Energy-Efficient Thread-Level Speculation

CHIP MULTIPROCESSORS WITH THREAD-LEVEL SPECULATION HAVE BECOME THE SUBJECT OF INTENSE RESEARCH. THIS WORK REFUTES THE CLAIM THAT SUCH A DESIGN IS NECESSARILY TOO ENERGY INEFFICIENT. IN ADDITION, IT PROPOSES OUT-OF-ORDER TASK SPAWNING TO EXPLOIT MORE SOURCES OF SPECULATIVE TASK-LEVEL PARALLELISM.

Jose Renau
University of California at Santa Cruz

Karin Strauss
Luis Ceze
Wei Liu
Smruti R. Sarangi
James Tuck
Josep Torrellas
University of Illinois at Urbana-Champaign

To speed up nonnumerical applications that are hard to parallelize, designers build sophisticated out-of-order processors with carefully tuned execution engines and memory subsystems. Unfortunately, these systems tend to have highly complex designs and yield diminishing performance returns—motivating the search for design alternatives. One such alternative is thread-level speculation (TLS) on a chip multiprocessor (CMP). CMPs are attractive because they are more energy efficient, more scalable, and less complex than wide-issue superscalar processors. With TLS, they can also execute in parallel challenging sequential codes, such as SPECint. TLS partitions hard-to-analyze applications into tasks that the processors optimistically execute in parallel, hoping to avoid any cross-task dependence violation. Special hardware support monitors the tasks' data accesses and detects runtime violations. If such a violation occurs, the hardware transparently rolls back the incorrect tasks and, after repairing the state, restarts them. The "Principles of Thread-Level Speculation" sidebar describes TLS foundations in more detail.

Although a TLS CMP offers major benefits, many contend that its energy efficiency is too low to seriously challenge conventional processors. The rationale is that aggressive speculative execution is not the best course when energy and power consumption are a processor's primary constraints.

We argue otherwise, and have identified simple energy-saving optimizations that make a TLS CMP an attractive option for high-performance, power-constrained processor design, even running SPECint codes. Our TLS CMP design relies on an efficient microarchitecture with out-of-order task spawning and a novel TLS compiler. When we evaluated this design, we found that TLS's energy consumption remained modest and that our TLS CMP provided a better energy-performance trade-off than a wider issue superscalar processor.[6,9]

Reducing energy consumption

Enhancing an *N*-issue superscalar to make it a CMP with several *N*-issue cores and TLS support naturally increases energy consumption. Part of this increase comes from the inefficiencies of parallel execution, and from having multiple on-chip cores and caches, but most of it is due directly to TLS. As Table 1 shows, this increase, ΔE_{TLS}, stems from four main sources:

- task squashing,
- hardware structures in the cache hierar-

Published by the IEEE Computer Society

Principles of Thread-Level Speculation

Because sequential code imposes a task order, TLS looks at tasks as predecessors and successors, and the safe (or nonspeculative) task precedes all speculative tasks. As tasks execute, special hardware support checks that no cross-task dependence is violated. If violations occur, the special hardware squashes incorrect tasks, repairs any polluted state, and reexecutes the tasks.

Cross-task data dependence violations

The cache controller typically monitors data dependences by tracking, for each 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 a successor task has read with an exposed read. Dependence violations lead to task squashes, which involve discarding the work the task produced.

State buffering

Stores from a speculative task generate speculative state that cannot merge with the program's safe state because it could be incorrect. Typically, the cache of the processor running the task stores such state. If the cache controller detects a violation, the cache discards the state. Otherwise, when the task becomes nonspeculative, the cache controller lets the state propagate to memory. When a nonspeculative task finishes execution, it commits. Committing informs the rest of the system that the state the task generated 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 can write to the same variable and thus produce different versions of the variable. The cache must buffer such versions separately and provide readers with the correct versions. Finally, as tasks commit in order, they must be able to merge data versions with the safe memory state, also in order.

Multiversioned caches

A cache that can hold state from multiple tasks is called multiversioned.[1-3] Such caches enhance performance because they avoid processor stall when tasks are imbalanced, and they enable lazy commit.

If tasks have load imbalance, a processor may finish a task and the task still be speculative. If the cache can hold state only for a single speculative task, the processor must stall until the task becomes safe.[4] An alternative is to move the task's state to some other buffer, but this complicates the design. Instead, the cache can retain the old task's state and let the processor execute another speculative task. Thus, the cache must be multiversioned.

In lazy commit,[5] when a task commits, it does not eagerly merge its cache state with main memory through ownership requests[3] or write backs.[6] Instead, the task simply passes the commit token to its successor. Its state remains in the cache and lazily merges with main memory later, usually because of cache line replacements. This approach improves performance because it speeds up the commit operation, but it requires multiversioned caches.

Multiversioned caches, in turn, require tagging each cache line with a version ID, which records what task the line belongs to. The version ID could be the long global task ID, but to save space, it is best to translate global task IDs into some arbitrary local IDs (LIDs) that are much shorter.[3] The LIDs are used only locally in the cache, to tag cache lines. Their translations into global IDs are kept in a small, per-cache LID table. Each cache has a different LID table.

Architecture and environment

Although support for TLS could take many forms,[3,6-11] we use a CMP because it is a low-complexity, energy-efficient platform. To maximize the

continued on p. 4

Table 1. Sources of energy consumption from TLS and energy-saving optimizations

Energy source	Reasons for consumption	Optimization
Task squashing	Work of the tasks that get squashed	StallSq, TaskOpt
	Tasks squash operations	
Hardware structures in the cache hierarchy for data versioning and dependence checking	Storage and 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	None
	Selection and combination of multiple versions	TrafRed
	Fine-grained data dependence tracking	
Additional dynamic instructions induced by TLS	Side effects of breaking the code in tasks	TaskOpt
	TLS-specific instructions	

*Already in the TLS baseline system.

continued from p. 3
use of commodity hardware, our CMP has no special hardware support for interprocessor register communication. Processors communicate only through the memory system.

References

1. M. Cintra, J.F. Martínez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 13-24.
2. S. Gopal et al., "Speculative Versioning Cache," *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 98), IEEE CS Press, 1998, pp. 195-205.
3. J. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 1-12.
4. M.J. Garzarán et al., "Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 191-202.
5. M. Prvulovic et al., "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 204-215.
6. V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers*, no. 48, vol. 9, Sept. 1999, pp. 866-880.
7. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VIII), ACM Press, 1998, pp. 58-69.
8. P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," *Proc. Int'l Conf. Supercomputing* (SC 99), IEEE CS Press, 1999, pp. 365-372.
9. G.S. Sohi, S.E. Breach, and T.N. Vijayakumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture* (ISCA 95), IEEE CS Press, 1995, pp. 414-425.
10. M.Tremblay, J.Chan, S.Chaudhry, A.R. Conigliaro, and S.S. Tse, "The MAJC Architecture: A Synthesis of Parallelism and Scalability" IEEE Micro, vol 20, No.6 Nov/Dec 2000, p12-25.
11. J. Tsai et al., "The Superthreaded Processor Architecture," *IEEE Trans. Computers*, vol. 48, no. 9, Sept. 1999, pp. 881-902.

chy needed for data versioning and dependence checking,
- additional traffic in the memory system from the previous two effects, and
- additional dynamic instructions induced by TLS.

The optimizations we developed reduce consumption from all four sources. Our focus is on energy-centric optimizations, not on optimizations that reduce energy by first improving performance—we assume that these would already be present in a baseline TLS design. Although our optimizations do not noticeably increase performance (and can even slightly reduce it), they significantly reduce energy consumption.

Our optimizations are based on three guidelines: reduce the number of checks, reduce the cost of individual checks, and eliminate work with low performance returns.

Task squashing

A TLS source of energy consumption is the work of tasks that ultimately get squashed. In the TLS CMP we evaluated, 22.6 percent of all graduated instructions belong to such tasks. Not all such work is wasted, however, since a squashed task can bring useful data into the caches.

The actual squash operation also consumes energy. However, in our system, the frequency of squash operations is only 1 per 3,211 instructions on average. Consequently, the total energy that the actual squash operations consume is negligible.

As Table 1 shows, the optimizations that address task squashing are *StallSq* (task stalling) and *TaskOpt* (eliminating energy-inefficient tasks). The aim of StallSq is to decrease the number of instructions in squashed tasks. We do this by limiting the times that a task is permitted to restart after a squash. With StallSq, when a task has been squashed N times, the hardware does not give the task a CPU again until it becomes nonspeculative. To select N, we performed experiments while always restarting tasks after they were squashed. We found that 73.0 percent of the dynamic tasks are never squashed; 20.6 percent are squashed once; 4.1 percent, twice; 1.4 percent, three times; and 0.9 percent, four times or more. Restarting a task after its first squash can be beneficial, since the cache has warmed up. Restarting after further squashes delivers diminishing returns. Consequently, we reset and stall a task after its second squash.

The aim of TaskOpt is to not spawn inefficient tasks. A profiler pass in our compiler

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 program running time. The energy-centric optimization is to change the model into one that tries to minimize the product of the energy and the square of the running time of the program. On average, the profiler eliminates 39.9 percent of the static tasks in the baseline TLS, and 49.2 percent in energy-centric mode—a significant change.

We also enhanced TaskOpt with a static compilation pass that aggressively prunes tasks that are either very small or whose spawn point has not been moved up very far in the code (such tasks offer little parallelism). We use threshold values to decide the size and spawn hoist distance of the tasks to prune; thresholds are more aggressive in energy-centric mode than in baseline TLS. This static pass eliminates 36.1 percent of the static tasks in energy-centric mode and 34.7 percent in performance-centric mode.

Versioning and dependence checking structures

TLS systems require data versioning because the cache hierarchy must often hold multiple versions of the same datum, such as when speculative tasks have write-after-write (WAW) and write-after-read (WAR) dependences with predecessor tasks. The system buffers the version that the speculative task creates, typically in the processor's cache. If multiple speculative tasks coexist in a single processor, a cache might have to hold multiple versions of the same datum. In such cases, the cache can identify data versions using a version ID on the cache line—in our case, a local ID (LID).

TLS systems must also be able to perform dependence checking, which means that caches must record how the processor accessed each datum. Typically, support for dependence checking augments each cached datum with two access bits—write and exposed-read—which set a bit on a write and exposed read.

A variety of cache access operations read or update the hardware LID and access bits. On an external access to the cache, for example, dependence checking compares the LID of an address-matching line in the cache to the ID of the incoming message. From the comparison and the value of the access bits, the cache can conclude that a violation has occurred, or it can instead supply the data normally.

A distinct use of these TLS structures is in tag group operations, which involve changing the tag state of cache line groups. There are three main cases. The first is when a task is squashed, and its cache lines must be invalidated. The second is when a task commits in eager-commit systems;[1] all the task's dirty cache lines merge with main memory through write backs[2] or ownership requests.[3] Finally, in lazy-commit systems,[1] when a cache has no free LIDs left, it must recycle one, typically by selecting a long-committed task and writing back all its dirty cache lines to memory. That task's LID then becomes free, and the cache can reassign it.

These TLS tag group operations often induce significant energy consumption, however. For some operations, schemes might use a hardware finite state machine (FSM) that periodically and repeatedly walks the cache tags. In one scheme,[4] to recycle LIDs, a FSM periodically selects the LID of a committed task from the LID table, walks the cache tags writing that task's dirty lines back to memory, and finally frees up the LID. The FSM operates in the background using free cache cycles. In another scheme,[3] to commit a task, 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. Execution thus 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, 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.[5] In caches with many versions, however, it is likely to adversely affect the cache access time. For example, the cache in our design uses 6-bit LIDs per cache line.

To address these sources of energy consumption, our TLS CMP design extends each LID table entry 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. These bits are never accessed in the critical path of an L1 cache hit.[6]

With this extra information, the TLS CMP

can perform all the tag group operations lazily and efficiently. A task squash or commit involves only setting a killed or committed bit in the LID table. As replacements eliminate the lines belonging to a squashed or a committed task, the cache controller decrements the corresponding count in the LID table. When the count reaches zero, its associated LID becomes unused and is ready for recycling. Consequently, LID recycling is also very fast.

This energy-efficient design of the LID table is the essence of our *NoWalk* optimization, so called because it largely avoids the eager walk of the cache tags in any tag group operation. It only activates a background walk of the cache tags when there is only one free LID left. With this optimization, a processor might occasionally have to stall because of a temporary lack of LIDs. However, NoWalk eliminates many tag checks. In contrast, the baseline TLS architecture aggressively recycles LIDs;[4] 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* reduces the cost of tag checking by using short LIDs for cache lines rather than global task IDs. As a result, each tag check consumes less energy. Because baseline TLS already uses this optimization,[3] we did not evaluate its impact on energy consumption.

Additional traffic

A TLS CMP system generates more traffic beyond the private L1 caches than does a superscalar processor. Although some of the increase is from parallel execution, there are three main TLS-specific sources of additional traffic.

First, caches must often 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. Consequently, there is less cache space for the currently running task, which causes additional misses.

Second, multiple versions of the same line in the system can cause additional messages. Specifically, when a processor requests a line, multiple versions of it might be provided, and the coherence protocol then selects the 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, speculative cache-coherence protocols should track dependences at a fine grain, typically by using the write and exposed-read bits. If these bits are kept per line, lines that exhibit false sharing might appear to be involved in data dependence violations and, as a result, cause squashes.[7] For this reason, many proposed TLS schemes keep access information at a finer grain, such as per word. Unfortunately, per-word dependence tracking can come at the cost of higher traffic, since the cache controller might need to send a distinct message (such as an invalidation) for every word of the line.

One of the aims of the *TrafRed* optimization is to decrease the version-ID checks needed, and therefore the traffic. TrafRed extends cache lines with a Newest and an Oldest bit. Every time a cache loads a line, the cache controller sets the Newest bit if the line contains the latest cached version of the corresponding address. Likewise, it sets the Oldest bit if the line contains the earliest cached version of the corresponding address. As execution proceeds, an access by another task might reset the Newest bit.

This support eliminates many messages. When a processor writes to a line cached in a nonexclusive state, the baseline TLS checks all the caches with a version of the requested line. The goal is to detect any exposed read to the line from a more speculative task. Such an event would cause a squash. With TrafRed, however, 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 the line's older versions, which cannot remain cached. With TrafRed, if the displaced line has the Oldest bit set, there is no need to check other caches for older versions.

Additional instructions

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 percent. These additional instructions come

from two sources. The first, low quality code due to side effects of breaking the code into tasks, accounts for 88.3 percent of the increase. The quality of TLS code is lower than non-TLS code in part because conventional compiler optimizations are not very effective at optimizing code across task boundaries. In addition, in CMPs such as the ones we consider, where processors communicate only through memory, the compiler must spill registers across task boundaries, adding instructions.

The remaining 11.7 percent instruction increase is from TLS-specific instructions, including task spawn and commit. TaskOpt helps reduce these additional TLS-specific dynamic instructions.

POSH compiler

We have developed POSH, a novel, fully-automated compiler that generates TLS code from sequential integer applications.[8,9] By default, POSH generates code with out-of-order task spawning, although it can also restrict code generation to in-order task spawning. The "Out-of-Order Task Spawning" sidebar describes this approach in more detail.

Out-of-Order Task Spawning

Most proposed TLS systems form tasks with iterations from a single loop level,[1,2] with the continuation of calls to subroutines that do not spawn other tasks,[3] or with execution paths out of the current task.[4] In these proposals, an individual task can spawn at most one correct task in its lifetime. A correct task is one that is in the program's sequential execution rather than in a wrong branch path. Consequently, the processor spawns correct tasks in-order—the same order as in sequential execution.

Figures A1 and A2 show examples of in-order task spawning. In Figure A1, when parallelizing a loop, each task spawns the next iteration. The leftmost task is safe (or nonspeculative); the more a task is to the right, the more speculative it is. Figure A2 shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while the processor spawns a more speculative task to execute the continuation.

In contrast, Figure A3 shows an example of out-of-order task spawning as supported by POSH. The figure shows nested subroutines. The safe task first spawns a task for the continuation of subroutine S1 and then 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, and the order of spawning is strictly the reverse of sequential execution. The task order from less to more speculative is S2 Cont. and then S1 Cont., but these tasks are spawned in reverse order. The same effect occurs in tasks built from iterations of nested loops.

Enabling more parallelism

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 execute in parallel before the program has even spawned the tasks in the code between the two sections. The main disadvantage is that, since all tasks can potentially spawn multiple times, parallelism expands in unexpected parts of the task tree dynamically. As a result, in decentralized architectures such as CMPs, it is hard to support out-of-order spawning and maintain task ordering and efficient resource allocation—two cornerstones of TLS.

Several time-critical TLS operations require task ordering. A task must know its immediate successor to communicate the commit token or prop-

continued on p. 8

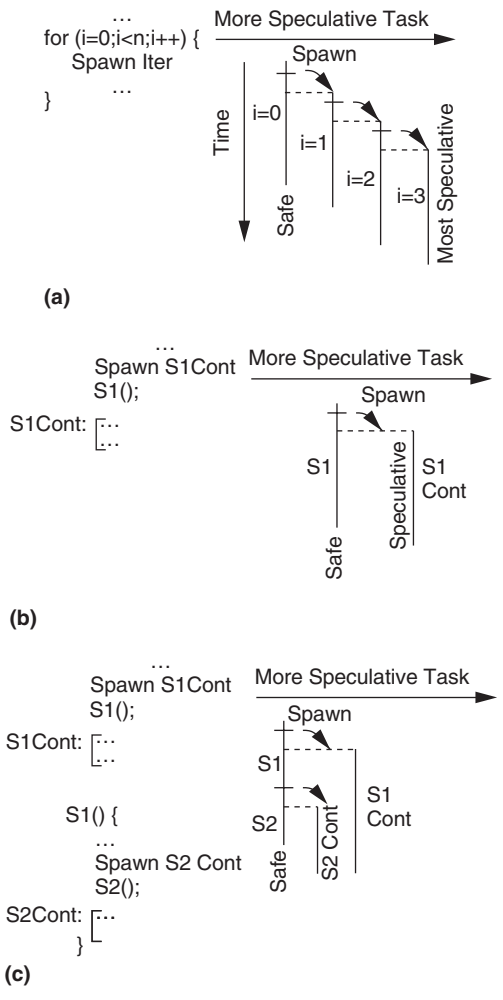

Figure A. Examples of task trees: when parallelizing a loop (a), finding a leaf subroutine (b), and nested subroutines (c). Cont and Iter denote continuation and iteration.

continued from p. 7

agate a squash signal. Moreover, any communication between two tasks requires knowing the tasks' relative order. Unfortunately, with out-of-order spawning, high-speed task ordering is difficult.

Efficiently allocating resources such as CPU or cache space is crucial for TLS performance. Such resources should ideally be assigned to tasks with a high chance to commit. With out-of-order spawning, however, highly speculative tasks might have been running for a long time. If a processor must spawn a less speculative task but there are no free CPUs, should it kill one of the highly speculative tasks?

Microarchitectural mechanisms

We propose three architectural mechanisms to enable high-speed task ordering with out-of-order spawning in a TLS CMP: splitting timestamp intervals, immediate-successor list, and dynamic task merging.[5]

Splitting timestamp intervals

Under in-order task spawning, recording task order is easy; tasks are created in order, so a parent gives its child its timestamp plus one. Under out-of-order task spawning, recording is more complex. To address this, we represent a task with a timestamp interval, given by a base and range timestamp: {*B*,*R*}. When a task spawns, it splits its timestamp interval in two, giving the higher range subinterval to its child (since it is more speculative) and keeping the lower range subinterval. With this support, the parent task can successively provide timestamps to less and less speculative children.

Immediate-successor list

Under in-order task spawning, a task can easily find its immediate successor because the task has spawned it. Under out-of-order task spawning, identifying the immediate successor is not straightforward. In Figure A3, if task S2 Cont. is squashed, it is not trivial for it to identify and squash its successor task S1 Cont., which was spawned before and independently of it. Consequently, we propose that 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 pointer in hardware to its IS. On a spawn, the child gets the value of 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 spawning, highly speculative tasks can hog resources and starve less speculative tasks that are spawned later. To address this issue, we propose dynamic task merging, which consists of the transparent, hardware-driven merging of two consecutive tasks at runtime on the basis of dynamic load conditions. With task merging, running tasks can merge, therefore freeing resources for less speculative tasks. Alternatively, a task can skip the spawn instruction for a child, therefore merging with its child.

References

1. V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers*, vol. 48, no. 9, Sept. 1999, pp. 866-880.
2. J. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 1-12.
3. M Chen and K. Olukotun, "Exploiting Method-Level Parallelism in Single-Threaded Java Programs, *Proc. 7th Int'l Conf. Parallel Architectures* and Compilation Techniques (PACT 98), IEEE CS Press, 1998, pp. 176-184.
4. T. Vijaykumar and G. Sohi, "Task Selection for a Multiscalar Processor," *Proc. 31st Ann. Int'l Symp. Microarchitecture* (Micro-31), IEEE CS Press, 1998, pp. 81-92.
5. J. Renau et al., "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," *Proc. Int'l Conf. Supercomputing* (SC 05), IEEE CS Press, 2005, pp. 179-188.

POSH adds several passes to gcc 3.5, which uses a static single assignment (SSA) tree as the high-level intermediate representation.[10] By building on this software, we were able to leverage a complete compiler infrastructure. Also, by working at the intermediate representation level, we have better information, and it is easier to perform pointer and dataflow analysis than if we worked at register-transfer level.

Task generation and hoisting

POSH uses four modules as potential tasks: subroutines from any nesting level, their continuations, loop iterations from any loop-nesting level, and loop continuations. It also handles recursion seamlessly. In out-of-order task spawning mode (default), it can select all subroutines and loop iterations that are larger than a certain size. In in-order task spawning mode, POSH is more careful, since a task can have only one 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, POSH eliminates tasks from the graph until it can guarantee that each task has only one child.

In either case, once it has selected the tasks, POSH inserts spawn instructions and tries to hoist them to boost parallelism. It hoists a spawn as far as possible, but not above statements that can cause data or control dependence violations. A final task-cleanup pass looks for spawns that POSH hoisted above

only a handful of instructions. In this case, it eliminates the spawn and integrates the two corresponding tasks, thus reducing overhead.

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

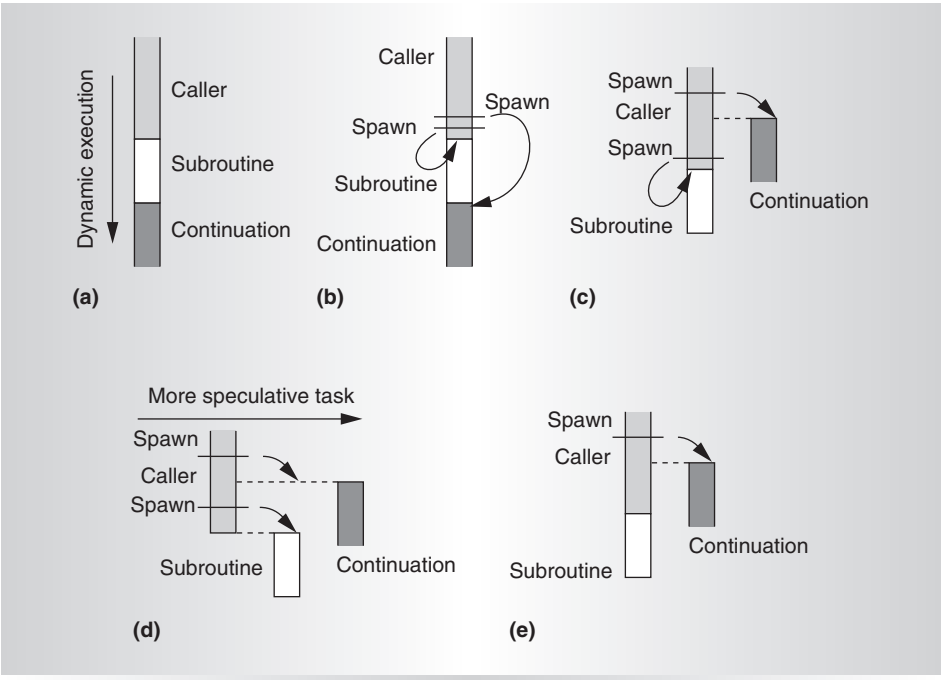

Figure 1. How POSH generates tasks out of a subroutine and its continuation: dynamic execution in and out of the subroutine (a), marking of the subroutine and continuation as tasks and insertion of two spawn instructions in the caller (b), hoisting of the spawn for the continuation (c) and for the subroutine (d), and result of the cleanup pass that eliminates the subroutine spawn because it had little hoisting (e).

Task profiling

POSH includes a profiling pass that uses a simple model to identify additional tasks for elimination, such as those that, because of squashes, are not likely 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 that predecessor tasks have written. With this, it can detect potential runtime violations. The profiler also models a very simple cache to estimate the number of cache misses. The model does not include cache timing. On average, the profiler takes around 5 minutes to run on a 3-GHz Pentium 4. Details on the profiler are available elsewhere.[8,9]

Evaluation

We compared a TLS CMP to a non-TLS chip with a single processor of the same or larger issue width. We used the SESC execution-driven simulator,[11] enhanced with models of dynamic and leakage energy from Wattch,[12] Orion,[13] and HotLeakage.[14] The TLS CMP that we modeled—the TLS4-3i—has four three-issue cores with private L1 caches and a shared L2 cache. The non-TLS chips have a single superscalar core with on-chip L1 and L2 caches. We considered two such chips—one with a six-issue superscalar (Uni-6i) and the other with a three-issue superscalar (Uni-3i) core. The TLS4-3i and Uni-6i chips have approximately the same area.[15,16] We also simulate a TLS CMP with only in-order task spawning—TLS4-3i InOrder.

We measured SPECint 2000 applications with the Reference data set. Uni-3i and Uni-6i ran the binaries compiled with our TLS passes disabled.

Energy cost of TLS

Figure 2 characterizes the energy cost of TLS (ΔE_{TLS}), which is the difference between the energy consumed by our TLS CMPs and Uni-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}.

Each bar is broken into the contributions of the TLS-specific energy consumption

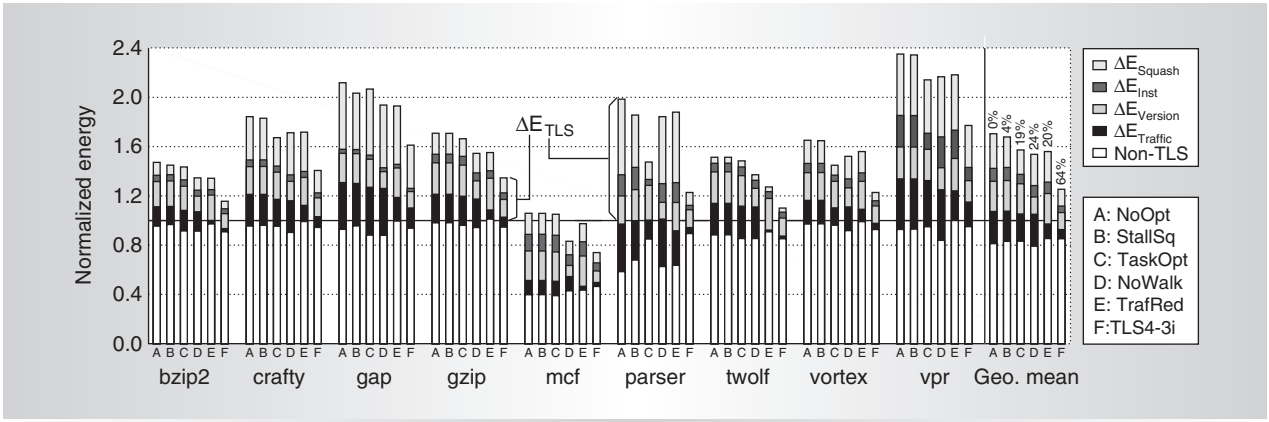

Figure 2: Energy cost of TLS for our four-core TLS CMP chip with and without energy-centric optimizations. The percentages written above the average bars represent the decrease in ΔE_{TLS} (energy cost of TLS) when we applied the optimizations. For each application, the six bars refer to the total energy the chip consumed without any of our energy optimizations (NoOpt); with individual optimizations enabled (StallSq, TaskOpt, NoWalk, or TrafRed); and with all four optimizations applied (TLS4-3i).

sources in Table 1: task squashing (ΔE_{Squash}), additional dynamic instructions in tasks that are not squashed (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, non-TLS should equal 1. In practice, this is not exactly the case. One of the main reasons is that 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.

The NoOpt bars show that ΔE_{TLS} is significant. On average, unoptimized TLS adds 70.4 percent to the energy that Uni-3i consumed. Moreover, all four TLS energy consumption sources contribute noticeably, with ΔE_{Squash} consuming the most.

Impact of optimizations

The other bars in Figure 2 show that our optimizations effectively reduce the TLS energy sources in Table 1. TaskOpt reduces both sources it aimed to reduce, namely ΔE_{Squash} and ΔE_{Inst}. It minimizes ΔE_{Squash} by reducing the fraction of squashed instructions from 22.6 to 17.6 percent on average. It minimizes ΔE_{Inst} by reducing the additional dynamic instructions in tasks that are not squashed from 12.5 to 11.9 percent on average.[6]

NoWalk's target was $\Delta E_{Version}$. It reduced the number of tag accesses relative to Uni-3i from 3.3 to 2.2 times on average.[6] TrafRed also addressed its target, $\Delta E_{Traffic}$, reducing traffic from 19.6 to 5.6 times on average, relative to Uni-3i.[6] Finally, StallSq addresses ΔE_{Squash} , although it has a smaller impact relative to the other optimizations.

Thus, TaskOpt, NoWalk, and TrafRed effectively reduce different energy sources and, combined, cover all the energy-consumption sources considered. When we combined all four optimizations into TLS4-3i, we eliminated on average 64 percent of ΔE_{TLS}. Compared to the overall on-chip energy that NoOpt consumed, this is a very respectable energy reduction of 26.5 percent.

The section of the resulting TLS4-3i bar that is over 1.00 shows TLS's energy cost with the four optimizations, which is on average only 25.4 percent—a remarkably low figure. Moreover, the application slowdown over NoOpt was less than 2 percent on average.[6]

Overall speedup and power consumption

We also evaluated the speed and power consumption of TLS4-3i, comparing it to Uni-3i, Uni-6i, and to our TLS CMP with in-order spawning (TLS4-3i InOrder). Figure 3a shows the application speedup of these architectures relative to Uni-3i. Figure 3b shows the average power consumed during execution. As a reference, the arithmetic mean of the average instructions per cycle of the applications on TLS4-3i is 1.38.

As Figure 3a shows, POSH successfully

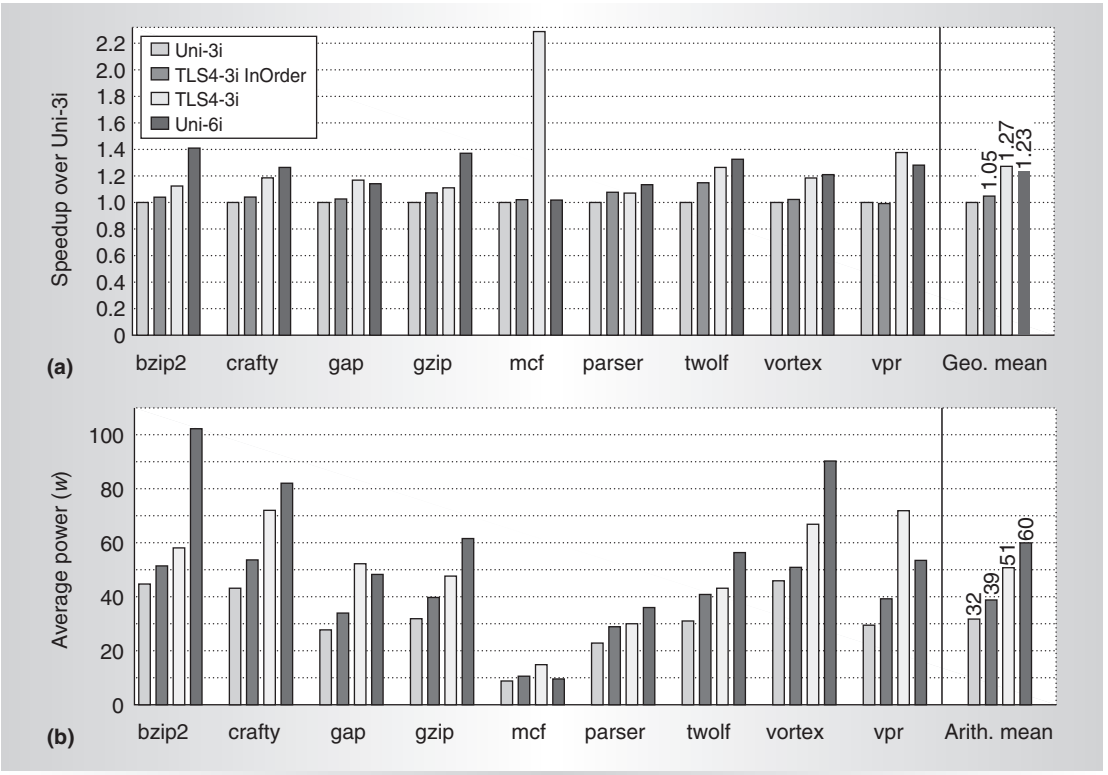

Figure 3. Execution speedup relative to Uni-3i (a) and average power consumption (b) for different chips. The mean for speedups is geometric. On average, TLS4-3i delivered a speedup of 1.27 over Uni-3i.

extracts good tasks from these irregular codes, and TLS4-3i is on average slightly faster than Uni-6i. The speculative parallelism that TLS4-3i enables in these hard-to-parallelize codes is more effective than doubling the issue width. This is a good result because it conservatively assumes the same frequency for both chips. Comparing TLS4-3i to TLS4-3i InOrder shows that an environment with in-order spawning handicaps TLS. TLS only obtains a 1.05 average speedup. For that reason, we favor supporting out-of-order task spawning.

The TLS4-3i speedup for mcf is very large because mcf benefits from TLS tasks that implicitly prefetch data for other tasks. Without considering mcf, the geometric mean of TLS4-3i's speedup is 1.18, which is still comparable to Uni-6i's speedup.

Figure 3b shows that the on-chip power TLS4-3i consumed is on average 15 percent lower than Uni-6i's consumption. Moreover, it never reaches the high values that Uni-6i dissipates in some applications, which shows that a TLS CMP is energy-efficient. On average, TLS4-3i is slightly faster than Uni-6i while consuming 15 percent less power.

Our work has demonstrated the feasibility of energy-efficient TLS CMPs. CMPs offer advantages that are particularly attractive for running explicitly-parallel codes. With TLS support, they can also run in parallel challenging sequential codes such as SPECint 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; by DARPA under grant NBCH30390004; by DOE under grant B347886; and by gifts from IBM and Intel.

References

1. M.J. Garzarán et al., "Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Architecture* (HPCA

03), IEEE CS Press, 2003, pp. 191-202.
2. V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers*, vol. 48, no. 9, Sept. 1999, pp. 866-880.
3. J. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 1-12.
4. M. Prvulovic et al., "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. 28th Int'l Symp. Computer Architecture* (ISCA 01), IEEE CS Press, 2001, pp. 204-215.
5. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VIII), ACM Press, 1998, pp. 58-69.
6. J. Renau et al., "Thread-Level Speculation on a CMP Can Be Energy Efficient," *Proc. Int'l Conf. Supercomputing* (SC 05), IEEE CS Press, 2005, pp. 219-228.
7. M. Cintra, J.F. Martínez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," *Proc.27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 13-24.
8. W. Liu et al., "POSH: A TLS Compiler that Exploits Program Structure," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, March 2006.
9. J. Renau et al., "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," *Proc. Int'l Conf. Supercomputing* (SC 05), IEEE CS Press, 2005, pp. 179-188.
10. "SSA for Trees—GNU Project," May 2003; http: //www.gccsummit.org/2003/view abstract.php?talk=2.
11. J. Renau et al., "SESC Simulator," Jan. 2005; http://sesc.sourceforge.net.
12. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Int'l Symp. Computer Architecture* (ISCA 00), IEEE CS Press, 2000, pp. 83-94.
13. H.S. Wang et al., "Orion: A Power-Performance Simulator for Interconnection Networks," *Proc. 35th Ann. Int'l Symp. Microarchitecture* (Micro-35), IEEE CS Press, 2002, pp. 294-305.
14. Y. Zhang et al., "HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects," Tech. Report CS-2003-05, Univ. of Virginia, CS Dept., 2003.
15. R. Kumar et al., "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. 38th Int'l Symp. Microarchitecture* (Micro-38), IEEE CS Press, 2003, pp. 64-75.
16. P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model," Tech. Report 2001/2, Compaq Computer Corp., 2001.

Jose Renau is an assistant professor of computer engineering at the University of California, Santa Cruz. His research interests include design effort and complexity estimators, CMPs, energy-performance trade-offs, thread-level speculation, processors in memory, and checkpointed architectures. Renau has a PhD in computer science from the University of Illinois, Urbana-Champaign. He is a member of the IEEE Computer Society and the ACM.

Karin Strauss is a PhD candidate in computer science at the University of Illinois, Urbana-Champaign. Her research interests include multiprocessor systems and cache coherence protocols. Strauss has a BEng and MEng in electrical engineering from the University of Sao Paulo, Brazil. She is a student member of the IEEE, IEEE Computer Society, ACM, and ACM SIGARCH and SIGMICRO.

Luis Ceze is a PhD candidate in computer science at the University of Illinois, Urbana-Champaign. His research interests include memory system optimizations for large-window processors, speculative multiprocessor architectures, and compiler support for such systems. Ceze has a BEng and MEng in electrical engineering from the University of Sao Paulo, Brazil. He is a student member of the IEEE, ACM, and ACM SIGARCH and SIGMICRO.

Wei Liu is a research scientist in the Department of Computer Science at the University

of Illinois, Urbana-Champaign. His research interests include computer architecture, compilers for thread-level speculation, and software debugging and testing. Liu has a PhD in computer science from Tsinghua University. He is a member of the IEEE Computer Society and the ACM.

Smruti R. Sarangi is a PhD student in computer science at the University of Illinois, Urbana-Champaign. His research interests include thread-level speculation systems, power management schemes, and processor reliability. Sarangi has a BTech in computer science and engineering from the Indian Institute of Technology, Kharagpur, and an MS in computer science from UIUC. He is a student member of the IEEE.

James Tuck is a PhD candidate in computer science at the University of Illinois, Urbana-Champaign. His research interests include multiprocessor architectures, compilation for speculative multiprocessor architectures, and memory system optimizations. Tuck has a BE in computer engineering from Vanderbilt University and an MS in electrical and computer engineering from UIUC. He is a student member of the IEEE, IEEE Computer Society, ACM, and ACM SIGARCH and SIGMICRO.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. He is also chair of the IEEE Technical Committee on Computer Architecture (TCCA). His research interests include multiprocessor computer architecture, thread-level speculation, low-power design, reliability, and debuggability. Torrellas has a PhD in electrical engineering from Stanford University. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to Jose Renau, Dept. of Computer Engineering, University of California, Santa Cruz, CA 95064; renau@soe.ucsc.edu.