb. 2002.

[23] I. Park, B. Falsafi, and T.N. Vijaykumar, "Implicitly-Multithreaded Processors," *Proc. 30th Ann. Int'l Symp. Computer Architecture,* pp. 39-51, June 2003.

[24] J.L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millinnium," *Computer,* vol. 33, no. 7, pp. 28-35, July 2000.

[25] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *Proc. IEEE Int'l Symp. Microarchitecture,* pp. 330-335, 1997.

[26] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE Computer Architecture Letters,* vol. 1, p. 7, June 2002.

[27] M.R. de Alba and D.R. Kaeli, "Characterization and Evaluation of Hardware Loop Unrolling," *Proc. First Boston Area Architecture Conf.,* Jan. 2003.

[28] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 58-69, Oct. 1998.

[29] D. Puppin and D. Tullsen, "Maximizing TLP with Loop-Parallelization on SMT," *Proc. Workshop Multithreaded Execution, Architecture, and Compilation,* 2001.

[30] J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. 1996 Conf. Parallel Architectures and Compilation Techniques,* Oct. 1996.

[31] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs," *Proc. ACM SIGPLAN 2004 Conf. Programming Language Design and Implementation,* pp. 71-81, 2004.

[32] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: A TLS Compiler That Exploits Program Structure," *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 158-167, 2006.
[33] S. Wang, X. Dai, K.S. Yellajyosula, A. Zhai, and P.-C. Yew, "Loop Selection for Thread Level Speculation," *Proc. 18th Ann. Workshop Languages and Compilers for Parallel Computing,* pp. 289-303, 2005.
[34] R.A. Hankins, G.N. Chinya, J.D. Collins, P.H. Wang, R. Rakvic, H. Wang, and J.P. Shen, "Multiple Instruction Stream Processor," *Proc. 33rd Ann. Int'l Symp. Computer Architecture,* pp. 114-127, June 2006.
[35] G. Ottoni et al. "From Sequential Programs to Concurrent Threads," *IEEE Computer Architecture Letters,* vol. 5, no. 1, Jan. 2006.
[36] P. Marcuello, A. González, and J. Tubella, "Speculative Multithreaded Processors," *Proc. 12th Int'l Conf. Supercomputing,* pp. 77-84, 1998.
[37] T.M. Rafacz, "Spawn Point Prediction for a Polyflow Processor," master's thesis, Univ. of Illinois at Urbana Champaign, 2005.
[38] T.N. Vijaykumar, S. Gopal, J.E. Smith, and G. Sohi, "Speculative Versioning Cache," *IEEE Trans. Parallel Distributive Systems,* vol. 12, no. 12, pp. 1305-1317, Dec. 2001.
[39] J. Chung et al. "The Common Case Transactional Behavior of Multithreaded Programs," *Proc. 12th Ann. Int'l Symp. High-Performance Computer Architecture,* pp. 266-277, 2006.
[40] M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Computers,* vol. 45, no. 5, pp. 552-571, May 1996.

David A. Zier received the BS, MS, and PhD degrees in electrical and computer engineering from the School of Electrical Engineering and Computer Science at Oregon State University, Corvallis, in 2002, 2004, and 2009, respectively. He is currently a hardware architect for the NVIDIA Corporation. His research interests include multithreading and thread-level speculation, computer architecture, embedded systems, computer graphics, and cryptography. He is a member of the IEEE and the IEEE Computer Society.

Ben Lee received the BE degree in electrical engineering in 1984 from the Department of Electrical Engineering at the State University of New York (SUNY) at Stony Brook and the PhD degree in computer engineering in 1991 from the Department of Electrical and Computer Engineering at the Pennsylvania State University. He is currently a faculty member in the School of Electrical Engineering and Computer Science at Oregon State University. He has published more than 80 conference proceedings, book chapters, and journal articles in the areas of embedded systems, computer architecture, multithreading and thread-level speculation, parallel and distributed systems, and wireless networks. He received the Loyd Carter Award for Outstanding and Inspirational Teaching and the Alumni Professor Award for Outstanding Contribution to the College and the University from the OSU College of Engineering in 1994 and 2005, respectively. He also received the HKN Innovative Teaching Award from Eta Kappa Nu, School of Electrical Engineering and Computer Science in 2008. He has been on the program and organizing committees for numerous international conferences, including the 2000 International Conference on Parallel Architecture and Compilation Technique (PACT), the 2001 and the 2004 IEEE Pacific Rim Dependable Computing Conference (PRDC), the 2003 International Conference on Parallel and Distributed Computing Systems (PDCS), the 2005-2008 IEEE Workshop on Pervasive Wireless Networking (PWN), and the 2009 IEEE International Conference on Pervasive Computing and Communications. He is currently the workshop chair for PerCom 2009. He was also an invited speaker at the 2007 International Conference on Embedded Software and System. His research interests include multithreading and thread-level speculation, computer architecture, embedded systems, and wireless networks. He is a member of the IEEE Computer Society.