Combining Thread Level Speculation, Helper Threads, and Runahead Execution*

Polychronis Xekalakis†, Nikolas Ioannou, and Marcelo Cintra

School of Informatics
University of Edinburgh

p.xekalakis@ed.ac.uk, nikolas.ioannou@ed.ac.uk, mc@staffmail.ed.ac.uk

ABSTRACT

With the current trend toward multicore architectures, improved execution performance can no longer be obtained via traditional single-thread instruction level parallelism (ILP), but, instead, via multithreaded execution. Generating thread-parallel programs is hard and thread-level speculation (TLS) has been suggested as an execution model that can speculatively exploit thread-level parallelism (TLP) even when thread independence cannot be guaranteed by the programmer/compiler. Alternatively, the helper threads (HT) execution model has been proposed where subordinate threads are executed in parallel with a main thread in order to improve the execution efficiency (i.e., ILP) of the latter. Yet another execution model, runahead execution (RA), has also been proposed where subordinate versions of the main thread are dynamically created especially to cope with long-latency operations, again with the aim of improving the execution efficiency of the main thread.

Each one of these multithreaded execution models works best for different applications and application phases. In this paper we combine these three models into a single execution model and single hardware infrastructure such that the system can dynamically adapt to find the most appropriate multithreaded execution model. More specifically, TLS is favored whenever successful parallel execution of instructions in multiple threads (i.e., TLP) is possible and the system can seamlessly transition at run-time to the other models otherwise. In order to understand the tradeoffs involved, we also develop a performance model that allows one to quantitatively attribute overall performance gains to either TLP or ILP in such combined multithreaded execution model.

Experimental results show that our unified execution model achieves speedups of up to 41.2%, with an average of 10.2%, over an existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over a flavor of runahead execution for a subset of the SPEC2000 Int benchmark suite.

*This work was supported in part by EPSRC under grant EP/G000697/1 and the EC under grant HiPEAC IST-004408.

†The author was supported in part by a Wolfson Microelectronics scholarship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.
Copyright 2009 ACM 978-1-60558-498-0/09/06 ...$5.00.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Parallel Architectures

General Terms

Performance

Keywords

Multi-cores, Thread-Level Speculation, Helper Threads, Runahead Execution

1. INTRODUCTION

With the scaling of devices continuing to progress according to Moore's law and with the non-scalability of out-of-order processors, multicore systems have become the norm. The design effort is thus alleviated from the hardware and placed instead on the compiler/programmer camp. Unfortunately parallel programming is hard and error-prone, sequential programming is still prevalent, and compilers still fail to automatically parallelize all but the most regular programs.

One possible solution to this problem is provided by systems that support Thread-Level Speculation (TLS) [7, 10, 12, 16, 17]. In these systems, the compiler/programmer is free to generate threads without having to consider all possible cross-thread data dependences. Parallel execution of threads then proceeds speculatively and the TLS system guarantees the original sequential semantics of the program by transparently detecting any data dependence violations, squashing the offending threads, and returning the system to a previously non-speculative correct state. Thus, the TLS model improves overall performance by exploiting Thread-Level Parallelism (TLP) via the concurrent execution of instructions in multiple threads [1] (Figure 1(a)).

Another possible solution to accelerate program execution in multicore systems without resorting to parallel programs is provided by systems that support Helper Threads (HT) [3, 4, 18, 20]. In these systems, the compiler/programmer extracts small threads, often called slices, from the main thread such that their execution in parallel with the main thread will lead to improved execution efficiency of the latter. Most commonly, helper threads are used to resolve highly unpredictable branches and cache misses before these are required by the main thread. In almost all cases, the benefits of HT are indirect in the sense that no actual program computation is executed in parallel with the main thread. Thus, the HT model improves overall performance by exploiting Instruction-Level Par-

[1] In reality, the basic TLS execution model also provides some indirect non-TLP performance benefits as explained later in the paper.

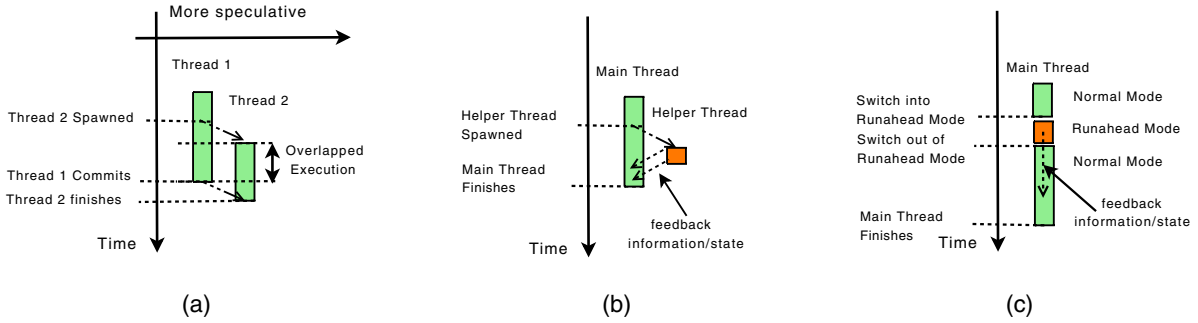

Figure 1: Different models of multithreaded execution: (a) Thread Level Speculation. (b) Helper Thread. (c) Runahead Execution.

allelism (ILP) via the improved execution efficiency within a single main thread (Figure 1(b)).

A solution related to HT is to support Runahead execution (RA) [1, 2, 6, 9, 13]. In these systems, the hardware transparently continues execution of instructions from the main thread past a long latency operation, such as a cache miss. Unlike with HT, the instructions in the runahead thread are not explicitly extracted and placed in a subordinate thread and are not executed concurrently with the main thread. In fact, RA can be even implemented in single-core systems with some enhanced context checkpointing support. Like HT, the benefits from RA are indirect and it improves overall performance by exploiting ILP (Figure 1(c)).

Each of these three multithreaded execution models has been separately shown to improve overall performance of some sequential applications. However, given the different nature of the performance benefits provided by each model, one would expect that combining them in a unified execution model would lead to greater performance gains and over a wider range of applications compared to each model alone. Moreover, much of the architectural support required by each model is similar, namely: some support for checkpointing and/or multiple contexts, some support to maintain speculative (unsafe) data, and some support for squashing threads. Despite these opportunities no one (to the best of our knowledge) has attempted to combine these multithreaded execution models.

In this paper we propose to combine all three multithreaded execution models in a super-set unified model that can exploit the benefits of each model depending on application characteristics. More specifically, the resulting system attempts to exploit TLP speculatively with TLS execution, but when this fails or when additional opportunities exist for exploiting ILP the system also employs a version of HT that is based on RA. We chose this model of HT so that its threads interact seamlessly with TLS threads and only small modifications to the TLS protocol and the TLS architectural support are required. In the paper we discuss in detail this interaction and how to tune the HT and TLS models to work synergystically. Another contribution of this paper is a simple methodology that allows one to model the performance gains with TLS and the unified execution model such that gains can be accurately attributed to either TLP or ILP. This methodology, then, allows one to reason about the behavior of the execution models and to investigate tradeoffs in the unified model.

Experimental results show that our unified execution model achieves speedups of up to 41.2%, with an average of 10.2%, over an existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over a flavor of runahead execution for a subset of the SPEC2000 Int benchmark suite.

The rest of this paper is organized as follows. Section 2 provides a brief description of the TLS, HT and RA execution models. Section 3 presents our proposed scheme to combine the three execution models. Section 4 presents our methodology for quantifying the contributions to performance gains that come from TLP and ILP. Section 5 describes the experimental methodology and Section 6 presents results. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2. BACKGROUND ON TLS, HT AND RA EXECUTION MODELS

2.1 TLS

Under the *thread-level speculation* (also called *speculative parallelization*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads from speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be re-executed. In most schemes a squash rolls the execution back to the start of the thread, but some proposals in the literature use periodic *checkpointing* of threads such that upon a squash it is only necessary to roll the execution back to the closest safe checkpointed state. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage (usually main memory or some shared higher-level cache). At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. Figure 1(a) depicts this execution model.

Speculative threads are usually extracted from either loop iterations or function continuations, without taking into consideration possible data dependence violations. The compiler marks these structures with a fork-like *spawn instruction*, so that the execution of such an instruction leads to a new speculative thread. The *par-*

ent thread continues execution as normal, while the *child* thread is mapped to any available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread. Threads formed from iterations of the same loop (and that, thus, have the same spawn point) are called *sibling* threads. For function calls, spawn points are placed just before the function call such that the non-speculative thread proceeds to the body of the function, and a speculative thread is created from the function's continuation.

The architectural support required by TLS consists of six main components: i) a mechanism to allow speculative threads to operate in their own context and to enforce that speculatively modified data be also kept separate, ii) a mechanism to track data accesses in order to detect any data dependence violations, iii) a mechanism to spawn threads in different cores or contexts, iv) a mechanism to rollback (i.e., squash and restart) incorrectly executed threads, v) a mechanism to commit the correctly speculatively modified data to the safe state, and vi) a mechanism to keep track of the ordering of threads with respect to the original sequential execution order.

2.2 HT

Under the *helper threads* (also called *subordinate microthreads*) approach, small threads are run concurrently with a main thread. The purpose of the helper threads is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Common ways to accelerate the execution of the main thread involve initiating memory requests ahead of time (i.e., prefetching; such that the results are hopefully in the cache by the time they are needed by the main thread) and resolving branches ahead of time. Usually, depending on how the helper threads are generated (see below), the execution of helper threads is speculative in that they may be following some incorrect control flow path and/or producing and consuming incorrect data. In this multithreaded execution model there is no particular ordering among multiple helper threads and all are *discarded* at the end of their execution. Figure 1(b) depicts this execution model.

Helper threads are usually generated by the compiler or programmer and often consist of *slices* of instructions from the main thread (e.g., only those instructions directly involved in the computation of some memory address or branch condition). Depending on the size and complexity of the helper threads it may be possible to keep all their intermediate results in the registers, but it may be necessary to allow for spills to the memory hierarchy, which in turn requires providing storage for speculative *versions* of data. The compiler marks the main thread with fork-like *spawn instructions* at points where particular helper threads should be initiated.

The architectural support required by HT consists of three main components: i) a mechanism to allow helper threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to spawn threads in different cores or contexts, and iii) a mechanism to discard threads when finished.

2.3 RA

Under the *runahead* approach, when the main thread hits a long-latency operation (e.g., an L2 miss) it halts execution and a runahead thread continues execution either ignoring or predicting the outcome of the long-latency operation. The purpose of the runahead thread is not to directly contribute to the actual program computation, which is often still performed in full by the main thread once it resumes, but to facilitate the execution of the main thread indirectly after it resumes. As with HT, common ways to accelerate the execution of the main thread involve prefetching and early branch resolution. Unlike HT, runahead threads do not run concurrently with the main thread. The execution of the runahead thread is speculative since the outcome of the long-latency operation is either ignored or predicted. Thus, in most proposed models the runahead thread is *discarded* once the main thread resumes execution. In more aggressive models, however, if the predicted outcome of the long-latency operation is correct the execution of the runahead thread is incorporated into the main thread before stopping the execution of the runahead thread. Figure 1(c) depicts this execution model.

Runahead threads are generated on-the-fly by the hardware and, like the common HT case, consist of a selection of instructions from the main thread. Strictly speaking, in many proposals in the literature, the runahead threads are in fact obtained by simply *checkpointing* the main thread and letting it run ahead instead of explicitly spawning a new thread elsewhere. Also like HT, it may be possible to keep all the intermediate results of the runahead thread in the registers, but it may be necessary to allow for spills to the memory hierarchy.

The architectural support required by RA consists of five main components: i) a mechanism to allow runahead threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to decide when to generate runahead threads or to switch the main thread into runahead mode, iii) a mechanism to discard incorrectly executed threads, and iv) and v) optional mechanisms to check if the runahead thread has executed based on correct or incorrect predicted outcomes and, if so, to incorporate the runahead state and data into the main thread.

Table 1 summarizes the architectural support required by the three multithreaded execution models in columns 2 to 4 (the last column shows the support used by our unified scheme, which is described in Section 3).

Mechanism	TLS	HT	RA	Unified
Data Versioning	✓	✓	✓	✓
Data Dep. Tracking	✓	X	X	✓
Spawn Threads	✓	✓	X	✓
Discard/Rollback	✓	✓	✓	✓
Commit State	✓	X	O	✓
Order Threads	✓	X	X	✓
Checkpoint Threads	O	X	✓	✓
Value Predict L2 Misses	X	X	O	✓

Table 1: Hardware support required for the different models of multithreaded execution and for our unified approach. *O* stands for optional feature, *X* stands for feature not required, and ✓ stands for feature required.

3. COMBINING TLS, HT, AND RA EXECUTION

3.1 Basic Idea

Each of the multithreaded execution models that we consider – TLS, HT, and RA – is best at exploiting different opportunities for accelerating the performance of a single-threaded application. We expect a unified scheme both to perform as well as the best model across a variety of applications and to even outperform the best model. The latter can happen if the unified scheme can adapt to the different acceleration opportunities of different program phases

or if the acceleration opportunities are additive (e.g., if ILP can be exploited in addition to TLP for some program phase).

The basic idea of our proposal for combining TLS, HT, and RA execution is to start with a TLS execution and to convert some TLS threads into helper threads by switching them to runahead execution mode. Threads that have been converted to helper threads execute the same instructions as they would in TLS mode, but runahead execution is achieved by allowing them to predict the results of L2 misses instead of stalling, as done in the RA execution model (as opposed to being achieved by some compiler/programmer slicing mechanism). These converted helper threads can no longer contribute to the actual parallel computation (i.e., they can never commit) but can only help the remaining TLS threads execute more efficiently. In a multicore environment with TLS this can be achieved when the converted helper threads bring data into L2 that will be later used by the remaining TLS threads. Note that in TLS the L1 cache is versioned so that no sharing and, thus, no prefetching, can occur across helper threads and TLS threads.

Combining TLS, HT, and RA execution is a reasonable approach for three main reasons. Firstly, because we start by employing only TLS, we first try to extract all the available TLP. This makes sense in a system with several cores since, if TLP can be exploited, it is more likely that this will yield better performance than trying to speed up a single main thread. When we fail to speculatively extract TLP, we may utilize the extra hardware resources to improve the ILP of the main thread, whereas a base TLS system would be idle. Secondly, accommodating HT and RA execution within the TLS execution model requires only slight modifications to the base TLS system (Table 1). Finally, starting from TLS threads and implicitly generating runahead slices according to the RA model is a simple and automatic way of generating helper threads (no programmer intervention is required).

While the basic idea is simple, developing a fully working system requires dealing with a few implementation issues. The key issue relates to the policy of *when, where and how to create HT*. These decisions are critical because in our HT/RA model threads do not contribute to TLP and consume TLP resources (e.g., cores, caches), so that a conversion policy must balance the potential increase in ILP with the potential loss of TLP. Another aspect of this is whether helper threads are simply converted from existing TLS threads in place, or whether new TLS threads are specifically created elsewhere for the purpose of becoming helper threads. This decision also affects how to manage helper threads in the context of an extended TLS environment. In particular, the TLS environment imposes a total ordering on the threads in the system, which is reasonable for TLS threads, but becomes slightly more involved when some threads are TLS and some are HT. Also, a question is what to do with a helper thread when it detects a data dependence violation and when one of its predecessors is squashed. These issues are discussed in Section 3.2. Finally, another important issue relates to the policy of converting threads back from HT to TLS threads. Since our simplified HT/RA model does not allow for their execution to be integrated into the TLS execution, this latter policy boils down to *how to destroy HT* and free up resources for further TLS threads. This issue is discussed in Section 3.3.

3.2 When, Where and How to Create HT

We identify two suitable occasions for creating a helper thread: at thread spawn and when a TLS thread suffers an L2 cache miss. By creating a helper thread on every thread spawn, we make the assumption that the original thread will benefit from prefetching, which may not be always true. On the other hand, creating a helper thread on an L2 miss will only help if the original thread will suffer more L2 misses later. Luckily TLS threads exhibit locality in their misses, that is, they either suffer many misses in the L2 cache or they do not suffer any (due to changes in the working set). Thus, assuming that an L2 miss will be followed by other misses is a reasonable assumption. We experimented with both approaches and found out that indeed this was the case and that the approach of spawning helper threads on a L2 miss performs better (Section 6.2.1).

As for the location where to execute the helper thread there are two possibilities: in the same core where the original TLS thread was executing (thus, effectively converting the TLS thread into a helper thread) or in a different idle core (thus, effectively *cloning* the original TLS thread and converting the clone into a helper thread, see Figure 2(a)). Obviously, the first option will sacrifice the exploitation of TLP, which may not be easily recovered by the benefits of the helper thread. On the other hand, the second option leads to an increased number of threads in the system, which increases the pressure on resources, possibly leading to performance degradation. If we decide to convert an existing thread, we simply have to *checkpoint* and mark the thread as a helper thread. This thread will proceed until the end of its execution disregarding long latency events and restart signals. If we instead create a new thread, we will have to do so using the existing TLS spawning model and marking the thread as a helper thread. We experimented with both approaches and found out that the latter performs better when it is coupled with throttling heuristics that restrict the system's load (Section 6.2.2).

Throughout this work we optimistically assume that TLS threads will perform useful work. Since helper threads require resources that could otherwise be used by TLS threads, indiscriminately creating helper threads at every L2 cache miss may prove detrimental to the final system's performance. Thus, it is important to make the helper threads as transparent as possible. A simple way of doing this is by allowing only a small number of helper threads to exist at any given time and to ensure that TLS threads will always be given priority over these helper threads. Although spawning on an L2 miss will create helper threads only for the threads that will likely benefit from prefetching, this may still result in the creation of too many threads. For this reason we only create a helper thread for the L2 misses if there is a free processor. Additionally, we do not allow any of the helper threads to perform any thread spawn themselves. In all cases, if we spawn a normal TLS thread and we do not have any free processor available, we pre-empt one of the running helper threads by killing it. We experimented with these different approaches and found out that keeping the number of helper threads small with the policies above gives better results (Section 6.2.3).

We also found that by allowing only the most speculative thread to spawn a helper thread on an idle core we can achieve most of the benefits one can achieve by allowing multiple helper threads to coexist (Section 6.2.4). This is to be attributed to a *chain-prefetching* effect, under which a helper thread prefetches only for the most speculative thread, which in turn goes faster and prefetches for its parent thread. An example of this can be seen in Figure 3, where under normal TLS (Figure 3(a)) both *Thread 1* and *Thread 2* suffer L2 misses at about the same time. When we clone *Thread 2*, we manage to get rid only of the second miss from *Thread 2* (Figure 3(b)). However, this makes *Thread 2* reach the third miss faster and, thus, prefetch it for *Thread 1*.

In addition to throttling the use of resources, another reason for only allowing the most speculative thread to spawn a helper thread is that it greatly simplifies the unified TLS+HT protocol. By doing so, we separate the TLS and the helper threads in the total thread ordering scheme. This in turn means that we do not have to deal with complex interactions, such as the case that a helper thread

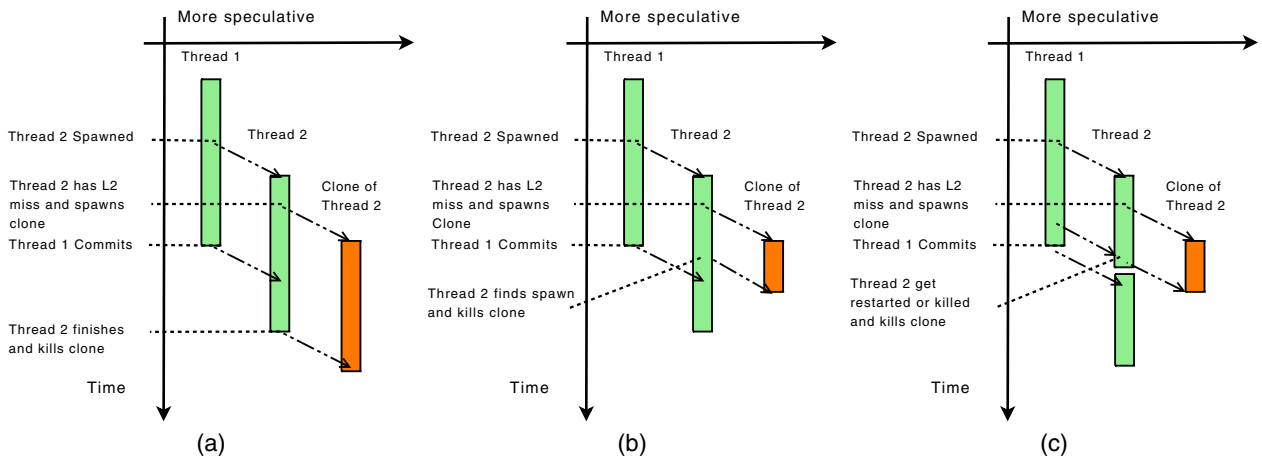

Figure 2: Helper threading with clones: (a) A thread is cloned on an L2 miss. (b) The clone is killed on a thread spawn. (c) The clone is killed on a restart/kill of the thread that spawned it.

triggers a data dependence violation with a more speculative TLS thread or that a more speculative TLS thread attempts to consume a version produced by the helper thread.

3.3 When to Terminate a Helper Thread

A helper thread should be terminated in any of the following five cases. The first case is when the helper thread reaches the commit instruction inserted by the compiler that denotes the end of the thread. The second case is when the parent thread that created the helper thread reaches the commit instruction (Figure 2(a)). The third case is when the parent thread finds a TLS spawn instruction (Figure 2(b)). Fourth, if the thread that created the helper thread receives a restart or kill signal, the helper thread has to be killed as well to facilitate keeping the ordering of threads in the system (Figure 2(c)). Finally, helper threads use predicted values for the level two cache misses and as such they might follow incorrect execution paths. So the fifth case occurs when one of these paths causes an exception.

4. A METHODOLOGY TO QUANTIFY ILP AND TLP PERFORMANCE GAINS IN SPECULATIVE MULTITHREADED EXECUTIONS

With multithreaded execution models part of the performance variation observed is due to overlapped execution of instructions from multiple threads where these instructions contribute to the overall computation – we call this a TLP contribution. Another part of the performance variation is due to indirect contributions that improve (or degrade) the efficiency of execution of the threads – we call this an ILP contribution. For instance, with TLS the parallel execution of threads leads to a TLP contribution but also prefetching effects may lead to an ILP contribution. This may happen when some threads share data so that the first thread to incur a cache miss effectively prefetches for the others such that the others will appear to have an improved ILP. Note that it is also possible that due to contention for resources, threads appear to have a degraded ILP. Note also that with speculative multithreaded models the possibility of squashes and re-execution of threads leads to even more intricate relationships between TLP and ILP contributions. Accurately quantifying the TLP and ILP contributions toward the final observed performance variation is critical in order to reason and act upon the behavior of multithreaded execution models. In this section we present a methodology to quantify these TLP and ILP contributions in multithreaded execution models using easy to collect timing information from the actual execution. The model is a variation of that proposed in [15]: it requires one extra simulation run but provides a more accurate estimate of the ILP contribution (Section 6.3).

The performance model is based on measuring the following quantities from the execution: 1) the execution time of the original sequential code (T_{seq}); 2) the execution time of the modified TLS code when executed in a single core (T_{1p}); 3) the sum of execution times among all threads that actually *commit* ($\sum T_i$, for all threads i that commit); and 4) the execution time of the modified TLS code when executed in multiple cores (T_{mt}). Figure 4 depicts these quantities for a simple example with two threads. With these quantities, the overall performance variation (S_{all}) is given by Equation 1 and the performance variation (usually a slowdown) due to the TLS instrumentation overhead (S_{1p}) is given by Equation 2. The latter is needed in order to account for the variations needed in the binaries that execute the sequential and the multithreaded versions of the program.

$$S_{all} = \frac{T_{seq}}{T_{mt}} \tag{1}$$

$$S_{1p} = \frac{T_{seq}}{T_{1p}} \tag{2}$$

The overall performance variation of the multithreaded execution *over the TLS code executed in a single core* (S_{comb}) is given by Equation 3. This performance variation reflects the combined effects of both ILP and TLP and the equality shown in Equation 4 holds.

$$S_{comb} = \frac{T_{1p}}{T_{mt}} \tag{3}$$

$$S_{comb} = S_{ilp} \times S_{tlp} \tag{4}$$

The performance variation of the multithreaded execution *over the TLS code executed in a single core* and that can be attributed to ILP effects (S_{ilp}) is given by Equation 5.

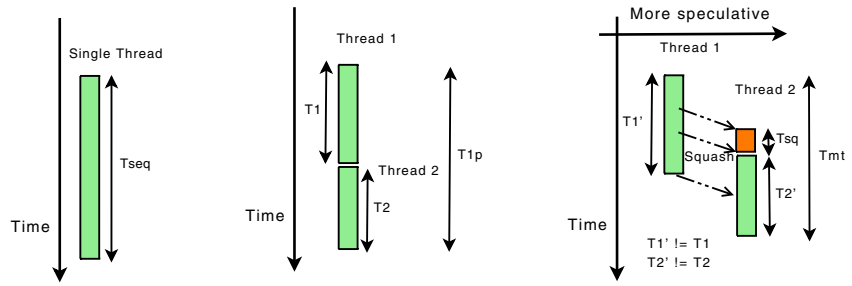

Figure 4: Quantifying ILP and TLP benefits.

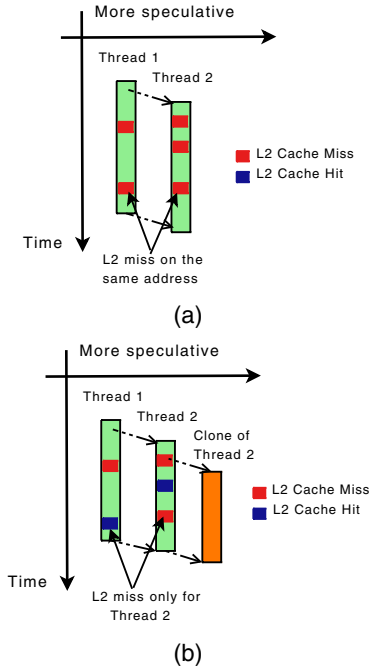

Figure 3: Chain Prefetching effect: (a) Under normal TLS execution both threads find L2 cache misses concurrently. (b) The clone prefetches for Thread 2, which in turn prefetches for Thread 1.

$$S_{ilp} = \frac{T_{1p}}{\sum T_i} \tag{5}$$

Thus, S_{ilp} can be computed with the measurements of T_{1p} and all T_i's. The performance variation of the multithreaded execution *over the TLS code executed in a single core* and that can be attributed to TLP effects (S_{tlp}) can be computed by substituting the results of Equations 3 and 5 into Equation 4. Finally, we observe that the equality shown in Equation 6 holds, which shows that the final observed performance variation can be quantitatively attributed to the variations in the binary, to the ILP contributions, and to the TLP contributions.

$$S_{all} = S_{1p} \times S_{ilp} \times S_{tlp} \tag{6}$$

Comparing our model with that proposed in [15] it can be shown that the key difference is that the ILP estimate in that model is ultimately derived from the "code bloat" factor f_{bloat}, while it is derived in our model from the actual instrumentation slowdown S_{1p}. The problem with using f_{bloat} as a proxy for the actual slowdown is that it implicitly assumes that the CPI of the unmodified sequential execution and that of the TLS-instrumented sequential execution are the same. In reality, however, the TLS instrumentation does affect the CPI as it involves the addition of mainly function calls and memory operations to set up thread spawns, which in turn have a different CPI from the rest of the thread. Obviously, this impact on CPI will be more pronounced for smaller threads than for larger ones. In our experiments (Section 6.3) we measured the difference in CPI and the ultimate impact on the ILP contribution estimation and found out that it can be significant in several cases.

5. EVALUATION METHODOLOGY

5.1 Simulation Environment

We conduct our experiments using the SESC simulator [14]. The main microarchitectural features are listed in Table 2. The system we simulate is a multicore with 4 processors, where each processor is 4-issue out-of-order superscalar. The branch predictor is a hybrid bimodal-gshare predictor. The minimum branch misprediction latency is 12 cycles while we also employ speculative updates of the global history register along the lines of [8]. Each processor has a multi-versioned L1 data cache and a non-versioned L1 instruction cache. All processors share a non-versioned unified L2 cache. For the TLS protocol we assume out-of-order spawning [15]. Our scheme requires on top of the aforementioned TLS support a value predictor so as to predict the value to be returned by missing loads. Throughout most of the evaluation we use a simple last-value predictor, but we show later that a better value predictor could improve the overall performance significantly. Our scheme also requires to transfer register state on a spawning of a clone thread. This is implemented using microcode and it adds an additional cost to the creation of a clone thread of 100 cycles. Note that normal TLS threads only require 20 cycles, since register state transfering is done through memory (the compiler inserts spills to memory).

5.2 Benchmarks

We use the integer programs from the SPEC CPU 2000 benchmark suite running the Reference data set. We focus on these applications because they represent a set of well accepted benchmarks that make difficult both the extraction of ILP (for RA/HT) and TLP (for TLS). We use the entire suite except *eon*, which cannot be compiled because our infrastructure does not support C++, and *gcc* and *perlbmk*, which failed to compile in our infrastructure. For

Parameter	TLS (4 cores)
Frequency	5GHz
Fetch/Issue/Retire Width	4/4/4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80/104
Ld/St Queue	54/46
Branch Predictor	96Kbit Hybrid Bimodal-Gshare
BTB/RAS	2K entries, 2-way/32 entries
Minimum Misprediction	12 cycles
Extra Hardware	
Value Predictor	4K entries, Last-Value

Table 2: Architectural parameters.

reference, the sequential (non-TLS) binaries where obtained with unmodified code compiled with the MIPSPro SGI compiler at the O3 optimization level. The TLS binaries were obtained with the POSH infrastructure [11]. In order to directly compare them, we execute a given number of simulation marks, which pinpoint specific code segments. This is necessary because the binaries are different, due to re-arrangements of the code by POSH. We simulate enough simulation marks so that the corresponding sequential application graduates more than 750 million instructions.

6. EXPERIMENTAL RESULTS

We start by showing the bottom-line results of our scheme when compared with both TLS and a flavor of runahead execution that uses value prediction but always reverts to the checkpoint made before going into runahead mode (i.e., it discards all computation done in the runahead thread and, thus, does not exploit any TLP). We next try to quantitatively explain how our scheme works and provide a detailed analysis of the reasons that led us to the proposed design.

6.1 Comparing TLS, Runahead and the Unified Scheme

Figure 5 shows how our proposed scheme performs when compared with both TLS and runahead execution with value prediction. Each of the bars shows the total speedup and the proportion of the speedup that can be attributed to ILP and TLP based on the methodology discussed in Section 4. Speedups are relative to sequential execution with the *original* sequential binary. With the light grey shade below the 1.0 point we denote the base case each of the schemes starts from when running on a single core (TLS and unified scheme have worse quality of code; this is the S_{1p} factor of Section 4) and with the next two shades the proportion of speedup due to ILP and TLP accordingly [2]. The leftmost bars correspond to the base TLS, the middle bars to runahead execution with value prediction, and the rightmost bars to our unified scheme.

Considering the base execution models alone, we first note that while TLS performs better than runahead execution for most applications, for some applications the performance of both schemes is comparable (*gzip* and *parser*). We also note that, despite the main target of TLS being the exploitation of TLP, for most applications

[2]Note that the breakdown shows proportions of speedup due to each category and the height of each portion cannot be directly read as the speedup coming from ILP and TLP.

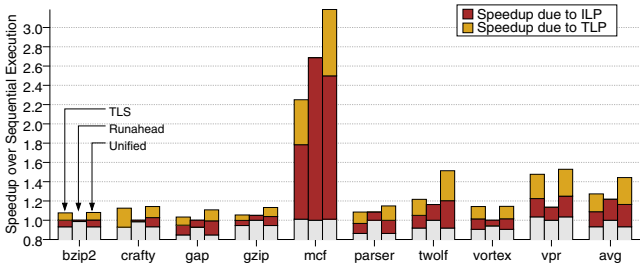

Figure 5: Speedup breakdown based on our model. Leftmost bar is for TLS, middle bar is for Runahead with value prediction, and rightmost bar is for our unified approach.

the benefits of ILP in the base TLS scheme are comparable to the TLP benefits.

Comparing our unified scheme with TLS and runahead, we see that the unified scheme performs at least as well as the best of other schemes for all applications and often outperforms the best scheme. The performance advantage of the unified scheme indicates that often the speedups of runahead execution and TLS are additive. In fact, even in applications where runahead execution fails to deliver any speedups (*crafty, gap* and *vortex*), our unified scheme achieves speedups equal to or better than TLS. When compared with TLS, we see that our unified scheme obtains greater performance gains from ILP while maintaining most of the TLP benefits. Interestingly, in some cases our unified scheme is better than the base TLS scheme even in terms of TLP. One possible reason for this is that faster speculative threads will uncover violating reads faster and thus perform restarts earlier. This is an effect similar to having lazy or eager invalidations, where it is known that eager invalidations procure better results due to increased TLP. When compared with runahead execution, we see that our unified scheme obtains gains from TLP while maintaining most of the ILP benefits. Again, in some cases our unified scheme leads to higher ILP benefits than the base runahead execution. The likely reason for this is the deeper prefetching effect that can be achieved by performing runahead execution from a speculative thread, which leads to the chain-prefetching effect described in Section 3.2. Overall, we see that our unified execution model achieves speedups of up to 41.2%, with an average of 10.2%, over the existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over runahead execution with value prediction.

Figure 6 depicts the number of L2 misses on the *commiting path* for all the schemes normalized to the sequential case. This figure allows us to quantify the ammount of prefetching happening in each scheme. Note that all three schemes have smaller miss rates than sequential execution on average since all of them perform some sort of prefetching. Runahead execution leads to only a relatively small reduction in L2 misses and, in fact, for some applications like *parser* and *twolf* it actually increases the number of L2 misses substantially. This happens because it suffers L2 misses on data that will not be useful for the main thread. For TLS prefetcing is more substantial, but still for some applications like *mcf*, *parser* and *twolf*) TLS suffers from more misses than the sequential execution. This happens due to code bloating and the overall lower quality of the code produced (some compiler optimizations are restricted by the TLS pass). The unified scheme on the other hand, is able to prefetch singnificantly more useful cache lines reducing the miss rate by 41% on average when compared with the miss rate

of sequential execution. We expect that the importance of prefetching will increase as the gap between the processor and the memory widens.

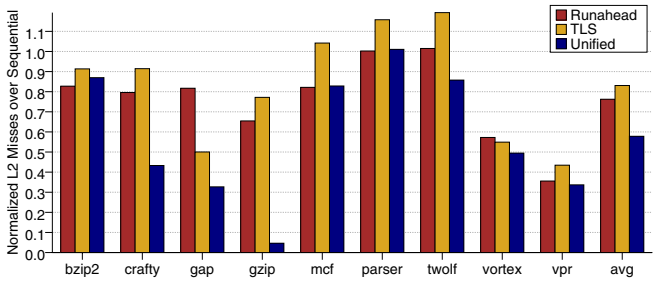

Figure 6: Normalized L2 misses over sequential execution for the commited path of the Runahead execution, TLS and our unified scheme.

Figure 7 shows the fraction of isolated and clustered misses seen *on the commited path* for the various execution models. Clustering is identified as the presence of other in-flight memory requests when the commit path suffers another L2 cache miss. This figure is then complementary to Figure 6 in explaining the prefetching effects of each model as it can capture *partial* prefetches (i.e., prefetches that do not completely eliminate a cache miss, but lead to a reduction in the waiting time). Note that Runahead execution does very well in clustering the misses and in fact is able to do much better than both TLS and our unified scheme. As noted above, this significant increase in number of outstanding memory requests with Runahead execution does not always translate into fewer L2 misses seen by the commit path (Figure 6), but in some cases it does lead to partial prefetches and improved ILP (Figure 5). Our unified scheme manages to cluster the misses much better than TLS does, leading to further benefits from partial prefetches.

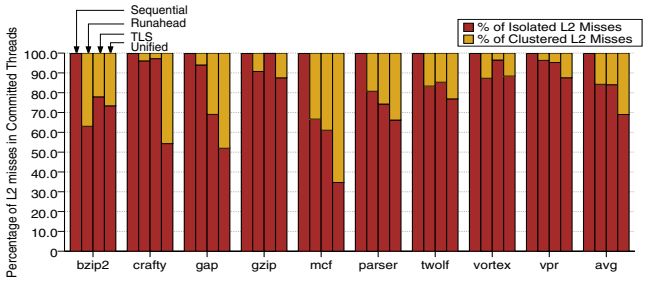

Figure 7: Breakdown of the L2 misses in isolated and clustered for sequential, Runahead, TLS, and our unified scheme.

6.2 Understanding the Trade-Offs in our Unified Scheme

6.2.1 *When to Create a HT*

As we discussed earlier in Section 3.2, there is a choice to be made whether to create a new helper thread on an L2 cache miss or at thread spawn time. Figure 8 shows the speedup of each of the two policies. As the figure clearly shows, cloning on L2 misses is always better. The reason is that it is more targeted and, thus, it does not increase the number of running threads unless there are prefetching needs (i.e., at least one actual L2 miss). This is evident from the ILP/TLP breakdown where we see that the main difference between the two scheme is mostly in the ILP benefits.

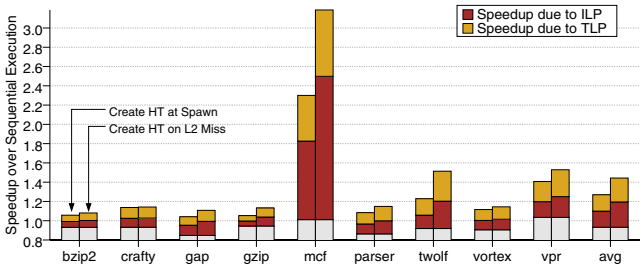

Figure 8: Impact of choosing between creating helper threads on an L2 Miss or at thread spawn

6.2.2 *Converting Existing Threads into HT vs. Spawning New HT*

An interesting trade-off is whether one should try to convert some of the normal TLS threads to helper threads (checkpointing), or whether one should create separate helper threads (cloning). As we discussed in Section 3.2, converting some TLS threads to helper threads will increase the ILP but it will do so at the expense of the TLP we can extract. Figure 9 compares the two schemes. As the figure shows, spawning a new helper thread leads to better performance in all but one case (*crafty*). It is interesting to note that in most cases the performance advantage of the cloning approach comes not only from increased TLP, as one would expect, but also from increased ILP. It is also worth noting that for the converting approach, although in all of the cases the ILP gains are significant, for some benchmarks this scheme performs even worse than the base TLS.

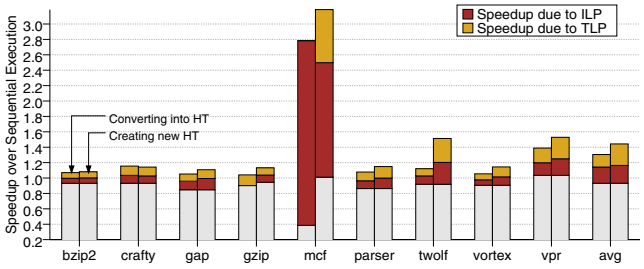

Figure 9: Converting TLS threads to Helper threads, as opposed to spawning distinct helper threads.

6.2.3 *Effect of the Load of the System*

Helper threads require resources that could otherwise be used by normal TLS threads. Figure 10(a) shows the average distribution of number of threads that exist at a given time in a four core system with TLS. We can see that almost 90% of the time there are only up to two threads running. This means that as long as we create a small number of helper threads we should not harm the extracted TLP.

In Figure 10(b) we first show a helper thread spawning policy under which we create a new helper thread on every L2 cache miss.

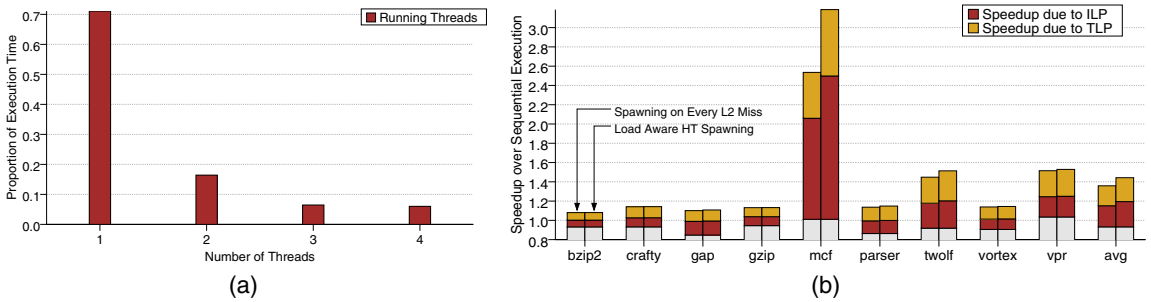

Figure 10: (a) Distribution of number of threads on a four core system. (b) Evaluating the effect of load aware HT spawning.

Under this scheme we allow multiple helper threads to co-exist. This scheme does not check if there is any available core to run the new helper thread on (if there is no free core, threads are placed in queues waiting to execute as we do with normal TLS spawns), and the clones are not killed when we spawn a new TLS thread. We compare this with our load-aware scheme, under which we only allow one helper thread to exist at a time and we kill helper threads in order to pre-empt them. The benefits of employing a load-aware scheme are more pronounced in applications with a large number of threads like *mcf* and *twolf*. The benefits come mainly from better ILP since, we are making the contention on the common L2 cache smaller and we are poluting it less.

6.2.4 *Single vs. Multiple Helper Threads*

In our approach, we chose to create a single helper thread by cloning the most speculative thread because of its fairly straightforward implementation overhead and reasonable expected performance benefits. We compare our scheme with two other schemes: one where we only allow the safe thread to create a clone, and one where we allow multiple clones to run in the system. Note that both schemes respect our load-aware policies (Section 6.2.3) so that they do not interfere negatively with the normal TLS threads. This means that for our four core system we can have at most two normal and two clone threads running concurrently (as opposed to our scheme where we will only have one clone thread). As Figure 11 shows, cloning only for the safe thread gives only a fraction of the achievable benefits. This is mainly due to worse ILP, which makes sense if we take into account that we are only prefetching for one thread. In fact, Figure 12 shows that creating a helper thread for the most speculative task performs substantially more useful prefetching than the scheme where we only create a helper thread for the safe thread. On the other hand, creating a clone for all the threads is slightly better than our scheme for all applications except *mcf*. Figure 13 shows the clustering of memory requests for all three schemes. In most cases the difference in clustering is not very significant. However, the clustering helps explain the case of *mcf* with our scheme and with cloning for all threads: even though the miss rates are comparable (Figure 12), the effect of partial prefetching is much more pronounced with the unified scheme.

6.2.5 *Effect of a Better Value Predictor*

Throughout our study we have employed a simple last value predictor. In this section perform a sensitivity analysis of our scheme on this building block. As we see in Figure 14, a better value predictor (i.e., a perfect one in this case) would lead to significantly increased benefits for the unified scheme. The improvement will come in terms of better ILP, which is justified by the improved

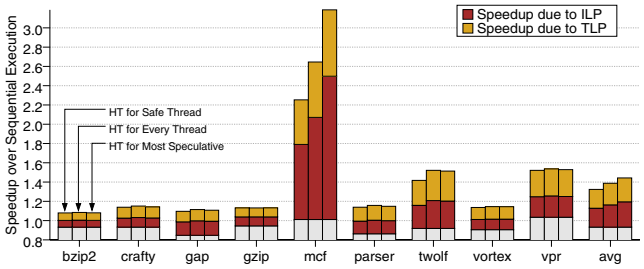

Figure 11: Comparing the effect in performance when creating multiple HT or a single HT.

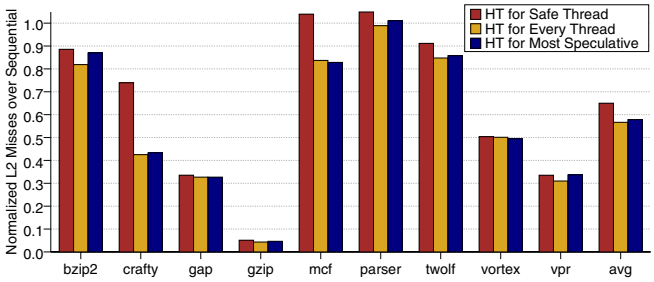

Figure 12: Normalized L2 misses over sequential execution when creating multiple HT or a single HT.

prefetching capability resulting from always following the correct path. We see that there is still ample room for improvement for at least four out of the nine benchmarks used in this paper. We intend to further investigate the effect of value prediction in future work.

6.3 Performance Model Comparison

As discussed in Section 4, the difference between our speedup breakdown model and that of [15] is that we propose to measure the actual execution time degradation of the TLS execution when running on a single core compared to the original sequential execution (S_{1p}), instead of estimating this factor with the instruction code bloat (f_{bloat}). In reality $1/S_{1p} \neq f_{bloat}$, which will lead to some innacuracy in the model of [15]. In fact, since the added TLS instrumentation consists of several memory operations and some function calls, we expect that $1/S_{1p} < f_{bloat}$ and the model of [15] will *under-estimate* the contribution of ILP.

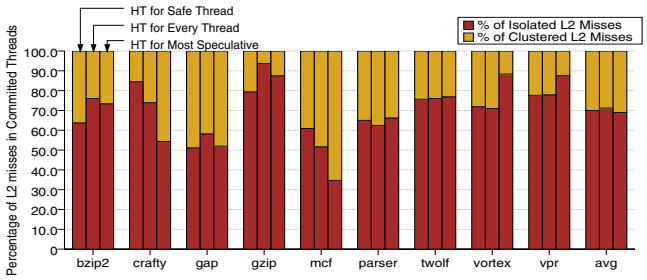

Figure 13: Breakdown of the L2 misses in isolated and clustered when creating multiple HT or a single HT.

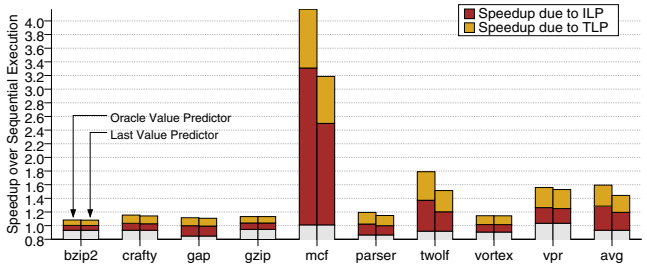

Figure 14: Performance impact of value prediction accuracy.

We measured the difference between the ILP and the TLP estimates of both models and show the results in Table 3. Although for some applications the errors are fairly small, this is not the case for some others like *gap* and *mcf* where there is a difference of 15.3% (for ILP estimation) and 13% (for TLP estimation). More importantly, in some cases the two models do not agree, so that the model proposed in [15] indicates that there is no ILP contribution (or there is a slowdown due to ILP degradation) whereas our model contradicts this. The benchmarks where this is the case are shown in Table 3 with bold. We thus believe that the extra simulations required by our model are well justified, since they provide a much clearer picture of what is happening.

Benchmark	bzip2	crafty	gap	gzip	mcf	parser	twolf	vortex	vpr	avg
% ILP Diff.	1.81	**0.45**	15.3	2.67	**1.11**	4.81	2.26	2.97	**3.49**	3.87
% TLP Diff.	5.09	7.19	0.02	2.89	13.0	9.32	5.98	6.64	2.01	5.79

Table 3: Difference of ILP and TLP benefit estimation between our performance model and the one proposed in [15]. With bold we denote cases where our model presents speedup whereas the previously proposed does not.

7. RELATED WORK

Thread Level Speculation.

Thread level speculation has been previously proposed (e.g., [7, 10, 12, 16, 17]) as a means to provide some degree of parallelism in the presence of data dependences. The vast majority of prior work on TLS systems has focused on architectural features directly related to the TLS support, such as protocols for multi-versioned caches and data dependence violation detection. All these are orthogonal to our work. In particular, we use the system in [15] as our baseline. Benefits due to prefetching were also reported for TLS systems in [15]. In our work we that by employing a unified speculative multithreading approach, one can obtain further benefits from prefetching. The work in [19] showed in a limit study that it might be beneficial to continue running some threads that are predicted to squash, so as to do prefetching. However, the benefits shown there were minimal if one considers the simplifications in the simulation infrastructure used. In general, deffering the squashes so as to prefetch is not beneficial because we negate the TLP benefits. Checkpointing TLS threads was also proposed in [5]. However, checkpointing in a TLS execution model is used to improve TLP by supporting squash and rollbacks to a checkpointed state instead of to the start of the thread. In our work we employ checkpointing in a combined execution model to allow for improved ILP while maintaining similar TLP levels. Finally, [15] also proposed a model to quantitatively break down the performance improvements between TLP and ILP. Our model is slightly more computationally intensive as it require one extra simulation run, but our results show that it is much more accurate.

Helper Threads.

It has been previously proposed to use small helper threads on otherwise idle hardware resources [3, 4, 18, 20]. There helper threads usually try to resolve highly unpredictable branches and cache misses that the main thread would have to stall upon otherwise. Because helper threads try to improve the ILP of the main thread, they fail to procure any significant benefits in applications where the out-of-order engine is able to extract most of the ILP. As we showed in previous sections, in these cases we can achieve some addtional improvements by trying to extract TLP as well.

Runahead Execution.

Runahead execution [1, 6, 13] is a scheme that generates accurate data prefetches by speculatively executing past a long latency cache miss, when the processor would otherwise be stalled. Runahead execution is similar to the helper thread schemes, although instead of using different hardware resources, it uses otherwise idle processor cycles to perform prefetching. Additionally, runahead execution does not rely on the programmer to manually extract the prefetching slices. Two more recent proposals, Checkpointed Early Load Retirement [9] and CAVA [2], build on [13] by adding value prediction. In contrast with runahead execution, correctly predicting the value of a missing load eliminates the need to rollback to a checkpoint when the load returns. The work in [2] showed, however, that most of the benefits from this scheme do not come from negating the roll-backs, but rather from the fact that by value predicting, prefetches are more accurate.

8. CONCLUSIONS

Thread Level Speculation, Helper Threads and Runahead Execution have been separately shown to improve overall performance of some sequential applications. However, given the different nature of the performance benefits provided by each model, one would expect that combining them in a unified execution model would lead to greater performance gains and over a wider range of applications compared to each model alone. Despite these opportunities no one has attempted to combine these multithreaded execution models.

In this paper we propose to combine all three multithreaded execution models in a super-set unified model that can exploit the benefits of each model depending on application characteristics. More specifically, the resulting system attempts to exploit TLP speculatively with TLS execution but when this fails or when additional

opportunities exist for exploiting ILP the system also employs a version of HT that is based on RA. We chose this model of HT so that its threads interact seamlessly with TLS threads and only small modifications to the TLS protocol and the TLS architectural support are required. In the paper we discussed in detail this interaction and how to tune the HT and TLS models to work synergystically. Another contribution of this paper is a simple methodology that allows one to model the performance gains with TLS and the unified execution model such that gains can be accurately attributed to either TLP or ILP. This methodology, then, allows one to quantitatively reason about the behavior of the execution models and to investigate tradeoffs in the unified model.

Experimental results show that our unified execution model achieves speedups of up to 41.2%, with an average of 10.2%, over an existing state-of-the-art TLS system and speedups of up to 35.2% with an average of 18.3% when compared with a flavor of runahead execution for a subset of the SPEC2000 Int benchmark suite.

9. REFERENCES

[1] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. M. Hwu. "Beating In-Order Stalls with 'Fea-Ficker' Two-Pass Pipelining." *Intl. Symp. on Microarchitecture,* pages 387-398, December 2003.

[2] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. "CAVA: Using Checkpoint-Assisted Value Prediction to Hide L2 Misses." *ACM Trans. on Architecture and Code Optimization,* vol. 3, no. 2, pages 182-208, June 2006.

[3] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. "Simultaneous Subordinate Microthreading (SSMT)." *Intl. Symp. on Computer Architecture,* pages 186-195, May 1999.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads." *Intl. Symp. on Computer Architecture,* pages 14-25, June 2001.

[5] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. "Tolerating Dependences Between Large Speculative Threads Via Sub-Threads." *Intl. Symp. on Computer Architecture,* pages 216-226, June 2006.

[6] J. Dundas and T. Mudge. "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss." *Intl. Conf. on Supercomputing,* pages 68-75, July 1997.

[7] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 58-69, October 1998.

[8] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. "Recovery Requirements of Branch Prediction Storage Structures in the Presence of Mispredicted-Path Execution." *Intl. Journal of Parallel Programming,* vol. 25, 1997.

[9] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martínez. "Checkpointed Early Load Retirement." *Intl. Symp. on High-Performance Computer Architecture,* pages 16-27, February 2005.

[10] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor." *Intl. Conf. on Supercomputing,* pages 85-92, June 1998.

[11] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS Compiler that Exploits Program Structure." *Symp. on Principles and Practice of Parallel Programming,* pages 158-167, March 2006.

[12] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing,* pages 365-372, June 1999.

[13] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. "Runahead Execution: An Alternative to Very Large Instruction Windows." *Intl. Symp. on High-Performance Computer Architecture,* pages 129-140, February 2003.

[14] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. "SESC simulator." *http://sesc.sourceforge.net*.

[15] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas "Tasking with Out-Of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation." *Intl. Conference on Supercomputing,* pages 179-188, June 2005.

[16] G. S. Sohi, S. E. Breach and T. N. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture,* pages 414-425, June 1995.

[17] J. G. Steffan and T. C. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization." *Intl. Symp. on High-Performance Computer Architecture,* pages 2-13, February 1998.

[18] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving Both Performance and Fault Tolerance." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 257-268, October 2000.

[19] F. Warg. "Techniques to Reduce Thread-Level Speculation Overhead." *PhD Thesis, Department of Computer Science and Engineering, Chalmers University*, 2006.

[20] C. Zilles and G. Sohi. "Execution-Based Prediction Using Speculative Slices." *Intl. Symp. on Computer Architecture,* pages 2-13, June 2001.