Thread-Level Speculation on a CMP Can Be Energy Efficient

Jose Renau   Karin Strauss[†]   Luis Ceze[†]   Wei Liu[†]   Smruti Sarangi[†]   James Tuck[†]   Josep Torrellas[†]

Dept. of Computer Engineering, University of California Santa Cruz
renau@soe.ucsc.edu

[†]Dept. of Computer Science, University of Illinois at Urbana-Champaign
{kstrauss,luisceze,liuwei,sarangi,jtuck,torrellas}@cs.uiuc.edu

ABSTRACT

While Chip Multiprocessors (CMP) with Thread-Level Speculation (TLS) have become the subject of intense research, processor designers in industry have reservations about their practical implementation. An often cited complaint is that TLS is too energy-inefficient to compete against conventional superscalars.

This paper challenges the commonly-held view that TLS is energy inefficient. We identify the main sources of dynamic energy consumption in TLS. We then present very simple energy-centric optimizations to a TLS CMP architecture that cut the energy cost of TLS by over 60% on average with minimal performance impact. This represents a 26.5% reduction of the total on-chip energy. The resulting TLS CMP with 3-issue cores speeds-up full SpecInt 2000 codes (not just loops) by 1.27 times on average, while holding the energy cost of TLS to only 25.4%. The TLS CMP is slightly faster than a 6-issue superscalar at the same frequency, while consuming only 85% of its total on-chip power for these very challenging applications.

1 Introduction

Substantial research 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) [5, 8, 9, 10, 13, 21, 22, 25, 26]. Under TLS, these hard-to-analyze applications are carefully partitioned into tasks, which are then optimistically executed in parallel, hoping that no data or control dependence will be violated. A hardware safety net monitors the tasks' control flow and 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.

Published results show that TLS CMPs can speed up difficult non-numerical applications (e.g., [5, 9, 10, 29]). This is significant because CMPs are attractive platforms; they provide a low-complexity, energy-efficient architecture, and have a natural advantage for explicitly-parallel codes.

Unfortunately, processor designers in industry have reservations about the practical implementation of TLS. In particular, it is felt that TLS is too energy-inefficient to seriously challenge superscalars. The rationale is that aggressive speculative execution is not the best course at a time when processors are primarily constrained by energy and power issues. Our initial experiments, shown in Figure 1, appear to agree: assuming constant frequency, a high-performance TLS CMP with four 3-issue cores is slightly faster than a 6-issue superscalar for SpecInt codes, but consumes on average 15% more on-chip power. Clearly, before TLS CMPs can be considered, their energy and power requirements must be shown to be at least competitive with wide-issue superscalars.

This is the first paper that addresses the problem of energy and power consumption in a TLS CMP. We show that, perhaps contrary to commonly-held views, TLS need not consume excessive energy. We show that a TLS CMP can be a very 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 TLS CMP microarchitecture and by minimizing wasted TLS work. Then, such a TLS CMP provides a better energy-performance trade-off than a wider-issue superscalar simply because, as the size of the processor structures increases, energy scales superlinearly and performance sublinearly.

This paper offers three contributions. The first one is to identify and quantify the main sources of energy consumption in TLS. These sources are task squashing, hardware structures in the cache hierarchy for data versioning and dependence checking, additional traffic in the memory subsystem due to the same two effects, and additional instructions induced by TLS.

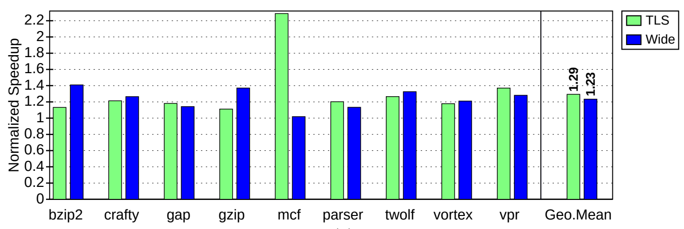
The second contribution is to present and evaluate simple energy-centric optimizations for TLS. They are based on reducing the number of checks, reducing the cost of individual checks, and eliminating work with low performance returns. These optimizations cut *the energy cost of TLS* by over 60% on average. In global terms, they eliminate on average 26.5% of the total on-chip energy in the TLS CMP with minimal performance impact.

The third contribution is to show that a TLS CMP can provide a very desirable energy-performance trade-off, even for SpecInt codes. Specifically, a TLS CMP with four 3-issue cores speeds-up full SpecInt 2000 codes (not just loops) by 1.27 times on average, while keeping the energy cost of TLS to only 25.4%. Such TLS CMP is slightly faster than a 6-issue superscalar at the same frequency, while consuming only 85% of its total on-chip power. We expect better results for floating point, multimedia, or more parallel codes.

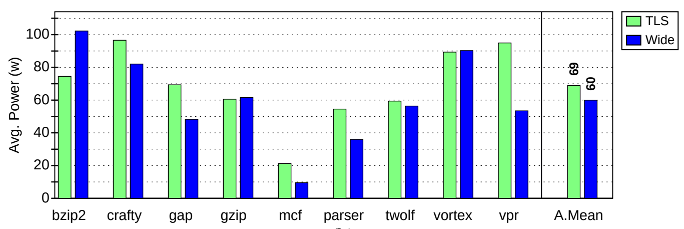
This paper is organized as follows: Section 2 provides a background; Section 3 examines why TLS consumes more energy; Section 4 outlines our TLS architecture and compiler; Section 5 describes simple optimizations to save energy in TLS; Sections 6 and 7 present our methodology and evaluation; and Section 8 lists related work.

2 Background: Thread-Level Speculation (TLS)

Overview. 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



(a)



(b)

Figure 1: Comparing the performance (a) and power (b) of a high-performance TLS CMP with four 3-issue cores (*TLS*) and a 6-issue superscalar (*Wide*) for SpecInt codes. The experiments, which are described in detail later, use a constant frequency. The bars in (a) are normalized to the performance of a 3-issue superscalar.

is repaired, and the tasks are re-executed.

Cross-Task 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. A control dependence violation occurs when a task is spawned in a mispredicted branch path. 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 [6, 8, 22]. 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. An alternative is to move the task's state to some other buffer, but this complicates the design. Instead, it is best that the cache retain the state from the old task and allow the processor to execute another task. If so, the cache has to be multi-versioned.

Lazy commit [17] is an approach where, when a task commits, it does not eagerly merge its cache state with main memory through ownership requests [22] or write backs [10]. Instead, the task simply passes the commit token to its successor. Its state remains in the cache and is lazily merged with main memory later, usually as a result of cache line replacements. This approach improves performance because it speeds up the commit operation. However, it requires multi-versioned caches.

Tagging Multi-Versioned Caches. Multi-versioned caches typically require that we tag each cache line with a version ID, which records what task the line belongs to. Intuitively, such version ID could be the long global ID of the task. However, to save space, it is best to translate global task IDs into some arbitrary Local IDs (LIDs) that are much shorter [22]. These LIDs are used only locally in the cache, to tag cache lines. Their translations into global IDs are kept in a small, per-cache table that we call LID Table. Each cache has a different LID Table.

Architecture and Environment Considered. While TLS can be supported in different ways, we use a CMP because it is a low-complexity, energy-efficient platform. To maximize the use of commodity hardware, the CMP has no special hardware support for inter-processor register communication. Processors can only communicate via the memory system. In addition, to gain usability, the speculative tasks are generated automatically by a TLS compiler. Finally, we concentrate on SpecInt 2000 applications because they are very challenging to speed-up.

3 Sources of Energy Waste in TLS

Enhancing a superscalar into a CMP with TLS support causes the energy consumption to increase. We loosely term the increase as *the energy cost of TLS* (ΔE_{TLS}). In practice, a portion of ΔE_{TLS} simply comes from having multiple cores and caches on chip, and from inefficiencies of parallel execution. However, most of ΔE_{TLS} is due to TLS-specific sources. We are interested in the latter.

We propose to classify TLS-specific sources of energy consumption into four main groups: (1) task squashing, (2) hardware structures in the cache hierarchy needed to support 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 Table 1.

3.1 Task Squashing

An obvious TLS source of energy consumption is the work of tasks that ultimately get squashed. In the TLS CMP that we evaluate in Section 7, 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 or train the branch predictor.

The actual squash operation also consumes energy: a squash signal is sent to the target processor, and some re-initialization code may run there. Such code may involve restoring the register state, but does not require accessing any large chunk of state in the caches. In practice, the frequency of squashes is low, namely 1 squash per 3211 instructions on average in our system. Using our model of Section 6.2, we estimate that each operation takes 320 pJ. Consequently, the total energy consumed by the *actual* squash operations is negligible.

3.2 Hardware Structures for Data Versioning and Dependence Checking

The two main characteristic operations of TLS systems are maintaining data versioning and performing dependence checking. These operations are largely supported in the cache hierarchy. Data versioning is needed when the cache hierarchy can 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

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.

version ID [6, 8, 22] — in our case the LID (Section 2).

To perform dependence checking, caches record the data accessed by each speculative task and how it was accessed. Typically, this is done 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 in 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 instances. First, when a task is squashed, its dirty cache lines are invalidated. Second, in eager-commit systems, when a task commits, its dirty cache lines are merged with main memory through write backs [10] or ownership requests [22]. Finally, in lazy-commit systems, 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 cache lines to memory. Then, that task's LID becomes free and can be re-assigned.

These TLS 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 [17], a FSM periodically selects the LID of a committed task from the LID Table, walks the cache tags writing back to memory the lines of that task, and finally frees up the LID. The FSM operates in the background *eagerly*, using free cache cycles. Other schemes perform similar hardware walks of tags while stalling the processor to avoid causing races. For example, to commit a task in [22], a special hardware module sequentially requests ownership for a group of cache lines whose addresses are stored in a buffer. Since the processor stalls, execution takes longer and, therefore, consumes more energy. Finally, some schemes use "one-shot" hardware signals that change the tag state of a group of lines in a handful of cycles. For example, this is done to invalidate the dirty lines of a squashed task. Such hardware is reasonable when the cache can hold data for only a single or very few speculative tasks [6, 9, 22]. 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 line. A "one-shot" clear of the valid bit of all the lines belonging to a given task would require to keep, for each line tag, 6 NXOR gates that feed into one (possibly cascaded) AND gate. Having such logic per tag entry is likely to slow down the common case of a plain cache access, and result in longer, more energy-consuming executions.

3.3 Additional Traffic for Data Versioning and Dependence Checking

A TLS CMP system generates more traffic than a superscalar. The increase is 460% in our system (Section 7). While some of the increase is the result of parallel execution, there are three main TLS-specific sources of additional traffic.

One reason is that 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, there is less space in the cache for data that may be useful to the task currently running locally. This higher cache pressure increases evictions of useful lines and subsequent misses.

The presence of multiple versions of the same line in the chip also 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 to L2, 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, which creates additional traffic. To see why, recall that these protocols typically track dependences by using the write and exposed-read bits. If this access information is kept per line, tasks that exhibit false sharing may appear to violate data dependences and, as a result, cause squashes [6]. For this reason, many TLS proposals keep some 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.

3.4 Additional Dynamic Instructions Induced by 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: 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 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 where processors communicate only through memory, the compiler must spill registers across task boundaries.

TLS-specific instructions are the other source. They include task spawn and commit instructions. The spawn instruction sends some state from one processor to another. Task commit in lazy implementations sends the commit token between processors [17]. These instructions contribute with 11.7% of the instruction increase.

4 High-Performance TLS Architecture and Compiler

Before we examine ways to reduce TLS energy consumption, it is helpful to outline the high-performance TLS CMP architecture and compiler that we use as baseline in our work. More details can be found in [18], [27], and [19].

4.1 TLS CMP Architecture

The CMP connects four modest-issue processors in a virtual ring. Each processor has a private, multi-versioned L1. The ring is also connected to a small, multi-versioned victim cache. Finally, there is a plain, shared L2 that only holds safe data (Figure 2-(a)). We use a ring interconnect to minimize races in the coherence protocol. The victim cache is included to avoid the more expensive alternative of designing a multi-versioned L2. Figures 2-(b) and (c) show the extensions required by TLS to the processors, L1s, and victim cache. Each structure shows its fields in the form bit_count:field_name.

Each processor has an array of *TaskHolders*, which are hardware structures that hold some state for the tasks that are currently loaded (Figure 2-(b)). Each TaskHolder contains the task's LID, its spawn address (PC), its stack pointer (SP), and a few additional bits that will be discussed later. The register state is kept on the stack.

A copy of the LID for the task currently going through rename is kept in the *CurrentID* register of the load-store queue (Figure 2-(b)). This register is used to tag loads and stores as they are inserted in the load-store queue. With this support, when a load or store is sent to the L1, it includes the task's LID. Note that a processor can have multiple in-flight tasks, although only one being renamed at a time.

In the L1s and the victim cache, each line tag is augmented with an LID and, for each word in the line, with one Write and one Exposed-Read bit (Figure 2-(c)). As per Section 2, each cache keeps its own LID Table to translate LIDs to global task IDs (Figure 2-(c)). The LID Table is direct mapped. Each entry has information for one LID.

A novel feature of this architecture is that each LID Table entry also contains a kill bit and a commit bit for the corresponding task, and a counter of the number of lines in the cache with that LID. These fields are used to speed-up some of the tag-group operations of Section 3.2, as we will see in Section 4.2. Each entry also has a pointer to the corresponding TaskHolder.

The architecture does not include special hardware for register communication between cores. All dependences are enforced through memory. The reason is to minimize changes to off-the-shelf cores.

4.2 Example: Use of the LID Table

To show that our baseline TLS architecture is high performing, we give as an example the novel use of the LID Table for tag-group operations. Each LID Table entry is extended with summary-use information: the number of lines that the corresponding task has in the cache, and whether the task has been killed or committed. With this extra information, all the tag-group operations of Section 3.2 are performed with minimal impact on processor performance.

Specifically, when a task receives a squash or commit signal, its LID Table entry is updated by setting the Killed or the Committed bit, respectively (Figure 3-(a)). No tag walking is performed.

At any time, when a processor issues a load, if the load's address and the LID match one of the L1 tag entries, a hit occurs. In this case, the LID Table is *not* accessed. In all other cases, a miss occurs and the LID Table is accessed. We index the LID Table with the request's LID, obtain the corresponding global task ID, and include it in a request issued to the ring. Moreover, to decide which line to evict from the L1, we also index the LID Table with the LIDs of the lines that are currently using the cache set where space is needed (LID1 and LID2 in Figure 3-(b)). These are not time-critical accesses. For the entries that have the Killed bit set (LID1), the count of cached lines is decremented, and the corresponding line in the cache is either chosen as the replacement victim or invalidated. Also, for the entries with the Committed bit set (LID2), the count is decremented, and the line in the cache is written back to L2 to make room. If any one of these counters reaches zero, that LID is automatically recycled.

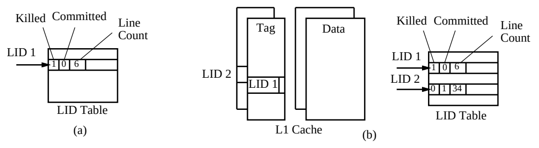


Figure 3: Using the LID Table on a task kill (a) and a cache line replacement (b).

4.3 TLS Compiler

We have built a TLS compiler [27] that adds several passes to a development branch of gcc 3.5. The branch uses a static single-assignment tree as the high-level intermediate representation [7]. With this approach, we leverage a complete compiler infrastructure, including an advanced control flow graph structure. The TLS passes generate tasks out of loop iterations and the code that follows (i.e., the continuation of) subroutines. The compiler first marks the tasks, and then tries to place spawn statements for each of these tasks as early in the code as it can. Only spawns that have moved up the code significantly are retained.

Before running the compiler, we run SGI's source-to-source optimizer (copt from MIPSPro), which performs PRE, loop unrolling, inlining, and other optimizations. As a result, the non-TLS code has a quality comparable to the MIPSPro SGI compiler for integer codes at O3. Code quality when TLS is enabled is not as good, as explained in Section 3.4.

The compilation process includes a simple profiler. The profiler takes the initial TLS executable, runs a tiny data set, and identifies those tasks that should be eliminated because they are unlikely to be of much benefit (Section 5.3.2). Then, the compiler re-generates the executable by eliminating these tasks.

5 Simple Optimizations to Save Energy in TLS

To reduce the energy consumed by the TLS sources of Section 3, we can use many performance-oriented TLS optimizations proposed elsewhere. Examples are improvements to the cache hierarchy to minimize conflicts [6] or enhancements to the coherence protocol to reduce communication latency [23]. While these optimizations improve performance, they typically also reduce the energy consumed by a program.

In this paper, we are not interested in these optimizations. If they are cost-effective, they should already be included in any baseline TLS design. Instead, we are interested in *energy-centric* optimizations. These are optimizations that do not increase performance noticeably; in fact, they may even slightly reduce it. However, they reduce energy consumption significantly. They would not necessarily be included in a performance-centric TLS design.

We propose three guidelines for energy-centric optimizations: (1) reduce the number of checks, (2) reduce the cost of individual checks, and (3) eliminate work with low performance returns. As examples, we propose very simple, yet effective techniques.

5.1 Reducing the Number of Checks

5.1.1 Avoid Eagerly "Walking" the Cache Tags in the Background *(NoWalk)*

With the LID Table design described in Section 4.2, tag-group operations are very fast. A task squash and a commit only involve setting a bit in the LID Table. As lines belonging to squashed or committed tasks are eliminated from the cache due to replacements, the corresponding count in the LID Table is decremented. When a count reaches zero, its LID can be recycled. Consequently, LID recycling is also very fast.

However, waiting for LIDs to get "naturally" recycled in this way may hurt performance. One approach that is used in [17] to recycle task IDs is a hardware FSM that, *eagerly* in the background when the cache is idle, repeatedly walks the tags of the cache identifying lines from a long-committed task. These lines are written back to memory. When all are, the task ID is recycled.

Our baseline TLS architecture uses a similar eager approach in the background. It uses the LID Table to identify killed or committed tasks and then, when the cache is idle, a FSM eagerly walks the cache to eliminate their lines.

The energy optimization that we propose is to avoid any eager walk of the cache tags. Instead, we rely on the "natural" LID recycling. We

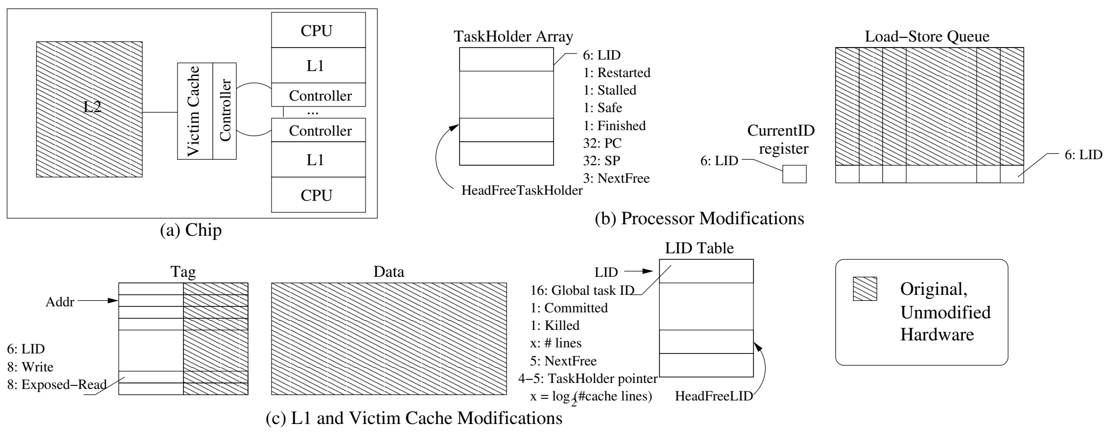


Figure 2: Proposed architecture of a high-performance TLS CMP.

only activate an eager background FSM in one case: to recycle LIDs when there is only one free LID left and, therefore, the cache is about to run out of them. Overall, this optimization eliminates many tag checks.

5.1.2 Reduce Traffic to Check Version-IDs *(TrafRed)*

In TLS protocols, many messages are sent to check version IDs. For example, when a processor writes to a non-exclusive line, all the caches with a version of the requested line are typically checked, to see if there is an exposed read to the line from a more speculative task. Such task will be squashed. Similarly, on displacement of a committed line to L2, those same caches are checked, to invalidate older versions of the line. Such versions cannot remain cached anymore.

We propose a simple optimization to reduce the number of checks needed and, therefore, the traffic. Cache lines are extended 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 of the corresponding address, respectively. As execution proceeds, Newest may be reset on an access by another task. With this support, if a processor writes on a Newest line cached locally, there is no need to check other caches for exposed reads. Similarly, if a processor displaces a committed line with the Oldest bit set, there is no need to check other caches for older versions. This optimization applies to two rows in Table 1.

5.2 Reducing the Cost of Individual Checks

A simple example is to tag cache lines with short LIDs rather than global task IDs. This approach of using indirection is well known [22]. Consequently, we already use it in the baseline TLS CMP and, therefore, we do not evaluate its impact. In Table 1, we call it *Indirect*.

5.3 Eliminating Low-Return Work

5.3.1 Stall a Task After Two Squashes *(StallSq)*

A simple technique is to limit the number of times that a task is allowed to be squashed and restarted. After a task has been squashed *N* times, it is not given a CPU again until it becomes non-speculative.

We performed experiments always restarting tasks immediately after they are 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 and branch predictor have been warmed up. Restarting after further squashes delivers low performance returns while steadily consuming more energy. Consequently, we reset and stall a task after its second squash. This is accomplished with two bits per TaskHolder entry (Figure 2): the Restarted bit is set after the task has been squashed and restarted once; the Stalled bit is set after the second squash.

5.3.2 Energy-Aware Task Pruning *(TaskOpt)*

The profiler in our compilation pass includes a simple model that identifies tasks that should be eliminated because they are unlikely to be beneficial. The main focus is on tasks that cause squashes. For the baseline TLS architecture, the model minimizes the duration of the program. The energy-centric optimization is to use a model that minimizes the product $Energy \times Delay^2$ for the program.

Our compiler generates a binary with task spawn instructions (Figure 4-(a)). 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 simple cache to estimate the number of misses in the machine's L2. For performance, cache timing is not modeled in detail. On average, the profiler takes around 5 minutes to run on a 3 GHz Pentium 4.

The profiler estimates if a task squash will occur and, if so, the number of instructions squashed $I_{squashed}$ (Figure 4-(b)) and the final instruction overlap after re-execution $I_{overlap}$ (Figure 4-(c)). In addition, the profiler estimates the number of L2 misses $M_{squashed}$ in the squashed instructions. These misses will have a prefetching effect that will speed up the re-execution of *T2*.

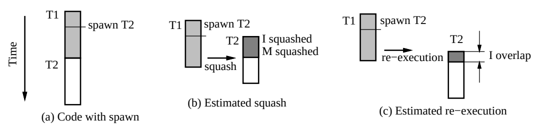


Figure 4: Modeling a task squash and restart. T1 and T2 are tasks.

Assuming that each instruction takes T_i cycles to execute, and an L2 miss stalls the processor for T_m cycles, the estimated execution time reduction (T_{red}) is $I_{overlap} \times T_i + M_{squashed} \times T_m$. Assuming that the energy consumed by each instruction is E_i, the approximate increase in energy (E_{inc}) is $I_{squashed} \times E_i$.

Our profiler focuses on tasks that have a rate of squashes per commit higher than R_{squash}. In the baseline architecture, it eliminates a task if T_{red} is less than a threshold T_{perf}. In our our energy-optimized architecture, it eliminates a task if subtracting T_{red} from the program time and adding E_{inc} to the program energy, the program's $E \times D^2$ product increases. In this case, voltage-frequency scaling could (ideally) do better.

The values of the thresholds and parameters used are listed in Table 2. This optimization has significant impact: on average, the profiler eliminates 39.9% of the static tasks in performance mode, and

49.2% in energy mode.

5.3.3 Eliminate Low-Return Tasks *(TaskOpt)*

Another energy-centric optimization is for the compilation pass to aggressively remove tasks whose size is small or whose spawn point has not been moved up in the code much. We use a threshold size $Size_{energy}$ and threshold spawn hoist distance $Hoist_{energy}$ that are more aggressive than their performance-centric counterparts ($Size_{perf}$ and $Hoist_{perf}$). These optimizations reduce task boundaries and code bloat. They eliminate 36.1% of the static tasks in energy mode compared to 34.7% in performance mode. For ease of presentation, we combine this technique and the previous one into *TaskOpt* in our evaluation, since they are very related.

5.4 Summary

We place the optimizations in the corresponding row of Table 1. As indicated in Section 5.2, *Indirect* is not evaluated.

6 Evaluation Setup

To assess the energy-efficiency of TLS, we compare an TLS CMP to a non-TLS chip that has a single processor of the same or wider issue width. We use execution-driven simulations, with detailed models of out-of-order superscalars and advanced memory hierarchies, enhanced with models of dynamic and leakage energy from Wattch [3], Orion [28], and HotLeakage [30].

6.1 Architectures Evaluated

The TLS CMP that we propose has four 3-issue cores, the microarchitecture of Section 4, and the energy-centric TLS optimizations of Section 5. We call the chip *TLS4-3i*. The non-TLS chips have a single superscalar with a conventional L1 and L2 on-chip cache hierarchy. We consider two: one is a 6-issue superscalar (*Uni-6i*) and the other a 3-issue superscalar (*Uni-3i*). We choose to compare the *TLS4-3i* and *Uni-6i* designs because both chips have approximately the same area, as can be estimated from [11, 20].

Table 2 shows the parameters for *TLS4-3i* and *Uni-6i*. As we move from 3-issue to 6-issue cores, we scale all the processor structures (e.g., ports, FUs, etc) according to the issue width of the core. We try to create a balanced processor as much as possible, by scaling up the processor resources. This is the same approach used in IBM's Power 4.

In our comparison, we favor *Uni-6i*. We assume that *Uni-6i* has the same frequency and the same pipeline depth as the cores in *TLS4-3i*. This helps *Uni-6i* because, in practice, a 6-issue core would not cycle as fast as a 3-issue core with the same pipeline. For example, according to CACTI [20], the access time of the register file and the instruction window in *Uni-6i* would be at least 1.25 times higher and 1.35 times higher, respectively, than in the *TLS4-3i* cores. Moreover, extrapolating results from [15], the bypass network would have 2.6 times longer latency than *Uni-6i* (assuming a fully-connected bypass network). In our simulations, however, we assume the same frequency for *Uni-6i* and *TLS4-3i*.

Since both processors have the same pipeline depth and branch misprediction penalty, we feel that it is fair to also give them the same branch predictor. In addition, both processors have an integer and an FP cluster. Since we run integer codes in the evaluation, the FP cluster is clock-gated almost all the time.

The tag array in *TLS4-3i*'s L1 caches is extended with the LID, and the Write and Exposed-Read bits (Figure 2). At worst, the presence of these bits increases the access time of the L1 only slightly. To see why, note that the LID bits can simply be considered part of the line address tag, as a hit requires address and LID match. Moreover, in our protocol, the Write and Exposed-Read bits are not checked before providing the data to the processor; they may be updated after that. However, to be conservative, we increase the L1 access latency in *TLS4-3i* one cycle over *Uni-6i*, to *3 cycles*.

Uni-3i is like *Uni-6i* except that the core is 3-issue, like those in *TLS4-3i*, and the L1 cache only has 1 port. For completeness, we also evaluate one additional chip: *TLS2-3i*. *TLS2-3i* is a TLS CMP like *TLS4-3i*, but with only two cores.

While TLS can be supported in different ways, we use a CMP because it is a low-complexity, energy-efficient platform. To maximize the use of commodity hardware, the CMP has no special hardware support for inter-processor register communication. Processors can only communicate via the memory system. [19, 18] have more details about the architecture evaluated.

6.2 Energy Considerations

We estimate and aggregate the dynamic and leakage energy consumed in all chip structures, including processors, cache hierarchies, and on-chip interconnect. For the dynamic energy, we use the Wattch [3] and Orion [28] models. We apply aggressive clock gating to processor structures in all cores. In addition, unused cores in the TLS CMP are also clock gated. Activating and deactivating core-wide clock gating takes 100 cycles each. Clock-gated structures are set to consume 5% of their original dynamic energy, which is one of the options in Wattch. We extend the Wattch models to support our deeper pipelines and to take into account the area when computing the clock energy. The chip area is estimated using data from [11] and CACTI [20].

Leakage energy is estimated with HotLeakage [30], which models both sub-threshold and gated leakage currents. We use an iterative approach suggested by Su *et al.* [24]: the temperature is estimated based on the current total power, the leakage power is estimated based on the current temperature, and the leakage power is added to the total power. This is continued until convergence. The maximum temperature at the junction for any application is not allowed to go beyond 85°C, as recommended by the SIA Roadmap [1].

From our calculations, the average power consumed by the *Uni-3i* and *Uni-6i* chips for the SpecInt 2000 applications is 32 and 60 W, respectively (more data will be shown later). Of this power, leakage accounts for 38% and 32%, respectively. The majority of the power increase from *Uni-3i* to *Uni-6i* is due to five structures that more than double their dynamic contribution, largely because they double the number of ports. These are the rename table, register file, I-window, L1 data cache, and data TLB. In addition, the data forwarding network also increases its dynamic contribution by 70%. We base our confidence in the accuracy of these numbers on the fact that Wattch and HotLeakage have been validated for similar superscalars [3, 30]. The additional structures added by the TLS CMP are largely regular SRAM structures, which can be modeled by CACTI, Wattch, and HotLeakage.

6.3 Applications Evaluated

We measure *full* SpecInt 2000 applications with the *Reference* data set except *eon*, which is in C++, and *gcc* and *perlbmk*, which our compiler infrastructure does not compile. By *full applications*, we mean that we include all the code in the measurement, not just the more parallel sections such as loops. *Uni-3i* and *Uni-6i* run the binaries compiled with our TLS passes disabled. Such binaries have a code quality comparable to integer codes generated by the MIPSPro SGI compiler with *O3* (Section 4.3).

TLS and non-TLS binaries are very different. Therefore, we cannot compare the execution of a fixed number of instructions. Instead, we insert "simulation markers" in the code and simulate for a given number of markers. After skipping the initialization (several billion instructions), we execute up to a certain number of markers so that *Uni-6i* graduates from 750 million to 1.5 billion instructions.

7 Evaluation

In our evaluation, we first characterize the TLS CMP architecturally, with and without the energy optimizations. Then, we examine the

TLS CMP with four 3-issue cores (*TLS4-3i*)	
Processor	
Frequency: **5.0 GHz** @ 70 nm	Fetch/issue/comm width: 6/**3**/3
Branch penalty: 13 cyc (min)	I-window/ROB size: 68/126
RAS: 32 entries	Int/FP registers: 90/68
BTB: 2K entries, 2-way assoc.	LdSt/Int/FP units: 1/2/1
Branch predictor (spec. update):	Ld/St queue entries: 48/42
bimodal size: 16K entries	TaskHolders/processor: 8
gshare-11 size: 16K entries	TaskHolder access time, energy: 1 cyc, 0.25nJ

Cache	D-L1	VC	L2		D-L1	VC
Size:	16KB	4KB	1MB	LID Table:		
RT:	**3 cyc**	8 cyc	10 cyc	entries/ports:	64/2	32/1
Assoc:	4-way	4-way	8-way	acc time/energy:	1cyc/0.11nJ	1cyc/0.07nJ
Line size:	64B	64B	64B			
Ports:	1	1	1	Latency from spawn to new thread: 14 cyc		
Pend ld/st:	16	64	64			

6-issue superscalar chip (*Uni-6i*)	
Processor	
Frequency: **5.0 GHz** @ 70 nm	Fetch/issue/comm width: 6/**6**/6
Branch penalty: 13 cyc (min)	I-window/ROB size: 104/204
RAS: 32 entries	Int/FP registers: 132/104
BTB: 2K entries, 2-way assoc.	LdSt/Int/FP units: 2/4/2
Branch predictor (spec. update):	Ld/St queue entries: 66/54
bimodal size: 16K entries	
gshare-11 size: 16K entries	

Cache	D-L1	L2
Size:	16KB	1MB
RT:	**2 cyc**	10 cyc
Assoc:	4-way	8-way
Line size:	64B	64B
Ports:	2	1
Pend ld/st:	16	64

I-L1	Size: 16KB; RT: 2 cyc; assoc: 2-way; line size: 64B; ports: 1
Bus & Memory	FSB frequency: 533MHz; FSB width: 128bit; memory: DDR-2; DRAM bandwidth: 8.528GB/s; memory RT: 98ns
Profiling parameters	R_{squash}: 0.8; T_i: 1; T_m: 200 cyc; E_i: 8pJ; T_{perf}: 90 cyc; $Size_{energy}$: 45; $Size_{perf}$: 30; $Hoist_{energy}$: 120; $Hoist_{perf}$: 110

Table 2: TLS CMP with four 3-issue cores (*TLS4-3i*) and 6-issue superscalar chip (*Uni-6i*) modeled. In the table, RAS, FSB, RT, and VC stand for Return Address Stack, Front-Side Bus, minimum Round-Trip time from the processor, and Victim Cache, respectively. Cycle counts refer to processor cycles.

energy cost of TLS and the savings of the optimizations. Finally, we compare the energy, power, and performance of the different chips. In the following, *NoOpt* is *TLS4-3i* without the optimizations.

7.1 Architectural Characterization of the TLS CMP

We measure architectural characteristics of the 4-core TLS CMP that are related to Table 1's sources of TLS energy consumption and optimizations. The data are shown in Table 3. In the table, we compare the chip before optimization (*NoOpt*), to the chip with one optimization at a time (*StallSq*, *TaskOpt*, *NoWalk*, or *TrafRed*).

The first TLS source of energy consumption in Table 1 is task squashing. Column 2 of Table 3 shows that, on average, *NoOpt* loses to task squashes 22.6% of the dynamic instructions executed. This is a significant waste. With our optimizations, we reduce the number of such instructions. Specifically, the average fraction becomes 20.7% with *StallSq* (Column 3) and 17.6% with *TaskOpt* (Column 4). Although not shown in the table, the fraction becomes 16.9% with both optimizations combined.

The next few columns of Table 3 provide more information on the impact of *StallSq* and *TaskOpt*. Under *NoOpt*, the average number of busy CPUs is 2.00 (Column 5). Since *StallSq* stalls tasks that are likely to be squashed and *TaskOpt* removes them, they both reduce CPU utilization. Specifically, the average number of busy CPUs is 1.95 and 1.78 with *StallSq* and *TaskOpt*, respectively (Columns 6 and 7). With both optimizations, the average can be shown to be 1.75.

TaskOpt has a significant impact on the tasks. Recall from Sections 5.3.2 and 5.3.3 that, on average, *NoOpt* already prunes 74.6% of the static tasks using performance-only models. On top of that, *TaskOpt* prunes an additional 10.7% of the static tasks (Column 8). As a result, *TaskOpt* increases the average task size from 544 instructions in *NoOpt* (Column 9) to 635 (Column 10). Moreover, the average $E \times D^2$ product of the applications, a metric for time and energy efficiency of computation [14], decreases by 6.5% (Column 11).

The second TLS source of energy in Table 1 is dominated by accesses to L1 cache tags. Such accesses in TLS are both more expensive (since tags have version IDs) and more frequent (e.g., due to tag-group operations). Column 12 of Table 3 shows that, on average, *NoOpt* has 3.3 times the number of tag checks in *Uni-3i*. However, with our *NoWalk* optimization, we eliminate many of these checks. Specifically, Column 13 shows that, with *NoWalk*, TLS only has 2.2 times as many tag checks as *Uni-3i*. Note that these figures include the contribution of squashed tasks.

The third TLS source of energy is additional traffic. Column 14 of Table 3 shows that, on average, *NoOpt* has 19.6 times the traffic of *Uni-3i*. To compute the traffic, we add up all the bytes of data or control passed between caches. This traffic increase is caused by the factors described in Section 3.3. However, after we apply our *TrafRed* optimization, the traffic reduces considerably. On average, with *TrafRed*, TLS only has 5.6 times the traffic of *Uni-3i* (Column 15).

The fourth TLS source of energy is additional instructions. Column 16 shows that *NoOpt* executes on average 12.5% more dynamic instructions in non-squashed tasks than *Uni-3i*. The *TaskOpt* optimization, by eliminating small and inefficient tasks, reduces the additional instructions to 11.9% on average (Column 17).

7.2 The Energy Cost of TLS (ΔE_{TLS})

In Section 3, we defined the energy cost of TLS (ΔE_{TLS}) as the difference between the energy consumed by our TLS CMPs and *Uni-3i*. Figure 5 characterizes ΔE_{TLS} for our 4-core TLS CMP. The figure shows six bars for each application. They correspond to the total energy consumed by the chip without any optimization (*NoOpt*), with individual optimizations enabled (*StallSq*, *TaskOpt*, *NoWalk*, and *TrafRed*), and with all 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 ΔE_{TLS}.

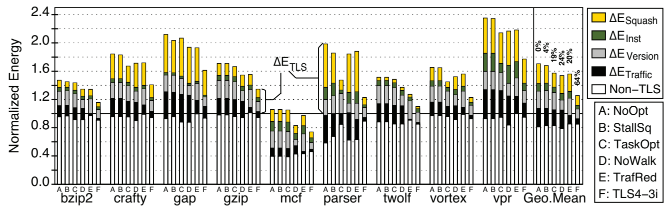


Figure 5: Energy cost of TLS (ΔE_{TLS}) for our 4-core TLS CMP chip with and without energy-centric optimizations. The percentages listed above the average bars are the decrease in ΔE_{TLS} when the optimizations are enabled.

Each bar in Figure 5 is broken into the contributions of the TLS-specific sources of energy consumption listed in Table 1. These include task squashing (ΔE_{Squash}), additional dynamic instructions in non-squashed tasks (ΔE_{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, ΔE_{TLS} should be roughly equal to the addition of the four TLS-specific sources of energy consumption and, therefore, *Non-TLS*

Apps	Squashed Instructions (%)			Busy CPUs			Pruned Tasks (%)	Task Size (Instructions)		ED^2 Reduc. (%)	Ratio of Tag Accesses (TLS/*Uni-3i*)		Traffic (TLS/*Uni-3i*)		Add'l Instruct. in Non-Squashed Dyn. Tasks (%)	
	No Opt	Stall Sq	Task Opt	No Opt	Stall Sq	Task Opt	Task Opt	No Opt	Task Opt	Task Opt	No Opt	No Walk	No Opt	Traf Red	No Opt	Stall Sq
bzip2	9.9	7.5	9.9	1.40	1.35	1.41	21.8	743.4	751.7	-0.3	3.2	1.3	14.0	2.5	5.6	5.6
crafty	26.2	25.4	18.9	1.97	1.95	1.70	6.2	932.0	1064.0	6.8	2.9	2.0	7.1	3.6	5.6	5.6
gap	35.2	31.6	35.1	2.07	1.94	2.06	14.6	1270.3	1280.4	-0.8	3.6	2.2	16.7	8.4	3.8	3.8
gzip	14.0	14.0	11.9	1.49	1.48	1.49	7.5	626.6	634.2	0.3	3.5	1.9	12.3	4.0	6.5	6.5
mcf	28.8	28.7	28.8	2.38	2.38	2.38	0.4	47.9	47.9	0.0	3.8	2.7	42.0	11.5	31.9	31.9
parser	39.3	29.9	13.8	2.03	1.85	1.25	18.9	167.3	261.6	26.4	3.6	3.2	9.8	7.1	20.8	18.0
twolf	4.4	4.4	4.4	1.62	1.62	1.62	0.4	409.4	409.4	0.0	3.3	1.6	55.9	3.2	6.5	6.5
vortex	15.7	15.4	7.7	1.82	1.81	1.49	9.2	488.3	881.5	16.0	2.9	1.9	7.4	3.9	7.5	7.4
vpr	29.5	29.2	27.9	3.14	3.13	2.61	17.0	212.9	389.1	10.4	3.2	3.1	10.9	6.4	23.9	21.2
Avg	22.6	20.7	17.6	2.00	1.95	1.78	10.7	544.2	635.5	6.5	3.3	2.2	19.6	5.6	12.5	11.9

Table 3: Architectural characteristics of the 4-core TLS CMP related to TLS sources of energy consumption and their optimization.

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. In fact, since for all applications the TLS CMP is faster than *Uni-3i* (i.e., the TLS bars in Figure 1-(a) are over 1), *Non-TLS* is less than 1: non-TLS hardware structures have less time to leak or to spend dynamic energy cycling idly.

If we consider the *NoOpt* bars, we see that the energy cost of *unoptimized* TLS (ΔE_{TLS}) is significant. On average, unoptimized TLS adds 70.4% to the energy consumed by *Uni-3i*. We also see that all four of our TLS sources of energy consumption contribute noticeably. Of them, task squashing consumes the most energy, while additional instructions consumes the least.

7.3 The Impact of Energy-Centric Optimizations

The rest of the bars in Figure 5 show the impact of our energy-centric optimizations on the TLS energy sources. From the figure, we see that 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 5, *TaskOpt* reduces ΔE_{Squash} and ΔE_{Inst} — its targets in Table 1. This is consistent with Table 3, where *TaskOpt* 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%.

Consider now *NoWalk*. In Figure 5, *NoWalk* mostly reduces $\Delta E_{Version}$ — its target in Table 1. This was expected from Table 3, where *NoWalk* reduces the number of tag accesses relative to *Uni-3i* from 3.3 times to 2.2 times. In addition, since it reduces the temperature, it also reduces the leakage component in *Non-TLS* slightly.

If we consider *TrafRed* in Figure 5, we see that it mostly reduces $\Delta E_{Traffic}$ — its target in Table 1. Again, this is consistent with Table 3, where *TrafRed* reduces the traffic relative to *Uni-3i* from 19.6 times to 5.6 times on average.

Finally, *StallSq* only addresses ΔE_{Squash}, which is its target in Table 1. As expected from the modest numbers in Table 3, where it reduces squashed instructions from 22.6% to 20.7%, it has a small impact in Figure 5.

This analysis shows that each of *TaskOpt*, *NoWalk*, and *TrafRed* effectively reduces a different energy source, and that the three techniques combined cover all sources considered. Consequently, when we combine all four optimizations in *TLS4-3i*, all TLS sources of consumption decrease substantially. The resulting *TLS4-3i* bar shows the *true energy cost* of TLS. If we measure the section of the bar over 1.00, we see that this cost is on average only 28%. We feel that this is a remarkably low energy overhead for TLS.

With our very simple optimizations, we have been able to eliminate on average 64% of ΔE_{TLS}. Compared to the overall on-chip energy consumed by *NoOpt*, this is a very respectable energy reduction of 26.5%. Moreover, as we will see later, the applications have only been slowed down on average by less than 2%.

Finally, an analysis of individual applications reveals many interesting facts. Unfortunately, space limitations prevent any meaningful discussion. We only note that *mcf* has a negative ΔE_{TLS} in some cases. The reason is that, without TLS, the L2 suffers frequent misses; with TLS, threads prefetch data for other threads, removing misses and speeding up the execution significantly (Section 7.5). The result is that the TLS CMP has less time to leak and to spend dynamic energy cycling, hence *Non-TLS* is tiny.

7.4 Comparing Energy Consumption across Chips

Figure 6 compares the energy consumed by our optimized *TLS4-3i* chip and *Uni-3i*, *Uni-6i* and, for completeness, *TLS2-3i*. Each bar is normalized to *Uni-3i* and broken down into dynamic energy consumed by the clock, core, and memory subsystem, and leakage energy. The memory category includes caches, TLBs, and interconnect.

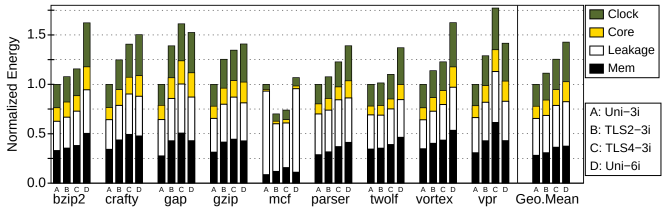


Figure 6: Comparing the energy consumption after TLS CMP optimization. All bars are normalized to *Uni-3i*.

Consider first *Uni-6i*. Its core, clock, and memory categories are larger than in *Uni-3i* because of the bigger structures in the wide processor. Specifically, the rename table, register file, I-window, L1 data cache, and data TLB have twice the number of ports. This roughly doubles the energy per access [20]. Furthermore, all these structures but the cache and TLB also have more entries. Finally, the forwarding network also increases its complexity and, therefore, its consumption. The figure also shows that leakage has increased. The reason is that, while *Uni-6i* is faster than *Uni-3i*, it consumes the highest average power (Section 7.5) and, therefore, has a higher temperature. Temperature has an exponential impact on leakage.

Compared to *Uni-6i*, *TLS4-3i* has smaller core and clock energies because it has the simpler hardware structures of *Uni-3i*. Its leakage is also smaller because its average power (Section 7.5) and, therefore, temperature are smaller than *Uni-6i*. Its memory category, however, is slightly higher than *Uni-6i*. The reason is *TLS4-3i*'s higher consumption in structures and traffic to support data versioning and dependence checking.

7.5 Comparing Performance and Power Across Chips

Finally, we take the optimized *TLS4-3i* and compare its performance, and average power to other chips. Figure 7-(a) shows application speedups relative to execution on *Uni-3i*, while Figure 7-(b) shows the average power consumed during execution. As usual, *TLS2-3i* is also shown. As a reference, the arithmetic mean of the average IPC of the applications on *TLS4-3i* is 1.38.

Figure 7-(a) shows that, on average, *TLS4-3i* delivers a speedup of 1.27 over *Uni-3i*. This shows that our TLS compiler successfully extracts good tasks from these irregular codes. This speedup is slightly lower than the 1.29 speedup shown in Figure 1-(a). The reason is that our energy-centric optimizations reduce performance slightly.

Figure 7-(a) 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 we conservatively assume the same frequency for both chips. In practice, designing the wider issue processor at this high frequency is likely to be more challenging.

Note that while the *TLS4-3i* speedup for most codes ranges from 1.10 to 1.35, *mcf* exhibits a higher speedup. As indicated in Section 7.3, *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 even without giving frequency advantage to *TLS4-3i*.

On the other hand, Figure 7-(b) shows that the average 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.

We also compare the average $E \times D^2$ product of *TLS4-3i* and *Uni-6i*. Unfortunately, due to lack of space, we cannot show the complete set of data. On average, *TLS4-3i*'s $E \times D^2$ product is 7.6% lower than *Uni-6i*'s. We conclude, therefore, that *TLS4-3i* is more energy-efficient than *Uni-6i*.

We can get further insight if we analytically apply *ideal* voltage-frequency scaling. We assume that performance is linearly proportional to frequency and scale frequency and voltage proportionally. We also assume that average dynamic power is proportional to the cube of frequency and that average leakage power is linearly proportional to voltage [4]. Then, for each chip, we can derive a curve that relates the average power consumption with performance as:

$$P_{new}^{total} = P_{orig}^{dyn} \times \left(\frac{Speedup_{new}}{Speedup_{orig}}\right)^3 + P_{orig}^{leak} \times \left(\frac{Speedup_{new}}{Speedup_{orig}}\right)$$

Figure 8 shows the resulting curves for *TLS4-3i* and *Uni-6i*. Each curve follows possible speedup-power working points for one chip. The lower a curve is, the more energy-efficient the architecture is. Each curve shows one data point, which corresponds to the actual working conditions in our experiments.

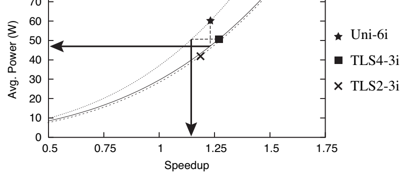


Figure 8: *Ideal* relation between speedup and average power.

We can see that *Uni-6i* is less energy-efficient than *TLS4-3i*. If we scale down *TLS4-3i*'s frequency until *TLS4-3i*'s performance is equal to *Uni-6i*'s, *TLS4-3i* consumes 20% less power than *Uni-6i* (horizontal arrow). Alternatively, if we scale down *Uni-6i*'s frequency until *Uni-6i*'s power is equal to *TLS4-3i*'s, *TLS4-3i* is 13% faster than *Uni-6i* (vertical arrow).

Finally, Figure 8 also shows a curve for *TLS2-3i*. The data shows that *TLS2-3i* has a small efficiency advantage over *TLS4-3i*.

7.6 Summary

The fundamental reason why TLS-CMPs can be more energy-efficient than wider-issue superscalars is that energy scales superlinearly and performance sublinearly with the size of processor structures. Consequently, multiple simple TLS cores can be more efficient than a single wide core, as long as (i) TLS's hardware overheads and (ii) TLS's wasted work are kept to a minimum.

We have addressed these issues with a lean TLS CMP microarchitecture and a set of energy-centric optimizations. The efficient operation of the final *TLS4-3i* and *TLS2-3i* designs is shown in Table 4. Specifically, on average 1.40 and 1.75 cores in *TLS2-3i* and *TLS4-3i*, respectively, are busy (Columns 2 and 3). Moreover, while busy, these cores execute instructions from squashed tasks for only 10.3% and 16.9% of the cycles, respectively (Columns 4 and 5). This is in contrast to *TLS4-3i*, *NoOpt*: on average, 2.00 cores are busy (Column 5 of Table 3), and they execute instructions from squashed tasks for 22.6% of the time (Column 2 of Table 3).

Apps	Busy CPUs		Squashed Instr. (%)	
	TLS2-3i	*TLS4-3i*	*TLS2-3i*	*TLS4-3i*
bzip2	1.17	1.36	4.6	7.5
crafty	1.46	1.68	17.0	18.2
gap	1.56	1.93	21.7	31.4
gzip	1.40	1.48	14.5	13.2
mcf	1.68	2.38	7.1	28.7
parser	1.10	1.25	6.4	13.9
twolf	1.29	1.62	1.7	4.4
vortex	1.31	1.49	6.5	7.8
vpr	1.58	2.58	12.9	27.2
Avg	1.40	1.75	10.3	16.9

Table 4: Characterizing the optimized *TLS4-3i* and *TLS2-3i* chips.

8 Related Work

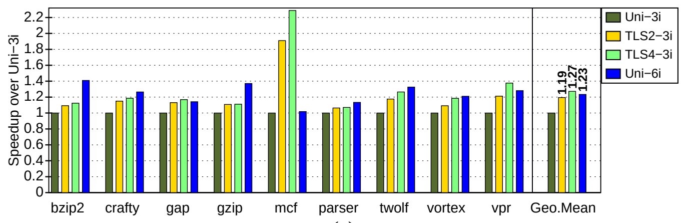
Past work on TLS CMP architectures with TLS support has focused on performance rather than energy (e.g., [5, 8, 9, 10, 13, 21, 22, 25, 26]). There has been work on reducing the energy consumed in the pipeline due to instruction-level speculation following a branch prediction [2, 12]. However, the issues addressed are very different.

Concurrently to our work, Petric and Roth [16] developed an infrastructure for selecting pre-execution (prefetching) threads in an SMT processor. To select threads, they use models that minimize execution time, energy consumption, or $E \times D^2$ product. While their models are somewhat related to those used in our *(TaskOpt)* optimization, they are fundamentally different. Most importantly, our models are focused on trading off performance and energy in the event of a *task squash*. Such an event does not exist in their models. Unlike our tasks, their threads never get squashed, do not offload computation, are only spawned from the main thread, and are used in an SMT processor. Therefore, the models are different.

9 Conclusions

This paper challenges the commonly-held view that TLS consumes excessive energy and power. Its thesis is based on three contributions. The first one is identifying the main sources of energy consumption in TLS: task squashing, structures for data versioning and dependence checking, additional traffic due to these two effects, and additional instructions. The second contribution is proposing very simple energy-centric optimizations to mitigate them. These optimizations cut the energy cost of TLS by over 60% on average. In global terms, they eliminate on average 26.5% of the total on-chip energy in the TLS CMP with minimal performance impact.

The third contribution is showing that a TLS CMP can offer a very desirable energy-performance trade-off, even for SpecInt codes. An optimized TLS CMP with 4 3-issue cores delivers an average speedup


(a)

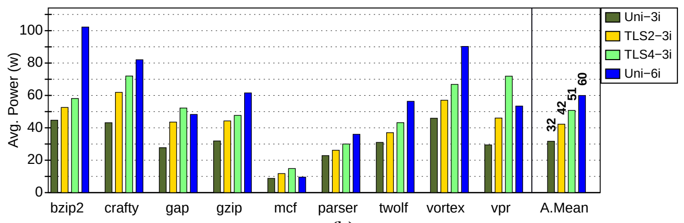

(b)

Figure 7: 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.

of 1.27 over a 3-issue superscalar on full SpecInt 2000 codes (not just loops), while consuming only 25.4% more energy. Moreover, the TLS CMP is slightly faster than a 6-issue superscalar for the same frequency, while consuming only 85% of its total on-chip power and yielding a 7.6% lower $E \times D^2$ product for these challenging applications.

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 TLS CMPs can also speed up these most challenging SpecInt codes, both better and with less energy than wider superscalars. We expect better results for more parallel codes.

REFERENCES

[1] *International Technology Roadmap for Semiconductors*. Semiconductor Industry Association, 2002.

[2] J. L. Aragon, J. Gonzalez, and A. Gonzalez. Power-Aware Control Speculation Through Selective Throttling. In *Proceedings of the 9th High-Performance Computer Architecture Conference*, pages 103–112, February 2003.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, 2000.

[5] M Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[6] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[7] SSA for trees - GNU project. URL, May 2003. ”http://www.gccsummit.org/2003/view_abstract.php?talk=2”.

[8] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[9] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[10] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, December 2003.

[12] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[13] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 365–372, June 1999.

[14] A. J. Martin, M. Nystroem, and P. Penzes. ET2: A metric for time and energy efficiency of computation. Technical Report CSTR:2001.007, Caltech, Dec. 2001.

[15] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA’97)*, June 1997.

[16] V. Petric and A. Roth. Energy-effectiveness of pre-execution and energy-aware p-thread selection. Technical Report MS-CIS-03-34, University of Pennsylvania, Nov. 2003.

[17] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA’01)*, pages 204–215, June 2001.

[18] J. Renau. *Chip Multiprocessors with Speculative Multithreading: Design for Performance and Energy Efficiency*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[19] 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 *Proceedings of the 19th ACM International Conference on Supercomputing*, June 2005.

[20] 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.

[21] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[22] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[23] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the 8th High-Performance Computer Architecture Conference*, February 2002.

[24] Haihua Su, Frank Liu, Anirudh Devgan, Emrah Acar, and Sani Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, August 2003.

[25] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[26] 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.

[27] J. Tuck. A novel compiler framework for a chip-multiprocessor architecture with thread-level speculation. Master’s thesis, University of Illinois at Urbana-Champaign, 2004.

[28] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002.

[29] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[30] 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, Univ. of Virginia Dept. of Computer Science, March 2003.