

# Energy-Efficient Thread-Level Speculation on a CMP

Jose Renau<sup>†</sup> Karin Strauss Luis Ceze Wei Liu Smruti Sarangi James Tuck Josep Torrellas

<sup>†</sup>Dept. of Computer Engineering, University of California Santa Cruz

<http://masc.soe.ucsc.edu>

Dept. of Computer Science, University of Illinois at Urbana-Champaign

<http://iacoma.cs.uiuc.edu>

Draft. To appear in IEEE Micro Special Issue on: Top Picks  
from Computer Architecture Conferences, Jan.-Feb. 2006

## 1 Introduction

Substantial effort is currently being devoted to speeding up hard-to-parallelize non-numerical applications such as SPECint codes. Designers build sophisticated out-of-order processors, with carefully-tuned execution engines and memory subsystems. Unfortunately, these systems tend to combine high design complexity with diminishing performance returns, motivating the search for design alternatives.

One such alternative is Thread-Level Speculation (TLS) on a Chip Multiprocessor (CMP) (e.g., [7, 8, 11, 17, 18, 19, 20]). Under TLS, these hard-to-analyze applications are partitioned into tasks, which are then optimistically executed in parallel, hoping to avoid any cross-task dependence violation. Special hardware support monitors the tasks's data accesses and detects violations at run time. Should one occur, the hardware transparently rolls back the incorrect tasks and, after repairing the state, restarts them. See Sidebar 1 for TLS principles.

A major concern regarding TLS CMPs is energy efficiency. TLS is suspected of being too energy inefficient to seriously challenge conventional processors. The rationale is that aggressive speculative execution is not the best course at a time when processors are primarily constrained by energy and power consumption.

In this paper, we refute the claim that TLS is energy inefficient. We identify and quantify the main sources of energy consumption in TLS. We then propose simple energy-saving optimizations for TLS. As a result, this paper is the first one to show that a TLS CMP can be a desirable design for high-performance, power-constrained processors, even under the very challenging SPECint codes.

Fundamentally, the energy cost of TLS can be kept modest by using a lean, high-performing TLS CMP microarchitecture

and by minimizing wasted TLS work. Our TLS CMP design relies on efficient microarchitecture for tasking with *out-of-order task spawn*, and a novel TLS compiler. See Sidebar 2 for details on out-of-order task spawn. The resulting TLS CMP provides a better energy-performance trade-off than a wider-issue superscalar.

## 2 Sources of Energy Consumption in TLS and Energy-Saving Optimizations

Enhancing an  $N$ -issue superscalar into a CMP with several  $N$ -issue cores and TLS support causes the energy consumption to increase. We loosely call the increase *the energy cost of TLS* ( $\Delta E_{TLS}$ ). Of course, a portion of  $\Delta E_{TLS}$  simply comes from having multiple cores and caches on chip, and from inefficiencies of parallel execution. However, most of  $\Delta E_{TLS}$  is due to TLS-specific sources.

We classify TLS-specific sources of energy consumption into four groups: (1) task squashing, (2) hardware structures in the cache hierarchy needed for data versioning and dependence checking, (3) additional traffic in the memory system due to these same two effects, and (4) additional dynamic instructions induced by TLS. These sources are detailed in Column 2 of Table 1.

TLS-Specific Source		Optimization
Task squashing	Work of the tasks that get squashed	StallSq, TaskOpt
	Task squash operations	
Hardware structures in the cache hierarchy for data versioning and dependence checking	Storage & logic for data version IDs and access bits	Indirect
	Tag-group operations	NoWalk
Traffic due to data versioning and dependence checking	Evictions and misses due to higher cache pressure	—
	Selection & combination of multiple versions	TrafRed
	Fine-grain data dependence tracking	
Additional dynamic instructions induced by TLS	Side-effects of breaking the code into tasks	TaskOpt
	TLS-specific instructions	

Table 1: Main TLS-specific sources of energy consumption and proposed energy-centric optimizations.

We also propose optimizations to reduce these sources. We are not interested in optimizations that reduce energy by first improving performance; we assume these are already present in a performance-centric TLS design. Instead, we focus on *energy-centric* optimizations: while they do not increase performance noticeably (in fact, they may slightly reduce it), they reduce energy consumption significantly.

Such optimizations are based on three guidelines: (1) reduce the number of checks, (2) reduce the cost of individual checks, and (3) eliminate work with low performance returns. The names of our optimizations are shown in Column 3 of Table 1. In the following, we discuss the sources and the optimizations.

### 2.1 Task Squashing

A TLS source of energy consumption is the work of tasks that ultimately get squashed. In the TLS CMP that we evaluate in Section 4, 22.6% of all graduated instructions belong to such tasks. Note, however, that not all such work is wasted: a squashed task may bring useful data into the caches.

The actual squash operation also consumes energy. However, in our system, the frequency of squashes is only 1 per 3211 instructions on average. Consequently, the total energy consumed by the actual squash operations is negligible.

### 2.1.1 Optimizations: Task Stalling (StallSq) and Eliminating Energy-Inefficient Tasks (TaskOpt)

A first optimization to reduce the number of instructions in squashed tasks is to limit the number of times that a task is allowed to restart after a squash. After a task has been squashed  $N$  times, it is not given a CPU again until it becomes non-speculative. To select  $N$ , we performed experiments while always restarting tasks after being squashed. We found that 73.0% of the tasks are never squashed, 20.6% are squashed once, 4.1% twice, 1.4% three times, and 0.9% four times or more. Restarting a task after its first squash can be beneficial, as the cache has been warmed up. Restarting after further squashes delivers diminishing returns. Consequently, we reset and stall a task after its second squash.

The second optimization involves not spawning inefficient tasks (*TaskOpt*). We do this with a profiler pass in our compiler. The profiler includes a simple model that identifies tasks that are often squashed and do not help by warming the cache. Such tasks are not spawned. For the baseline TLS architecture, the model tries to minimize the overall duration of the program. The energy-centric optimization is to change the model into one that tries to minimize the product  $Energy \times Delay^2$  for the program. This change has a significant impact: on average, the profiler eliminates 39.9% of the static tasks in the baseline TLS, and 49.2% in energy-centric mode.

This second optimization is also enhanced with a static compilation pass that aggressively prunes tasks that are either very small or whose spawn point has not been moved up in the code much (and therefore offer little parallelism). We use threshold values to decide the size and spawn hoist distance of the tasks to prune. The thresholds used in energy-centric mode are more aggressive than their baseline TLS counterparts. This static pass eliminates 36.1% of the static tasks in energy-centric mode compared to 34.7% in performance-centric mode.

## 2.2 Hardware Structures for Data Versioning and Dependence Checking

TLS systems need to maintain data versioning and perform dependence checking. Data versioning is needed when the cache hierarchy needs to hold multiple versions of the same datum. Such versions appear when speculative tasks have WAW or WAR dependences with predecessor tasks. The version created by the speculative task is buffered, typically in the processor's cache. If multiple speculative tasks co-exist in the same processor, a cache may have to hold multiple versions of the same datum. In such cases, data versions are identified by tagging the cache lines with a version ID — in our case, a local ID (LID) (See Sidebar 1).

To perform dependence checking, caches record how each datum was accessed. Typically, this is supported by augmenting each cached datum with two access bits: an exposed-read and a write bit. They are set on an exposed read and a write, respectively.

The LID and access bits are read or updated in hardware in a variety of cache access operations. For example, on an external access to a cache, the LID of an address-matching line in the cache is compared to the ID of the incoming message. From the comparison and the value of the access bits, the cache may conclude that a violation occurred, or can instead supply the data

normally.

A distinct use of these TLS structures is in *tag-group* operations. They involve changing the tag state of groups of cache lines. There are three main cases. First, when a task is squashed, its cache lines need to be invalidated. Second, in eager-commit systems [4], when a task commits, all its dirty cache lines are merged with main memory through write backs [8] or ownership requests [18]. Finally, in lazy-commit systems [4], when a cache has no free LIDs left, it needs to recycle one. This is typically done by selecting a long-committed task and writing back all its dirty cache lines to memory. Then, that task’s LID becomes free and can be re-assigned.

These TLS tag-group operations often induce significant energy consumption. Specifically, for certain operations, some schemes use a hardware finite state machine (FSM) that, periodically and in the background, repeatedly walks the tags of the cache. For example, to recycle LIDs in [12], a FSM periodically selects the LID of a committed task from the LID Table, walks the cache tags writing back to memory the dirty lines of that task, and finally frees up the LID. The FSM operates in the background, using free cache cycles. As another example, to commit a task in [18], a special hardware module sequentially requests ownership for a group of cache lines whose addresses are stored in a buffer. In the meantime, the processor stalls. Therefore, execution takes longer and consumes more energy. Finally, some schemes use “one-shot” hardware signals that can change the tag state of a large group of lines in a handful of cycles. For example, this is done to invalidate the lines of a squashed task. Such hardware is reasonable when the cache can hold data for only a single or very few speculative tasks [7]. However, in caches with many versions, it is likely to adversely affect the cache access time. For example, in our system, we use 6-bit LIDs per cache line.

### **2.2.1 Optimizations: Lazy Group Operations (NoWalk) and Reducing the Cost of Checks (Indirect)**

We enhance the LID Table design described in Sidebar 1 as follows. Each table entry is extended with use information on the corresponding task: the number of lines that the task still has in the cache, and whether the task has been killed or committed. As discussed in [14], these bits are never accessed in the critical path of an L1 cache hit.

With this extra information, all the tag-group operations described above are performed lazily and very efficiently. A task squash or commit only involves setting a Killed or Committed bit, respectively, in the LID Table. As lines belonging to a squashed or a committed task are eliminated from the cache due to replacements, the corresponding count in the LID Table is decremented. When the count reaches zero, its associated LID becomes unused and can be recycled. Consequently, LID recycling is also very fast.

We call this energy-efficient design of the LID Table the *NoWalk* optimization. It largely avoids the eager walk of the cache tags in any tag-group operation. It only activates a background walk of the tags when there is only one free LID left. With this optimization, we occasionally may have to stall due to temporary lack of LIDs. However, we eliminate many tag checks. In contrast, the baseline TLS architecture recycles LIDs in an aggressive manner, inspired in previous work [12]. Specifically, a hardware FSM periodically walks the tags of the cache in the background when the cache is idle. It invalidates lines of killed

tasks and writes back dirty lines of long-committed tasks, therefore eagerly freeing up LIDs. This design never runs out of LIDs but consumes energy with many checks.

The second optimization, *Indirect*, simply consists in tagging cache lines with short LIDs rather than with global task IDs. As a result, each tag check consumes less energy. This approach of using indirection is well known [18]. Consequently, we already use it in the baseline TLS and do not evaluate its impact.

## 2.3 Additional Traffic for Data Versioning and Dependence Checking

A TLS CMP system generates more traffic beyond the private L1 caches than a superscalar. While some of the increase is the result of parallel execution, there are three main TLS-specific sources of additional traffic (Table 1). First, caches do not work as well. Caches often have to retain lines from older tasks that ran on the processor and are still speculative. Only when such tasks become safe can the lines be evicted. As a result, effective cache space for the currently-running task is reduced. This causes additional misses.

Second, the presence of multiple versions of the same line in the system causes additional messages. Specifically, when a processor requests a line, multiple versions of it may be provided, and the coherence protocol then selects what version to use. Similarly, when a committed version of a line is to be evicted from a cache, the protocol first invalidates all the other cached versions of the line that are older — they cannot remain cached anymore.

Finally, it is desirable that the speculative cache coherence protocol track dependences at a fine grain. To see why, recall that these protocols typically track dependences by using the write and exposed-read bits. If these bits are kept per line, lines that exhibit false sharing may appear to be involved in data dependence violations and, as a result, cause squashes [3]. For this reason, many TLS proposals keep access information at a finer grain, such as per word. Unfortunately, per-word dependence tracking may induce higher traffic: a distinct message (such as an invalidation) may need to be sent for every word of the line.

### 2.3.1 Optimization: Lower Traffic to Check Version-IDs (TrafRed)

To reduce the number of version-ID checks needed and, therefore, the traffic, we extend cache lines with a *Newest* and an *Oldest* bit. Every time that a line is loaded into a cache, we set the Newest and/or Oldest bit if the line contains the latest and/or the earliest cached version, respectively, of the corresponding address. As execution proceeds, Newest may be reset on an access by another task.

With this support, many messages are eliminated. Specifically, when a processor writes to a line cached in non-exclusive state, the baseline TLS checks all the caches with a version of the requested line to detect any exposed read to the line from a more speculative task — such event would cause a squash. With our optimization, if the written line has the Newest bit set, there is no need to check other caches for exposed reads. Moreover, when a processor displaces from its cache a committed line, the baseline TLS checks all the caches with a version of the line to invalidate older versions of the line — such versions cannot remain cached anymore. With our optimization, if the displaced line has the Oldest bit set, there is no need to check other caches

for older versions.

## 2.4 Additional Dynamic Instructions Due to TLS

TLS systems with compiler-generated tasks such as ours often execute more dynamic instructions than non-TLS systems. This is the case even counting only tasks that are not squashed. In our system, the increase is 12.5%. These additional instructions come from two sources (Table 1): side-effects of breaking the code into tasks and, less importantly, TLS-specific instructions.

The first source dominates. It accounts for 88.3% of the increase. One reason for this source is that conventional compiler optimizations are not very effective at optimizing code across task boundaries. Therefore, TLS code quality is lower than non-TLS code. In addition, in CMPs such as the ones considered here, where processors communicate only through memory, the compiler must spill registers across task boundaries, adding extra instructions.

TLS-specific instructions are the other source. They include task spawn and commit instructions. These instructions contribute with 11.7% of the instruction increase.

### 2.4.1 Optimization: Eliminating Energy-Inefficient Tasks (TaskOpt)

The same compiler and profiler optimization described in Section 2.1.1 to reduce the energy waste of squashed tasks (TaskOpt) can also aid in reducing the additional dynamic instructions due to TLS.

## 3 POSH: TLS Compilation Infrastructure

We have developed *POSH*, a novel compiler that generates fully-automated TLS code out of sequential, integer applications [10, 15]. By default, POSH generates code with *out-of-order task spawn* (See Sidebar 2), although it can also restrict code generation to in-order task spawn. POSH adds several passes to gcc 3.5 (an early version of the latest gcc 4.0), which uses a static single assignment tree as the high-level intermediate representation [5]. Building on this software allows us to leverage a complete compiler infrastructure. Also, working at this high level is better than using a low-level representation such as RTL: we have better information and it is easier to perform pointer and dataflow analysis. At the same time, our transformations are much less likely to be affected by unwanted compiler optimizations than if we were working at the source-code level.

### 3.1 Task Generation and Hoisting

POSH uses the following modules as potential tasks: subroutines from any nesting level, their continuations, and loop iterations from any loop nesting level. Recursion is handled seamlessly. Under out-of-order task spawn mode (the default one), all subroutines and loop iterations that are larger than a certain size are potentially selected. Under in-order task spawn mode, the compiler is more careful, since a task can only have a single child. Consequently, the in-order pass analyzes all the files in the program and generates a complete task call graph. Then, using heuristics about task size and overheads, it eliminates tasks from the graph

until it can guarantee that each task only has a single child.

Once the tasks are selected, the compiler inserts spawn instructions, and tries to hoist them to boost parallelism. A spawn is hoisted as far up as we can, but not above statements that can cause data or control dependence violations. Under in-order spawn, a spawn cannot be hoisted above the caller task.

A final task clean-up pass looks for spawns that were hoisted only a handful of instructions. In this case, the spawn is eliminated, and the two corresponding tasks integrated into one. This reduces overheads.

As an example, Figure 1 shows how the compiler generates out-of-order tasks out of a subroutine and its continuation. Chart (a) shows the dynamic execution into and out of the subroutine. The compiler marks the subroutine and continuation as tasks, and inserts two spawn instructions in the caller (Chart (b)). Then, it hoists the spawn for the continuation (Chart (c)) and subroutine (Chart (d)). In Chart (e), the clean-up pass eliminates the subroutine spawn because it had little hoisting.

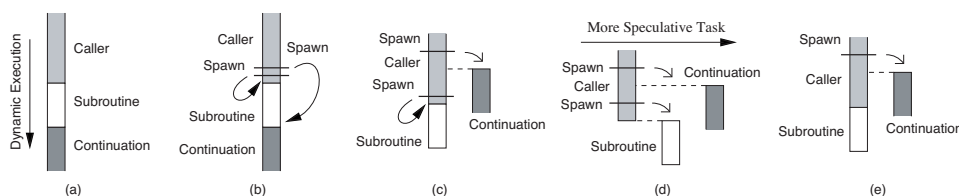


Figure 1: Generating tasks out of a subroutine and its continuation.

### 3.2 Task Profiling

POSH includes a profiling pass that, using a simple model, identifies additional tasks that should be eliminated. Such tasks are those that, due to squashes, are unlikely to be beneficial.

The profiler runs the binary sequentially, using the Train data set for SPECint codes. As the profiler executes a task, it records the variables written. When it executes tasks that would be spawned earlier, it compares the addresses read against those written by predecessor tasks. With this, it can detect potential run-time violations. The profiler also models a very simple cache to estimate the number of cache misses. For performance, cache timing is not modeled. On average, the profiler takes around 5 minutes to run on a 3 GHz Pentium 4. Details on the profiler can be found in [10, 15].

## 4 Evaluation

We compare a TLS CMP to a non-TLS chip that has a single processor of the same or wider issue width. We use execution-driven simulations [13], with detailed models of out-of-order superscalars and advanced memory hierarchies, enhanced with models of dynamic and leakage energy from Watch [1], Orion [22], and HotLeakage [23].

The TLS CMP that we model has four 3-issue cores with private L1 caches and a shared L2 cache. We call the chip *TLS4-3i*. The non-TLS chips have a single superscalar with on-chip L1 and L2 caches. We consider two such chips: one has a 6-issue superscalar (*Uni-6i*) and the other a 3-issue superscalar (*Uni-3i*). The *TLS4-3i* and *Uni-6i* chips have approximately the same

area, as can be estimated from [9, 16]. We also simulate a TLS CMP with only in-order task spawning. We call the system *TLS4-3i InOrder*.

We measure SPECint 2000 applications with the Reference data set. All the application code is measured, not just the more parallel sections such as loops. *Uni-3i* and *Uni-6i* run the binaries compiled with our TLS passes disabled.

#### 4.1 The Energy Cost of TLS ( $\Delta E_{TLS}$ )

Figure 2 characterizes the energy cost of TLS ( $\Delta E_{TLS}$ ), which Section 2 defined as the difference between the energy consumed by our TLS CMPs and *Uni-3i*. For each application, the figure shows six bars. They refer to the total energy consumed by the chip without any of our energy optimizations (*NoOpt*); with individual optimizations enabled (*StallSq*, *TaskOpt*, *NoWalk*, and *TrafRed*); and with all four optimizations applied (*TLS4-3i*). For each application, the bars are normalized to the energy consumed by *Uni-3i*. Consequently, the difference between the top of the bars and 1.00 is  $\Delta E_{TLS}$ .

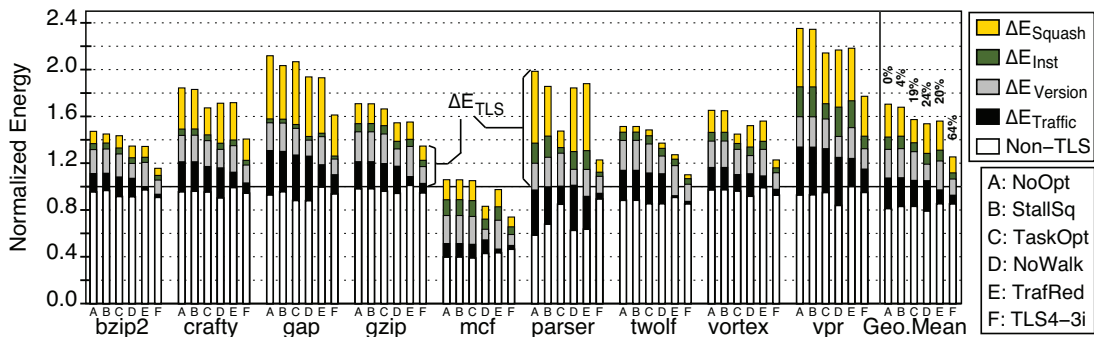


Figure 2: Energy cost of TLS ( $\Delta E_{TLS}$ ) for our 4-core TLS CMP chip with and without energy-centric optimizations. The percentages listed above the average bars are the decrease in  $\Delta E_{TLS}$  when the optimizations are enabled.

Each bar is broken into the contributions of the TLS-specific sources of energy consumption listed in Table 1. These include task squashing ( $\Delta E_{Squash}$ ), additional dynamic instructions in non-squashed tasks ( $\Delta E_{Inst}$ ), hardware for data versioning and dependence checking ( $\Delta E_{Version}$ ), and additional traffic ( $\Delta E_{Traffic}$ ). The rest of the bar (*Non-TLS*) is energy that we do not attribute to TLS.

Ideally, *Non-TLS* should equal 1. In practice, this is not the case because a given program runs on the TLS CMP and on *Uni-3i* at different speeds and temperatures. As a result, the “non-TLS” dynamic and leakage energy varies across runs, causing *Non-TLS* to deviate from 1.

Focusing on the *NoOpt* bars, we see that the energy cost of *unoptimized* TLS ( $\Delta E_{TLS}$ ) is significant. On average, unoptimized TLS adds 70.4% to the energy consumed by *Uni-3i*. We also see that all four TLS sources of energy consumption contribute noticeably. Of them, task squashing ( $\Delta E_{Squash}$ ) consumes the most energy.

#### 4.2 The Impact of Our Energy-Centric Optimizations

The rest of the bars in Figure 2 show the impact of our optimizations on the TLS energy sources. Each optimization effectively reduces the TLS energy sources that it is expected to minimize from Table 1. This is best seen from the average bars.

Consider *TaskOpt* first. In Figure 2, it reduces  $\Delta E_{Squash}$  and  $\Delta E_{Inst}$  — its targets in Table 1. *TaskOpt* saves energy because it reduces the fraction of squashed instructions from 22.6% to 17.6%, and decreases the additional dynamic instructions in non-squashed tasks from 12.5% to 11.9% [14].

Consider now *NoWalk*. In Figure 2, it mostly reduces  $\Delta E_{Version}$  — its target in Table 1. *NoWalk* reduces the number of tag accesses relative to *Uni-3i* from 3.3 times to 2.2 times [14].

If we consider *TrafRed* in Figure 2, we see that it mostly reduces  $\Delta E_{Traffic}$  — again its target in Table 1. *TrafRed* reduces the traffic relative to *Uni-3i* from 19.6 times to 5.6 times on average [14].

Finally, *StallSq* only addresses  $\Delta E_{Squash}$ , which is its target in Table 1. It has only a small impact in Figure 2.

This analysis shows that *TaskOpt*, *NoWalk*, and *TrafRed* effectively reduce different energy sources, and that the three techniques combined cover all sources considered. When we combine all four optimizations into *TLS4-3i*, we are able to eliminate on average 64% of  $\Delta E_{TLS}$ . Compared to the overall on-chip energy consumed by *NoOpt*, this is a very respectable energy reduction of 26.5%.

If we measure the section of the resulting *TLS4-3i* bar that is over 1.00, we see the *energy cost* of TLS after optimization. Such cost is on average only 25.4%. We feel that this is a remarkably low figure. Moreover, it can be shown that the applications have only been slowed down on average by less than 2%.

### 4.3 Overall Speedups and Power Consumption

Finally, we assess the speed and power consumption of our energy-optimized TLS CMP (*TLS4-3i*). We compare *TLS4-3i* to the single-core chips *Uni-3i* and *Uni-6i*, and to our TLS CMP where the compiler only generates tasks with in-order spawning (*TLS4-3i InOrder*). Figure 3-(a) shows the application speedup of these architectures relative to *Uni-3i*, while Figure 3-(b) shows the average power consumed during execution. As a reference, the arithmetic mean of the average IPC of the applications on *TLS4-3i* is 1.38.

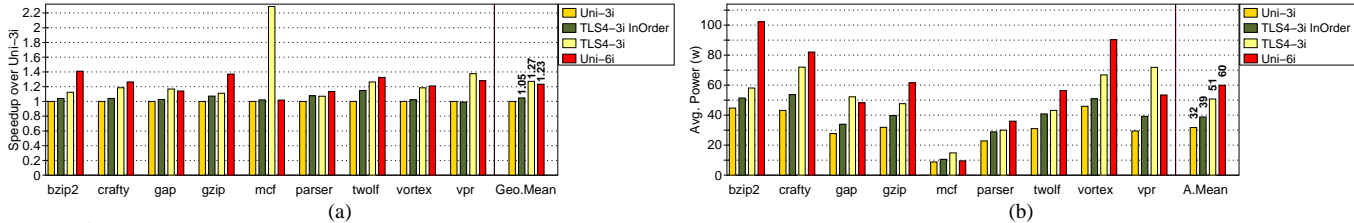


Figure 3: Execution speedup relative to *Uni-3i* (a), and average power consumption (b) for different chips. Note that the mean used for speedups is the geometric one.

Figure 3-(a) shows that, on average, *TLS4-3i* delivers a speedup of 1.27 over *Uni-3i*. This shows that POSH successfully extracts good tasks from these irregular codes. The figure also shows that *TLS4-3i* is on average slightly faster than *Uni-6i*. The speculative parallelism enabled by *TLS4-3i* in these hard-to-parallelize codes is more effective than doubling the issue width. This is a good result, especially because it conservatively assumes the same frequency for both chips.

Figure 3-(a) also compares *TLS4-3i* to *TLS4-3i InOrder*. The bars show that an environment with in-order spawning handicaps

TLS. TLS only obtains a 1.05 average speedup. Consequently, we recommend supporting out-of-order task spawn.

Note that the *TLS4-3i* speedup for *mcf* is very high. The reason is that *mcf* benefits from constructive data prefetching into L2 by TLS tasks. Without considering *mcf*, the geometric mean of *TLS4-3i*'s speedup is 1.18, which is still comparable to *Uni-6i*'s.

On the other hand, Figure 3-(b) shows that the on-chip power consumed by *TLS4-3i* is typically lower than *Uni-6i*'s. On average, it is 15% lower. Moreover, it never reaches the high values that *Uni-6i* dissipates in some applications. This shows that a TLS CMP is energy-efficient: on average, *TLS4-3i* is slightly faster than *Uni-6i* while consuming 15% less power.

## 5 Concluding Remarks

We hope that this work helps propel TLS into mainstream microprocessors. CMPs are attractive because they are more energy-efficient, more scalable, and less complex than wide-issue superscalars. Moreover, they have an advantage for explicitly-parallel codes. In this paper, we showed that CMPs with TLS support can also speed up hard-to-parallelize SPECint codes with energy and power efficiency.

## Acknowledgments

This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

## References

- [1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.
- [2] M. Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.
- [3] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, pages 13–24, June 2000.
- [4] M. J. Garzarán, M. Prvulovic, J. M. Llbería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 191–202, February 2003.
- [5] SSA for Trees - GNU Project, May 2003. "[http://www.gccsummit.org/2003/view\\_abstract.php?talk=2](http://www.gccsummit.org/2003/view_abstract.php?talk=2)".
- [6] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.
- [7] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.
- [8] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.
- [9] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, December 2003.
- [10] W. Liu, J. Tuck, L. Ceze, J. Renau, K. Strauss, and J. Torrellas. POSH: A Profiler-Enhanced TLS Compiler that Leverages Program Structure. In *Proceedings of the 2nd Watson Conference on Interaction Between Architecture, Circuits and Compilers (P=ac2)*, September 2005.
- [11] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *International Conference on Supercomputing*, pages 365–372, June 1999.
- [12] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *International Symposium on Computer Architecture*, pages 204–215, June 2001.
- [13] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. <http://sesc.sourceforge.net>.
- [14] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-Level Speculation on a CMP Can Be Energy Efficient. In *International Conference on Supercomputing*, June 2005.

- [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. In *International Conference on Supercomputing*, June 2005.
- [16] P. Shivakumar and N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.
- [17] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, pages 414–425, June 1995.
- [18] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *International Symposium on Computer Architecture*, pages 1–12, June 2000.
- [19] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.
- [20] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.
- [21] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.
- [22] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *International Symposium on Microarchitecture*, December 2002.
- [23] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia, Department of Computer Science, March 2003.

## SIDEBAR 1: Thread-Level Speculation (TLS) Principles

In TLS, a sequential program is divided into tasks, which are then executed in parallel, hoping not to violate sequential semantics. The sequential code imposes a task order and, therefore, we use the terms predecessor and successor tasks. The safe (or non-speculative) task precedes all speculative tasks. As tasks execute, special hardware support checks that no cross-task dependence is violated. If any is, the incorrect tasks are squashed, any polluted state is repaired, and the tasks are re-executed.

**Cross-Task Data Dependence Violations.** Data dependences are typically monitored by tracking, for each individual task, the data written and the data read with exposed reads. An exposed read is a read that is not preceded by a write to the same location within the same task. A data dependence violation occurs when a task writes a location that has been read by a successor task with an exposed read. Dependence violations lead to task squashes, which involve discarding the work produced by the task.

**State Buffering.** Stores issued by a speculative task generate speculative state that cannot be merged with the safe state of the program because it may be incorrect. Such state is stored separately, typically in the cache of the processor running the task. If a violation is detected, the state is discarded. Otherwise, when the task becomes non-speculative, the state is allowed to propagate to memory. When a non-speculative task finishes execution, it commits. Committing informs the rest of the system that the state generated by the task is now part of the safe program state.

**Data Versioning.** A task has at most a single version of any given variable. However, different speculative tasks that run concurrently in the machine may write to the same variable and, as a result, produce different versions of the variable. Such versions must be buffered separately. Moreover, readers must be provided the correct versions. Finally, as tasks commit in order, data versions need to be merged with the safe memory state also in order.

**Multi-Versioned Caches.** A cache that can hold state from multiple tasks is called multi-versioned [3, 6, 18]. There are two performance reasons why multi-versioned caches are desirable: they avoid processor stall when tasks are imbalanced, and enable lazy commit.

If tasks have load imbalance, a processor may finish a task and the task still be speculative. If the cache can only hold state for a single task, the processor has to stall until the task becomes safe [4]. An alternative is to move the task’s state to some other



Figure 4-(b) shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while a more speculative task is spawned to execute the continuation.

Consider now the case of nested subroutines. In Figure 4-(c), the safe task first spawns a task for the continuation of subroutine *S1*. Then, it enters *S1*, spawns a new task for the continuation of *S2*, and executes *S2* until its end. In this case, an individual task spawns multiple correct tasks. If so, correct tasks are spawned in strict reverse order compared to sequential execution. Specifically, while the task order (from less to more speculative) is *S2 Cont.* and then *S1 Cont.*, these tasks are spawned in reverse order. We call this approach *out-of-order* spawn. The same effect occurs in tasks built out of iterations of nested loops.

The main advantage of out-of-order spawning is that it enables more task parallelism: two code sections that are far-off in sequential execution can be executed in parallel *before* some of their intervening code sections have *even been spawned*. The main shortcoming is that, since all tasks can spawn and parallelism expands in unexpected parts of the task tree dynamically, in decentralized architectures such as CMPs, it becomes hard to maintain two cornerstones of TLS: task ordering and efficient resource allocation.

Task ordering is required in several time-critical TLS operations. Specifically, a task needs to know its immediate successor, to communicate the commit token or a squash signal. Moreover, any communication between two tasks requires knowing the tasks' relative order, which determines whether a violation is triggered. Unfortunately, with out-of-order spawn, high-speed ordering of tasks is difficult.

Efficient allocation of resources such as CPU or cache space is crucial for TLS performance. Ideally, resources should be assigned to tasks that have a high chance to commit. However, with out-of-order spawning, there may be highly-speculative tasks that have been running for a long time. In this case, if we want to spawn a less speculative task and there are no free CPUs, should we kill the highly-speculative tasks?

## 5.1 Proposed Novel Microarchitectural Mechanisms

We propose three microarchitectural mechanisms to enable high-speed tasking with out-of-order spawn in a TLS CMP [15].

**Splitting Timestamp Intervals.** Under in-order task spawn, recording task order is easy: since tasks are created in order, a parent gives to its child its timestamp plus one. Under out-of-order task spawn, we propose to represent a task with a *Timestamp Interval*, given by a *Base* and a *Range* timestamp ( $\{B,R\}$ ). On a task spawn, the parent splits its timestamp interval in two pieces: the higher-range subinterval is given to the child (since it is more speculative), while the lower-range subinterval is kept by the parent. With this support, the parent can successively provide timestamps to less and less speculative children.

**Immediate Successor List.** Under in-order task spawn, it is easy for a task to find its immediate successor: the task itself spawned its immediate successor. Under out-of-order task spawn, identifying the immediate successor is not straightforward. For example, in Figure 4-(c), if task *S2 Cont.* is squashed, it is not trivial for it to identify and squash task *S1 Cont.*, which was spawned before and independently of it. Consequently, we propose that the tasks dynamically link themselves in hardware in a list according to their sequential order. We call this list the *Immediate Successor (IS)* list. To build the IS list, each task has a

hardware pointer called the IS pointer. On a spawn, the child copies the parent's IS pointer, and the parent sets its IS pointer to point to the child.

**Dynamic Task Merging.** Under out-of-order task spawn, highly-speculative tasks may hog resources and starve less speculative tasks that are spawned later. To address this issue, we propose *Dynamic Task Merging*. It consists of transparent, hardware-driven merging of two consecutive tasks at run time based on dynamic load conditions. Under task merging, running tasks can be merged, therefore freeing resources for less speculative tasks. Alternatively, a task can skip the spawn instruction for a child, therefore merging with its child, rather than killing an already-running task.