TLS Chip Multiprocessors: Micro-Architectural Mechanisms for Fast Tasking with Out-of-Order Spawn

Draft paper submitted for publication. November 6, 2003.

Please keep confidential

Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, Josep Torrellas
University of Illinois

Abstract

Chip Multiprocessors (CMP) are flexible, high-frequency platforms on which to support Thread-Level Speculation (TLS). However, for TLS to deliver on its promise, CMPs must exploit multiple sources of speculative task-level parallelism, including any nesting levels of both subroutines and loop iterations. Unfortunately, these environments are hard to support in decentralized CMP hardware: since tasks are spawned out-of-order and unpredictably, maintaining key TLS basics such as task ordering and efficient resource allocation is challenging.

This paper is the first one to propose micro-architectural mechanisms that, taken together, fundamentally enable fast TLS with out-of-order spawn in a CMP. These simple mechanisms are: Splitting Timestamp Intervals, the Immediate Successor List, and Dynamic Task Merging. To evaluate them, we develop a TLS compiler with out-of-order spawn. With our mechanisms, a TLS CMP with 2 4-issue processors increases the average speedup of full SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (with out-of-order spawn). Moreover, the resulting CMP outperforms a very aggressive 8-issue superscalar. Specifically, with the same clock frequency, the CMP delivers an average speedup of 1.14 over the 8-issue processor.

1 Introduction

Chip Multiprocessors (CMP) with Thread-Level Speculation (TLS) are being put forward as flexible, high-frequency engines to extract the next level of parallelism from hard-to-analyze programs (e.g. [7, 8, 9, 11, 15, 16, 17, 18, 24]). In these architectures, irregular sequential codes are divided into tasks that are executed in parallel, optimistically assuming that sequential semantics will not be violated. As the tasks run, the architecture tracks their control flow and data accesses. If a cross-task dependence is violated, the offending tasks are destroyed (*squashed*). Then, a repair action is initiated and the offending tasks are re-executed.

While these architectures have shown good potential, often thanks to sophisticated compiler support [2, 5, 10, 19, 20, 23], the speedups obtained for non-numerical applications have typically been only modest. Part of the reason is that most designs have typically focused (often implicitly) on limited types of task structures: iterations from a single loop level (e.g. [4, 9, 23]), the code that follows (i.e. the continuation of) calls to subroutines that do not spawn other tasks (e.g. [3]), or

some execution paths out of the current task (e.g. [20]). In the cases mentioned, tasks are spawned *in order*, namely in the same order as they would in sequential execution. While exploiting only these task structures may simplify the CMP hardware, it cripples TLS potential.

High-level performance evaluation studies have pointed out that there is a sizable amount of other parallelism available [12, 13, 21, 22]. One can execute in parallel all subroutines and their continuations irrespective of their nesting, and iterations from multiple loop levels in a nest. If this additional parallelism is also leveraged, the speedups are predicted to be significantly higher.

In practice, exploiting these additional sources of parallelism requires supporting *out-of-order* task spawning. For example, consider nested subroutines. When a task finds a subroutine call, it spawns a more speculative task to execute the continuation, and proceeds to execute the subroutine. The same task can then find other subroutine calls, therefore spawning speculative tasks that are less speculative than the one spawned first. The same occurs for nested loops, and for combinations of loop and subroutine nesting.

With out-of-order spawning, the application offers unpredictable shapes of parallelism that are hard to manage at run time. The resulting TLS environment is challenging. Specifically, how do we manage task ordering, which is required to identify violations and to ensure commit and squash order? How do we balance resource allocation between highly speculative tasks that have been running for a long time, and just-spawned, less speculative tasks? To address these challenges with high speed in a CMP, we need special micro-architecture.

Unfortunately, no previous work has outlined such CMP micro-architecture. Hammond *et al.* [8] have proposed a TLS CMP system that supports out-of-order spawning with both subroutine and loop-iteration tasks of any nesting level. However, to control key parts of the speculation machinery, they use a co-processor running software handlers (Section 8). They conclude that their scheme has too much control software overhead to support subroutine parallelism.

Other work on tasking with out-of-order spawn has only focused on high-level performance evaluation [12, 13, 21, 22], often simulating ideal architectures. It has not described any micro-architecture design. The one design that presents some micro-architecture for out-of-order spawn is DMT [1], which is based on a single centralized, multithreaded CPU (Section 8). Such solution is not viable for the decentralized architecture of a CMP, which requires completely different support.

This paper is the first one to propose micro-architectural mechanisms that, taken together, fundamentally enable *high-speed* tasking with out-of-order spawn in a TLS CMP. These simple mechanisms support correct and efficient task ordering and resource allocation. Task ordering is enabled with *Splitting Timestamp Intervals* for low-overhead order management, and the *Immediate Successor List* for efficient task commit and squash. Efficient resource allocation is enabled with *Dynamic Task Merging*, which directs speculative parallelism to the most beneficial sections of the code.

To test our micro-architecture, we develop a gcc-based TLS compiler for out-of-order spawn. We show that a simulated TLS CMP with 2 4-issue processors increases the average speedup of

full SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (with out-of-order spawn). Moreover, the resulting CMP outperforms an 8-issue superscalar: with the *same clock frequency*, the CMP delivers an average speedup of 1.14 over the 8-issue processor. Overall, our micro-architectural mechanisms are very effective at boosting TLS speedups.

This paper is organized as follows: Section 2 introduces out-of-order spawning; Sections 3 and 4 present our micro-architectural design and implementation; Section 5 describes the compilation infrastructure; Sections 6 and 7 present our evaluation methodology and the evaluation; and Section 8 discusses related work.

2 Speculative Tasking with Out-of-Order Spawn

In most of the proposed TLS systems, tasks are formed with iterations from a single loop level (e.g. [4, 9, 23]), the code that follows (i.e. the continuation of) calls to subroutines that do not spawn other tasks (e.g. [3]), or some execution paths out of the current task (e.g. [20]). In these proposals, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution path of the program. As a result, tasks are spawned *in order*, namely in the same order as they would in sequential execution.

Figures 1-(a) and (b) show examples. Figure 1-(a) shows the task tree when parallelizing a loop. Each task spawns the next iteration. In the figure, the leftmost task is safe (or non-speculative); the more a task is to the right, the more speculative it is. Figure 1-(b) shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while a more speculative task is spawned to execute the continuation.

There is broad consensus that, for TLS to deliver on its promised speedups, it has to exploit more parallelism. Several high-level performance evaluation studies [12, 13, 21, 22], typically simulating simplified architectures, have pointed to the need to additionally support subroutines from any nesting level and iterations from multiple loop levels in a nest.

Figures 1-(c) and (d) show the two cases. In Figure 1-(c), the safe task first spawns a task for the continuation of subroutine *S1*. Then, it executes the beginning of *S1*, spawns a new task for the continuation of *S2*, and executes *S2* until its end. In Figure 1-(d), the safe task executes outer iteration 0. As it executes, it spawns outer iteration 1, enters the inner loop to execute inner iteration 0, and spawns inner iteration 1. When it completes inner iteration 0, it ends.

With these task choices, an individual task can spawn multiple correct tasks. If so, correct tasks are spawned in strict reverse order compared to sequential execution. For example, in Figures 1-(c) and (d), the safe task spawns two correct tasks, and does so out of order. Figure 1-(e) is a more complex example: the time-line for task creation proceeds from top to bottom (*1-2-3-4-5-6-7*), while sequential order is from left to right (*1-6-7-4-3-5-2*).

In the rest of the paper, to discuss out-of-order spawning, we give examples of tasks built out of any-nesting subroutines and loop iterations, as they are an obvious source of TLS parallelism. Our analysis also applies to any other task structure that maintains two conventions. First, if a

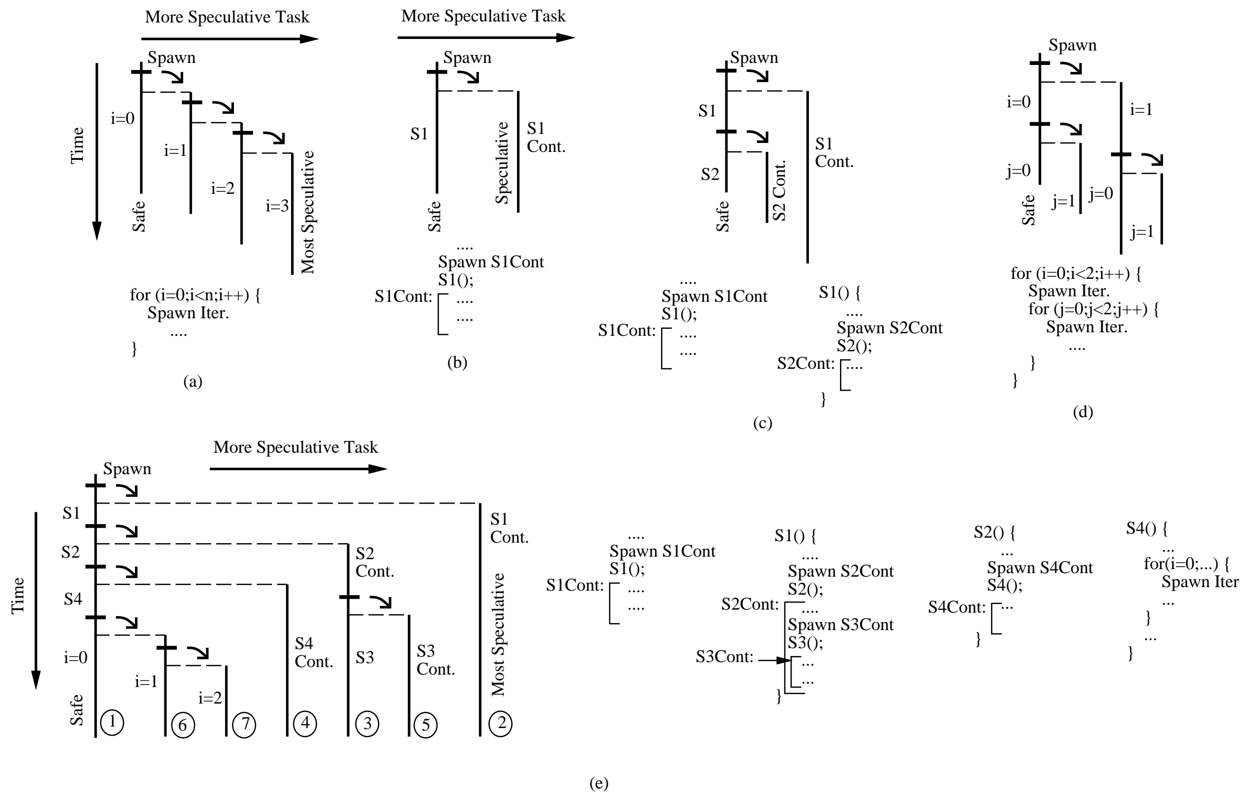

Figure 1: Task trees resulting from different approaches to build TLS tasks. In the figure, Cont and Iter mean continuation and iteration, respectively.

task spawns multiple tasks, the compiler inserts the spawns in strict reverse task order (last task is spawned first, etc). Second, the spawned tasks are less speculative than any task that was more speculative than their parent. These conventions are followed to make the spawn structure like that of nested loops and subroutines. Intuitively, these conventions are unlikely to limit the performance much, while they simplify the micro-architecture.

2.1 Why Supporting Out-of-Order Spawn in CMPs is Hard

Generally, with out-of-order spawn, *all* tasks can spawn, and parallelism expands in unexpected parts of the task tree at run time. As a result, in decentralized architectures such as CMPs, it becomes harder to maintain two cornerstones of TLS: task ordering and efficient resource allocation.

Task ordering is required in several TLS operations that are time-critical. Specifically, a task needs to know its immediate successor, to communicate the commit token or a squash signal. Moreover, any communication between two tasks requires knowing the tasks' relative order: such order determines whether a dependence violation is triggered, or what data version is returned to the requester. Unfortunately, when fine-grain tasks are spawned out of order, unpredictably and in different processors, high-speed ordering of tasks and its maintenance is hard.

Efficient allocation of resources (e.g. CPU or cache space) is crucial for TLS performance. Ideally, resources should be assigned to tasks that are safe or very likely to become so. However, with out-of-order spawning, there may be highly-speculative tasks that have been running for a long time. In this case, if the safe task wants to spawn and there are no free CPUs, should it kill the highly-speculative tasks? This is what past schemes do [8, 12]. Or should it abstain from spawning, do the work itself, and leave the highly-speculative tasks running?

Since decisions on task ordering and resource allocation have to be made very quickly, they need to be supported in the CMP micro-architecture. Given the complexity of TLS designs, however, such new micro-architecture needs to be simple.

2.2 Out-of-Order Spawning and Number of Processors

With out-of-order spawning, TLS can unlock additional parallelism: two code sections that are very separated in sequential execution can be executed *before* some of their intervening code sections have *even been spawned*. This feature enables more task overlap, and can benefit both machines with many processing elements (PE) and those with only few.

On the other hand, it is well-known that some integer applications have only modest coarse-grained parallelism. For example, for SpecInt, few-PE machines have often been a sweet spot. For these applications, even with out-of-order spawning, it is reasonable to target two-PE machines. Indeed, in code sections where, without out-of-order spawning, one of the two PEs of the machine would remain idle, we may now overlap the execution of two tasks that are far apart in sequential execution.

3 Novel Micro-Architectural Mechanisms

We propose three novel and simple micro-architectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. These mechanisms address the key issues of task ordering and efficient resource allocation, in an environment that is statically *unpredictable* (due to out-of-order spawn), *decentralized* (due to the CMP architecture), and has *no broadcast capabilities*. We enable high-speed task order management with *Splitting Timestamp Intervals* (Section 3.1) and the *Immediate Successor List* (Section 3.2). We enable high-speed decisions for efficient resource allocation with *Dynamic Task Merging* (Section 3.3). In the following, when we use the terms *successor* and *predecessor* task, we refer to sequential execution order.

3.1 Splitting Timestamp Intervals for Task Order Management

In any TLS system, tasks have a relative order, which they explicitly or implicitly embed in the CMP protocol messages they issue and the cached data they own. Such order is most obviously needed when two tasks communicate. For example, consider a task reading cached data produced by a second task. The relative order of the tasks is assessed, and the data is provided only if the former task is a successor of the latter. Similarly, consider an invalidation message from a task to data read by a second task. The task order is considered and, if the reader is a successor, a dependence violation is triggered.

Under in-order task spawn, recording task order is easy: since tasks are created in order, it suffices to assign monotonically increasing timestamps to newer tasks. A parent gives to its child its timestamp plus one. With this support, tasks with higher timestamps are successors of those with lower ones.

Unfortunately, such an approach does not work when tasks are created out of order. To maintain order now, we propose to represent a task with a *Timestamp Interval*, given by a *Base* and a *Limit* timestamp ({B,L}). Both base and limit timestamps are operated upon in a task spawn. Specifically, when a task spawns a child, it splits its timestamp interval in two pieces: the higher-range subinterval is given to the child (since it is more speculative), while the lower-range subinterval is kept by the parent. With this support, protocol messages and cached data are directly (or indirectly) associated with the base timestamp. When communication between tasks occurs, the base timestamps of the two tasks are compared *exactly* as in the in-order case.

As an example, Figure 2-(a) shows a program with a call to subroutine *S1*, which in turn calls *S2*. Assume that we use three tasks: task i executes the non-speculative code, j executes the continuation of *S1*, and k executes the continuation of *S2*. The resulting task tree is shown in Figure 2-(b), while Figure 2-(c) shows the timestamp intervals of each task.

The example assumes that the initial interval for task i is {B,L}, and that intervals are partitioned in half. When i spawns j, i keeps $\{B,\frac{L}{2}\}$ and j obtains $\{B+\frac{L}{2},\frac{L}{2}\}$. When i later spawns k, i retains $\{B,\frac{L}{4}\}$ and k obtains $\{B+\frac{L}{4},\frac{L}{4}\}$. With this scheme, as we move from safe to most speculative task following sequential order (i, k, and j), we encounter adjacent intervals ($\{B,\frac{L}{4}\}$, $\{B+\frac{L}{4},\frac{L}{4}\}$, $\{B+\frac{L}{2},\frac{L}{2}\}$) with increasing base timestamps.

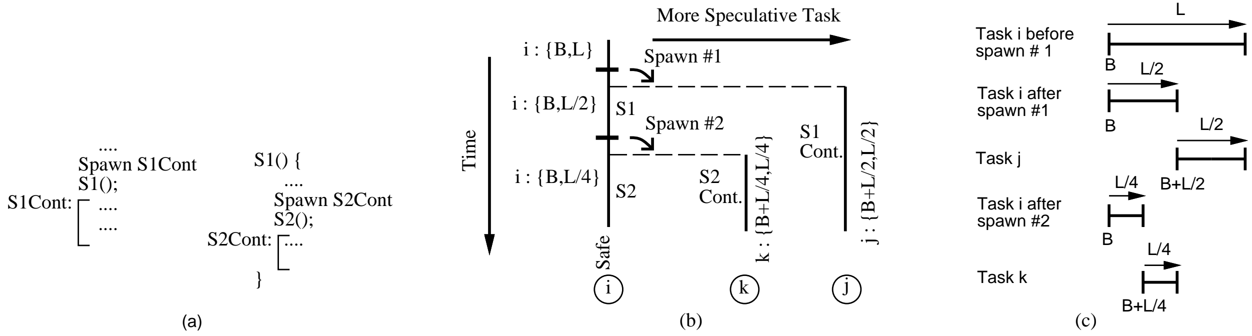

Figure 2: Changes in the base and limit timestamps when tasks are spawned. To save storage, the limit timestamp is encoded as an offset.

In general, a simple approach is to give $\frac{1}{2}$ of the current interval to the child. However, since a task rarely spawns more than a few other tasks, it makes sense to give a larger fraction of the interval to the child. In addition, there are two cases where we can be more efficient. The first one is when the parent knows that the child will not spawn any task; in this case, the parent can give it a single timestamp. The second case is when the parent knows that this is its last child; in this case, the parent can keep a single timestamp. These efficiencies may be obtainable with information gathered by the compiler or hardware predictors.

Our scheme assigns no L to the most speculative task, as it implicitly takes the maximum possible value (L_{max}). This allows the system to dynamically expand the range of used timestamps. Indeed, when the most speculative task spawns a child, it keeps the range {B,L_{max}} for itself, and sets the base of the child to B+L_{max}. The child is now the new most speculative task.

Finally, note that, in some cases, a task may reach a point where it needs to spawn a child and its interval has size 1. In addition, it is possible that a program exhausts the physically representable timestamp range. These infrequent cases are discussed in Section 4.2.

3.2 Immediate Successor List for Task Squash and Commit

In TLS, a task must be able to find its immediate successor very quickly, to perform the time-critical operations of commit and squash. Specifically, when the safe task commits, it passes the commit token to its immediate successor, which may be waiting for it to commit. As for squash, a task is squashed when it reads data prematurely (data violation) or is spawned in the wrong branch path (control violation). In the case of a control violation, the victim task receives a kill signal, which causes the destruction of any state modifications made by the task and terminates the task. In the case of a data violation, the victim task receives a restart signal, which induces the destruction of the state modifications and restarts task execution from its beginning – hoping that the re-execution will read correct data. In either case, a kill signal is also sent to the immediate successor of the victim task and, recursively, to the immediate successor of that one up until the most speculative task. This ensures that all possible side effects of the victim task are erased.

Under in-order task spawn, it is easy to find a task's immediate successor and, recursively,

immediate successors until the most speculative task. For example, consecutively spawned tasks are often allocated on contiguous processors, making it trivial to identify the immediate successor. In other designs, a table with immediate successor information is used, which is easy to maintain because only one task can spawn at a time. Finally, any scheme used is likely to be largely free of protocol races, as only one task spawns at a time.

Under out-of-order task spawn, identifying the immediate successor and all the more speculative tasks is not so straightforward. For example, in Figure 1-(e), if task *7* is killed, it is not trivial for it to identify and kill tasks *4*, *3*, *5*, and *2*, which were created before and independently of *7*. Moreover, any solution has to be carefully crafted to avoid inducing races in the TLS protocol of the distributed CMP if multiple operations happen concurrently.

To support efficient and race-free commit and squash, we propose that the tasks dynamically link themselves in hardware in a list according to their sequential order. We call this list the *Immediate Successor* (IS) list. To build the IS list, we add a hardware pointer to each task structure called the IS pointer. We leverage the fact that, at the time of the spawn, the child is always the immediate successor of its parent. Moreover, the child inherits the parent's immediate successor. Consequently, in our scheme, when a task spawns a child, the hardware gives the parent's IS to the child, and sets the parent's IS to point to the child. Moreover, when a task kills all its successors, the hardware sets its IS to nil. In the example of Figure 1-(e), the IS list links *1* to *6*, *6* to *7*, *7* to *4*, and so on. Task *2*'s IS pointer is nil.

With this support, when a task needs to pass the commit token, it uses the IS list. Moreover, when the victim task in a dependence violation needs to kill all its successors, it sends a kill signal with its own identity downstream the IS list. All successors are killed in turn. When the kill signal reaches a task with a nil IS, an acknowledgment is sent to the originating task, which sets its IS to nil. The result is very fast commit and squash. In addition, our proposal simplifies the TLS protocol implementation in a major way: even when multiple kill and commit signals occur concurrently, since all signals are serialized along the same path, the scheme minimizes protocol races.

3.3 Dynamic Task Merging for Efficient Resource Allocation

In TLS systems, tasks compete for CMP resources such as CPUs, on-chip contexts, and cache space. Under out-of-order task spawn, such competition is harder to manage than under in-order spawn. The reason is that highly-speculative tasks may hog resources and starve more critical (less speculative or even safe) tasks that are spawned later. For example, in Figure 1-(e), when safe task *1* is about to spawn *6*, all the CPUs and contexts in the CMP may be in use by more speculative tasks *4*, *3*, *5*, and *2*.

To allocate chip resources efficiently, we propose a new CMP microarchitectural technique that we call *Dynamic Task Merging*. It consists of transparent, hardware-driven merging of two consecutive tasks at run time. The merging may occur before or after the second task has been spawned. In effect, it enables the machine to prune some branches of the task tree based on

dynamic load conditions. The overall effects of dynamic task merging are an increase in the size of the running tasks and a reduction in their dynamic number.

These effects increase execution efficiency in several ways. First, highly-speculative tasks can be merged, therefore freeing resources for more critical tasks. Second, with large tasks, the overhead related to task spawn has a relatively lower weight, and both caches and branch predictors work better, as a CPU reuses their state for a longer time. Finally, given that the hardware can adjust the number of tasks at run time, the TLS compiler can be more aggressive at creating tasks, which may ultimately lead to higher performance.

Given a pair of tasks, we propose two types of dynamic task merging, depending on whether or not the second task has been spawned. If it has not, dynamic task merging typically involves skipping the spawn instruction of the second task and the task-end instruction of the first task. If the second task has already been spawned, dynamic task merging typically involves killing it and skipping the task-end instruction of the first task.

The first type of task merging can be triggered on any task when it is about to spawn a child. We call it *MergeNext*. The second type of task merging can be triggered on any pair of consecutive tasks in the CMP at any time. However, to maximize efficiency and simplify the implementation, we only trigger it on the two most speculative tasks in the CMP. Consequently, we call it *MergeLast*. Usually, we do it when a new task is about to be spawned somewhere in the CMP.

Note that *MergeNext* and *MergeLast* are not exclusive choices. Overall, every time that a task finds a spawn instruction, we select one of four possible choices: spawn normally, MergeNext, spawn and MergeLast, and both MergeNext and MergeLast. Figure 3 shows the choices when task *4* finds the spawn for *5*. In the rest of this section, we discuss the microarchitecture support for MergeNext and MergeLast, and the heuristics that we use to decide which of the four choices to select. Some compiler implementation details are discussed in Section 5.2.

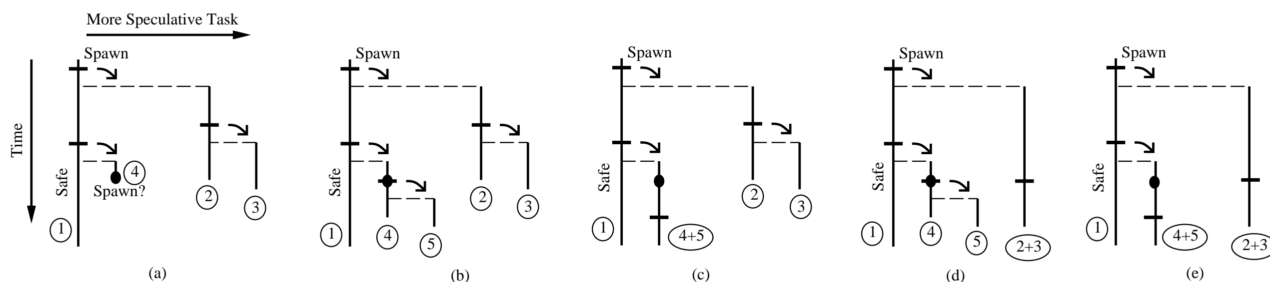

Figure 3: Choices when task *4* finds the spawn for *5*: spawn (b), MergeNext (c), spawn and MergeLast (d), and MergeNext and MergeLast (e).

3.3.1 MergeNext Microarchitecture

A task initiates a MergeNext by skipping a spawn instruction. After that, in the simplest case, the task will also have to skip the first task-end instruction that it finds, and finish only when it finds the second task-end. In general, if a task initiates N MergeNext operations by skipping N spawns,

it will have to also skip *N* task-ends and complete only when it finds the *N+1* one.

Consider now a task that skipped a spawn in a MergeNext and later spawns a child. In this case, since the child is more speculative, the responsibility to complete the task merge is "passed on" to the child: the child will skip the first task-end that it finds and finish only at the second one. As for the parent, it simply finishes at the first task-end that it finds.

The microarchitecture needed to support MergeNext is a counter in the processor called *Number of Ends to Skip* (NES). The NES belongs to the running task, and is checked and modified in hardware. Specifically, when a task initiates a MergeNext, the NES is incremented. When a task finds a task-end instruction, the NES is checked. If it is non-zero, it is decremented and the end instruction is skipped. Otherwise, the end is executed. Moreover, when a task spawns a child, its NES is copied to the child's and is then cleared. The child now owns the merges.

A task's NES is affected by two more events. First, when a task becomes the most speculative one (its IS pointer becomes nil), its NES ceases to matter — the task simply skips any task-end instruction that it finds. This is the appropriate behavior for the most speculative task, which should not be stopped by end instructions. However, if the task spawns a child, the NES of both tasks are updated as usual. The second special event occurs when a task gets restarted (Section 3.2). In this case as the task recovers its initial state, it also recovers its initial NES.

3.3.2 MergeLast Microarchitecture

MergeLast involves killing the most speculative task in the CMP and ensuring that, when the new most speculative task completes its own code, it executes the code of the killed task.

The microarchitecture needed to support MergeLast is the IS list (Section 3.2). A task initiates a MergeLast by sending a MergeLast hardware signal down the IS list. Each task in the list passes, in hardware, the signal and its own identity to its successor. When the signal reaches a task with a nil IS pointer, that task sends an acknowledgment to its immediate predecessor (whose identity it knows) and terminates. The immediate predecessor sets its IS pointer to nil, as it is now the most speculative task. No other action is necessary. When the latter task reaches its end, it will skip it and continue executing, effectively merging its code with that of the killed task. This is because, as discussed in Section 3.3.1, a task with a nil IS pointer skips task-ends.

Note that the operation of a task killing all its successors after a violation (Section 3.2) is similar to a MergeLast except that all the tasks downstream the IS list are killed. In fact, to keep the hardware simple, we implement such an operation as a set of MergeLast operations: the killing task keeps issuing MergeLast operations until it becomes the most speculative task.

3.4 Task Merge Heuristics

Every time that a task finds a spawn instruction, decisions on task merging are made. To keep the hardware simple and the overheads low, in this paper we propose a simple decision algorithm.

The algorithm is based on two notions. First, we conservatively assume that any running task,

even if highly speculative, is likely to perform useful work. Consequently, we try to avoid killing tasks. Second, we rely on squash information to reduce useless work. Specifically, if a task has been restarted twice due to violations, it is not allowed to get a CPU anymore. It simply remains in one of the several on-chip task contexts until it becomes safe. This policy prevents highly-speculative, frequently-squashed tasks from clogging the CPUs. It also allows the hardware to estimate the level of load in the CMP by examining the fraction of on-chip task contexts that are in use.

With this support, we use the following algorithm. We use the CPU usage to decide on MergeNext. If all CPUs are busy, since they appear to do useful work, we perform MergeNext. However, every *NumMNext* MergeNexts, we skip one to prevent tasks from becoming so large that a squash would be very costly.

As for MergeLast, we decide based on the estimated use of on-chip task contexts. If most of them are used, it is likely that many highly-speculative, frequently-squashed tasks are waiting. In this case, one could be killed with little performance penalty. While we could perform a MergeLast only when no context is free, the operation would then be in the critical path. Consequently, we use a threshold: if the estimated number of used contexts is over Th_{MLast} at the time of a spawn, we perform MergeLast.

4 Implementation Issues

To complete the architectural design for out-of-order tasking, this section discusses three related implementation details: task contexts, special cases in handling timestamp intervals, and scheduling tasks to CPUs.

4.1 Implementation of Task Contexts

Each processor has a table of task contexts, which keeps state for the tasks that are loaded on the processor. Of these tasks, only one is running at a time. Each context stores the following state for a task: {B,L} timestamp interval, IS pointer, NES, start PC of the task, and a pointer to a stack location with saved register state. This stack state is not read at the beginning of the task. Rather, it is read on a per-need basis. The context also has the Local ID (LID) associated to the task. As in many TLS systems, this LID is a short ID used to tag the cache lines accessed by the task. It acts as a form of indirection [16] that avoids the need to tag the lines with the whole B timestamp of the task.

The table of task containers is accessed by instructions such as spawn, and hardware signals such as restart or kill. Consider, for example, the case when a task must kill all its successors. In this case, the hardware passes the kill signal from the originating task down the IS list. For each task in the list, the operation is as follows. If the task is running, it is stopped. In all cases, the task's LID is marked as invalid, so that the task's cache lines become invalid and can be purged lazily. As in typical TLS systems, that LID remains unused until all its lines are purged from the cache; at that point, it can be reused.

If a task needs to be restarted, the initial PC and stack pointer are restored from the task

context. A new LID is then assigned.

4.2 Special Cases in Timestamp Intervals

There are two infrequent, special cases when handling timestamp intervals. The first one is when a task wants to spawn a child and has no interval to assign. In this case, it simply sends a kill signal down the IS list. This operation kills all successors, making the task the most speculative one. At this point, the task can obtain as many timestamps as needed (Section 3.1).

The second case is when a program exhausts the physically representable timestamp range. Our solution is to recycle old timestamps in chunks. For that, we divide the whole representable timestamp range into four chunks, based on the two most significant bits of B. When all the tasks with intervals in the lowest chunk (e.g. the 00 chunk) have committed, we recycle the chunk. This involves sending a reprogramming signal to the logic of the timestamp comparators so that timestamps in the recycled chunk are now the highest (i.e. 00 is more speculative than 11). Then, we can start assigning timestamps from the chunk to newer tasks.

The reprogramming signal is issued in the infrequent case that a task with an interval that straddles two chunks commits. With this approach, all the tasks in the CMP can at most use $\frac{3}{4}$ of the whole timestamp range at a time. To see how many tasks can be concurrently supported, assume that B and L have b and l bits, respectively. If, in the worst case, each task has a single child, and the child is given the maximum timestamp range possible (2^l), the maximum number of tasks is then $\frac{3}{4} \times 2^{b-l}$. Consequently, if we want to support about 20 concurrent tasks, $b - l$ should be at least 5.

4.3 Scheduling Tasks to CPUs

While all the tasks that have been spawned have their state loaded on on-chip task contexts, only as many tasks as CPUs can be running at a time. In practically all TLS proposals, tasks are scheduled strictly based on how speculative they are. Specifically, a less speculative task always preempts more speculative ones. Moreover, among the eligible tasks, the preempted one is the most speculative.

In practice, our evaluation will show that such a policy is an overkill, given the typical load and task sizes in our CMP, and our new task merging support. Consequently, we propose and use a simpler policy: we assign high priority to the non-speculative task, and a fixed low priority to all speculative tasks. There are no complex priorities and only the safe task can preempt.

4.4 Other Aspects

Most of the other aspects of a TLS CMP change little as we move from an in-order to an out-of-order spawning framework. For this reason and for brevity, we feel it is unnecessary to detail them. For example, our CMP uses a TLS protocol with lazy commit and multi-versioned L1 and L2 caches similar to [14]. As in that protocol, cache lines with speculative state cannot be displaced. If space is needed and the line is overwritten with a new address, the owner speculative task is sent a restart signal. When the task is re-scheduled again, it will restart. Context switches and

exceptions also cause restarts [14].

5 Compilation Support for Tasking with Out-of-Order Spawn

We have developed a full TLS compiler that generates in-order and out-of-order tasking out of sequential, integer applications. The compiler adds several passes to a still experimental branch of gcc 3.5. The branch uses a static single assignment tree as the high-level intermediate representation [6]. Building on this software allows us to leverage a complete compiler infrastructure. For example, we annotate the control flow graph structure with high-level information as we generate the tasks. Also, working at this high level is better than using a low-level representation such as RTL: we have better information and it is easier to perform pointer and dataflow analysis. At the same time, our transformations are much less likely to be affected by unwanted compiler optimizations than if we were working at the source-code level.

The resulting code quality, both when we enable and disable TLS, is comparable to the MIPSPro SGI compiler for integer codes at the O3 optimization level. This is because, in addition to using a much improved gcc version, we also use SGI's source-to-source optimizer (copt from MIPSPro). The latter performs PRE, loop unrolling, inlining, and other optimizations.

In the following, we highlight three issues: task generation, task merging, and profiling.

5.1 Task Generation and Hoisting

Our compiler extracts the following modules as individual tasks: subroutines from any nesting level, loop iterations from potentially multiple loops in a nest, and whole loops. All subroutines are extracted unless they are very small (in which case they are inlined) or they are libc functions that have system calls. Recursivity is handled seamlessly. In loop nests, the compiler makes decisions based on minimal loop iteration size.

As an example, Figure 4 shows how the compiler generates tasks out of a subroutine and its continuation. Chart (a) shows the dynamic execution into and out of the subroutine. The compiler first marks the subroutine and continuation as tasks, and inserts spawn statements (Chart (b)). After that, another compiler pass tries to hoist spawns. The goal is to get as much parallelism as possible, while making sure that the children of a task are spawned in reverse order, as discussed in Section 2. A spawn is hoisted as far up as we can, as long as the new position dominates the old one and both positions have execution equivalence [1]. We do not hoist past statements that can cause data or control dependence violations. Continuing with our example, Chart (c) hoists the continuation, while Chart (d) adds the hoisting of the subroutine. As usual, tasks on the right side are more speculative.

A final "task clean up" compiler pass looks for loops, subroutines, and iterations that were hoisted only a handful of instructions. In any such case, both the hoisting and the spawn instruction are eliminated, and the two corresponding tasks integrated into one. This pass eliminates

[1] We informally define execution equivalence as two blocks that are control equivalent and are executed the same number of times.

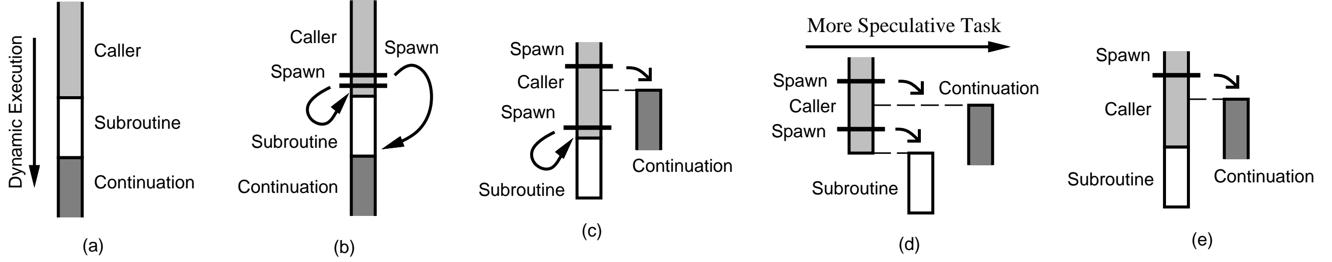

Figure 4: Generating tasks out of a subroutine and its continuation.

unnecessary spawn overheads. In the example, since the subroutine had little hoisting, the code is changed to Chart (e).

5.2 Task Merging

Task merging requires that, as a task completes its code, it goes on executing the code of its immediate successor. This means that the task must have a way of obtaining the live-in register values for its continuation code. In ordinary conditions, such code belongs to another task. However, with our compiler, a task can easily obtain the live-ins for its continuation and successor tasks. This is because all register values changed by a task that may be used by successors are sent to memory when the task finishes. Moreover, all the values needed by a task are read from memory.

5.3 Profiling Support

The compilation process for both in-order and out-of-order tasking includes running a simple profiler. The profiler takes the TLS executable and identifies those task spawn points that should be removed because they are likely to induce harmful squashes according to our models. The profiler returns the list of such spawns to the compiler. Then, the compiler generates the final TLS executable by removing these spawns and integrating their target tasks with the tasks' statically preceding code. On average, the profiler takes about two minutes to run.

The profiler executes the binaries sequentially, using the Train data set for SpecInt codes. As the profiler executes a task, it records the variables written. As 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 violations. The profiler also models a cache to estimate the number of misses in the real machine's L2, although no timing is modeled.

The profiler identifies those spawns where the ratio of squashes per task commit is higher than R_{squash}. For each of those spawns, it estimates the performance benefit that a task squash brings. Some benefit comes from the data prefetching provided by cache misses recorded before the task is squashed ($M_{squashed}$). Other benefit comes from true overlap of the instructions in the task with other tasks, as the task is re-executed after the violation ($I_{overlap}$). With these measurements, the profiler requests spawn removal if $T_I \times I_{overlap} + T_0 \times M_{squashed}$ is less than a threshold T_{perf}. In the formula, T_0 is the estimated average stall time per L2 miss, and T_I is the estimated execution time per instruction. The values for R_{squash}, T_0, T_I, and T_{perf} are listed in Section 6.

6 Evaluation Methodology

To evaluate TLS with out-of-order spawn, we use execution-driven simulations with detailed models of out-of-order superscalar processors and memory subsystems. The proposed architecture is a two-processor CMP with TLS support, which we call *TLS2*. Each processor in *TLS2* is a 4-issue core similar to a PowerPC 970. It has a private L1 cache that buffers multiversioned speculative data. Since an L1 cache may contain more than one version of the same line, we set its access time to a high value: 3 cycles. The L1 caches are connected through a crossbar to an on-chip shared L2 cache. The CMP uses a TLS coherence protocol with lazy task commit and multi-versioned L1 and L2 caches similar to [14]. One aspect of the protocol in [14] that is not supported is the overflow area.

For comparison purposes, we also model a chip with a single, very aggressive 8-issue processor. The chip has no TLS support. We call this architecture *8issue*. For this architecture, since the L1 does not support multiple versions, we set the L1 access time to a lower value: 2 cycles. Moreover, in our comparison, we use the same processor frequency for both *8issue* and *TLS2*. In a real implementation, the frequency of *TLS2* would be *higher* than *8issue*, therefore boosting the relative performance of *TLS2*. The complete set of parameters is shown in Table 1.

Processor Parameters	PROPOSED: *TLS2*	COMPETITION: *8issue*	Memory System and Tasking Parameters
Cores/chip	2	1	L2 cache size, assoc, line: 1 MB, 8, 64 B
Running tasks/core	1	1	L2 OC, RT: 2, 10
TLS hardware?	Yes	No	Mem bandwidth, RT: 8 GB/s, 400 cycles
Frequency	**5 GHz**	**5 GHz**	Task contexts/processor: 5
Fetch, issue, retire width	8, 4, 6	16, 8, 12	LIDs/processor: 128
ROB, I-window size	192, 96	360, 224	B, L timestamp size: 33, 28 bits
LD, ST queue	64, 48	128, 96	*NumMNext*: 10
Mem, int, fp units	2, 3, 2	4, 6, 4	Th_{MLast}: 16
Branch predictor:			Latency to kill mis-speculated task (min):
Penalty	17 cycles	17 cycles	From violation to proc
BTB	2 K, 2 way	2 K, 2 way	notification: 20 cycles
global gshare(11)	16 K	16 K	Time to drain proc pipeline: 17 cycles
local (2 bit)	16 K	16 K	Fraction of interval given to child: 3/4
L1 cache:			R_{squash}: 0.55
size, assoc, line	8 KB, 4, 32 B	8 KB, 4, 32 B	T_0: 390 cycles
OC, RT	1, **2**	1, **1**	T_I: 1 cycle
RT to neighbor's L1	7 cycles	—	T_{perf}: 100 cycles

Table 1: Architectures considered. In the table, OC and RT stand for occupancy and minimum-latency round trip from the processor, respectively. All cycle counts are in processor cycles. In our comparison, we use the same processor frequency for both *8issue* and *TLS2*.

In our experiments, we also use two more architectures built out of the 4-issue cores in *TLS2*: *4issue* and *TLS4*. *4issue* is a chip with a single core, one L1, one L2, and no TLS support. *TLS4* is an extension of *TLS2* that has 4 cores on chip; for simplicity, all parameters are the same as in *TLS2* except for the number of cores.

TLS2 uses as default the microarchitecture introduced in Sections 3 and 4.

Task spawn is performed by an instruction that takes the start PC of the child task. The instruction assembles a small packet that includes the child's start PC, timestamp interval, and

stack and IS pointers, as well as the parent's NES. In the meantime, to decide on task merging, the system estimates the load of the task contexts in the CMP, and if the other CPU is free. If it is decided to spawn on the other CPU, the packet is sent to it. Eventually, the child will execute there, accessing its live-ins through memory via the stack pointer.

Task commit is performed by an instruction that passes the commit token to its IS task. As for task context switch, most of the overhead is due to saving and restoring the registers. Note, however, that a task does not need to save all its 32 registers; only the ones marked as dirty are saved.

We drive our simulated architectures with the SpecInt 2000 applications running the Ref data set. We run all the SpecInt 2000 codes except four that either fail our compilation pass (*eon*, *gcc*, *perlbmk*), or cannot be run in our simulator (*vortex*). As shown in Table 2, we compare four different SpecInt binaries: unmodified binaries (*BaseApp*), TLS with in-order spawning (*InOrder*), and TLS with out-of-order spawning (*OutOrder*).

Name	TLS?	Description of Binary
BaseApp	N	Out-of-the-box, sequential version compiled with *O2*. No TLS instrumentation
OutOrder	Y	Our proposed out-of-order task spawning. Spawns to: (1) a procedure call (2) continuation of any procedure, and (3) iterations from multiple loops in nest
InOrder	Y	In-order task spawning. Selects the same tasks as *OutOrder*. Uses interprocedural analysis pass to eliminate tasks that violate the in-order spawning requirement.

Table 2: Versions of the SpecInt 2000 binaries executed.

These binaries are very different. Specifically, the TLS passes re-arrange the code into tasks and adds extra instructions for spawning and commit. In addition, these transformations obfuscate some conventional compiler optimizations, sometimes rendering them less effective. Consequently, to accurately compare the performance of the different binaries, we cannot simply time 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 (typically 1-6 billion instructions), we execute up to a certain number of markers for all binaries, so that the *BaseApp* binary graduated more than 500 million instructions.

7 Evaluation

7.1 Execution Speedups

To evaluate our proposed support for TLS with out-of-order spawn, we compare the execution time of the in-order TLS binary *InOrder* and the *OutOrder* one running on the *TLS2* architecture. Of course, only the *OutOrder* binary can leverage our microarchitectural mechanisms. For comparison purposes, we also measure the execution times of the *BaseApp* binary running on the *4issue* and *8issue* architectures. The comparison to *4issue* shows the speedup of TLS relative to a single processor of the same size; the comparison to *8issue* shows the speedup of TLS relative to a much larger processor under the same frequency. Finally, we also evaluate *OutOrder* running on *TLS4*, to assess the effect of the number of processors in the CMP.

Figure 5 shows the speedups of the different binary-architecture combinations relative to

BaseApp running on *4issue*. For consistency, all the speedups in Section 7 are shown relative *BaseApp* running on *4issue*. The figure shows data for each application and the harmonic mean. On top of some of the bars, we show the average speedups. In addition, for the TLS bars, we show the ideal contribution of parallelism to the speedup as a black dot on each of the bars. The effect of parallelism on performance is discussed later.

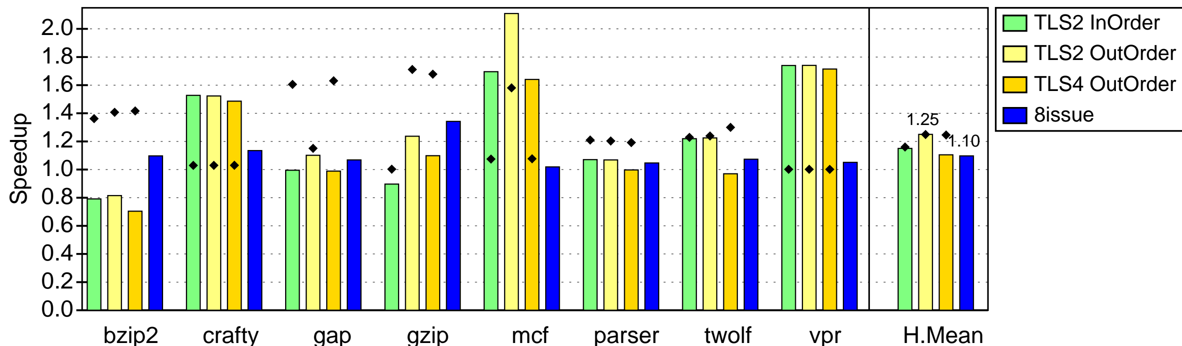

Figure 5: Speedups of different binary-architecture combinations relative to *BaseApp* running on *4issue*. The TLS bars also show the ideal contribution of parallelism to the speedup.

Consider first *OutOrder* on *TLS2* as compared to *InOrder* on *TLS2*. From the figure, we see that *TLS2 OutOrder* is equal-to or outperforms *BaseApp* on *4issue* for all of the applications. On average, it delivers a speedup of 1.25 over such an environment, while *InOrder* only attains an average speedup of 1.15. Two main factors give *OutOrder* the advantage. Firstly, *OutOrder* can select tasks from a larger number of locations in the program and can spawn the tasks earlier due to out of order hoisting. Better task selection enables *gzip* to attain a larger amount of parallelism, leading to good speedups. The second factor is our proposed microarchitectural mechanisms. Efficient hardware support for managing tasks lessens the penalty for restarting and killing tasks. Consequently, we are able to select tasks that provide good prefetching, as is the case with *mcf* which attains a superlinear speedup over executing on *4issue*. This is discussed more in Section 7.2. Overall, because the *OutOrder* binary can spawn tasks with fewer restrictions and benefits from the proposed microarchitectural mechanisms, it performs as well or better than *InOrder* on *TLS2*.

Consider now *OutOrder* running on *TLS2* or on *TLS4*. An increase in the number of processors on the CMP does not improve performance, rather, it decreases from 1.25 to 1.10, on average. When we distribute work across two more processors, we incur more overheads in the memory system and increase the likelihood of restarts because we spawn more tasks. Overall, *OutOrder* on *TLS4* is slightly worse, but is still an important design point for TLS systems because parallel and numerical applications can exploit larger numbers of processors.

Consider now *BaseApp* on *8issue*, which provides an average speedup of 1.10 over *4issue*. *TLS2* has an average speedup of 1.25 with *OutOrder*. Consequently, TLS with out-of-order spawning is able to attain a 1.14 speedup over a much larger processor operating at the same frequency. Overall, *OutOrder* on *TLS2* is competitive with a high frequency, wide-issue, microarchitecture.

7.2 Characterization

To understand the performance of *OutOrder* on *TLS2*, Table 3 provides some run-time measurements. Column 2 shows the average number of busy CPUs, which gives a sense for the degree of parallelism. On average, 1.4 CPUs are in use. These figures are small because of the limited parallelism present in SpecInt codes. However, not all the CPU activity is useful, since some tasks are squashed. Column 3 shows the fraction of busy cycles that execute non-squashed tasks. Fortunately, on average 93% of the work done by the CPUs is useful.

App	Busy CPUs	Useful/ Busy Cycles (%)	Extra Dynamic Instr (%)	Task Size (Instr)	Committed Out of Order Spawn (%)	# Spawns in Task Lifetime		# Events / Task Commit		
						No Merge	Merge	Merge Next	Merge Last	Restart
bzip2	1.708	82.40	12.435	22353	3.78	1.006	1.124	3.3488	0.0005	0.4064
crafty	1.032	99.84	9.451	3431	0.00	1.000	1.000	0.0002	0.0000	0.9980
gap	1.426	80.73	8.904	1401	0.00	1.352	1.001	0.7363	0.0000	0.9512
gzip	1.734	98.68	24.872	1884	0.07	1.271	1.000	3.8905	0.0000	0.2816
mcf	1.912	82.65	82.945	80	30.59	1.000	1.287	2.6479	0.2010	1.0234
parser	1.216	98.94	21.315	279	0.00	1.052	1.000	0.0699	0.0000	0.2382
twolf	1.239	99.97	29.493	115	0.00	1.000	1.000	0.0519	0.0000	0.0000
vpr	1.002	100.00	8.618	27311	0.000	1.000	1.001	0.2623	0.0000	0.0000
Avg	1.409	92.90	24.754	7107	4.30	1.085	1.052	1.3760	0.0252	0.4874

Table 3: Characterizing the run-time behavior of *OutOrder* on *TLS2*.

From the time spent by CPUs executing useful work, we can compute the *ideal* contribution of parallelism to TLS speedup. Figure 5 shows such a contribution for all TLS bars. Note that such a contribution is not equal to the distance between 1 and the top of the bars. The reason is that there are other effects that increase or decrease the speedup. Specifically, the speedup relative to *BaseApp* on *4issue* decreases due to at least two effects. First, the TLS environments have multiple L1 caches and processors on chip, which induces a higher number of cache misses and branch mispredictions, respectively. The second effect is the higher dynamic instruction count under TLS. The reason is the additional spawn, commit, and memory instructions, and the lower effectiveness of conventional compiler optimizations (Section 6). This effect can be seen in Column 4 of Table 3, which shows that TLS execution increases the dynamic instruction count by 25%.

On the other hand, TLS speedups can increase beyond that given by parallelism due to at least two factors. One is the data prefetching effect induced by speculative tasks, which bring into caches data that can later be used by other tasks. Prefetching can be beneficial to performance regardless of the degree of parallelism. The second factor is the additional resources on chip, which includes multiple BTBs and more functional units. However, the performance contribution due to additional resources will contribute only when parallelism can be exploited.

Overall, Figure 5 shows that some of these effects have at least as much importance as parallelism in TLS speedups on SpecInt applications. For example, in *mcf* using the *OutOrder* on *TLS2*, the parallelism is 1.58, while the speedup is 2.11. *mcf* obtains this speedup even in the presence of large overheads, as shown in Column 4 of Table 3. *mcf* also has a large percentage of wasted busy cycles which benefited execution via data prefetching. On the other hand, *gzip* exhibits a large degree of parallelism coupled with an average speedup. Since more than 98% of the work in *gzip* is useful, the speedup comes primarily from parallelism.

Column 5 of Table 3 shows the average number of graduated instructions in the tasks that

commit. On average, such tasks contain 7107 instructions. Given the frequent task merge operations, this is a small number of instructions. Obtaining large speedups with such small tasks is challenging.

The next few columns show parameters related to out-of-order spawning: the average number of tasks that are spawned out-of-order and successfully commit, and the average number of children spawned by a task in its lifetime (Columns 7-8) with and without the benefits of dynamic task merging. *mcf* uses the out-of-order hardware the best, with 30% of its committed tasks being spawned out-of-order. However, we can see from Column 6 that many benchmarks do not exploit out-of-order spawning when merging is enabled. These small percentages do not mean that out-of-order spawn should not be supported. Merging makes out-of-order spawning less frequent because we choose to skip a spawn instruction. Furthermore, out-of-order spawn is necessary to get speedups in *gap* and *gzip*. Without out-of-order spawn support, we could not select the tasks in *gzip* that provide good performance. *gzip* does benefit from out-of-order spawning even though the effect of merging reduces the number of spawns in a task lifetime from 1.3 down to 1.0. Overall, task merging decreases the number of spawns per task, thereby reducing the contribution of out-of-order spawns in the execution of the benchmark.

Finally, the last three columns show the frequency of key events in occurrences per task commit. We can see that all these events occur during execution. MergeNext is the most frequently occurring event of the three. For example, *gzip* performs a MergeNext approximately four times per task, providing it with one of the larger average task sizes. Its high percentage of useful work and large parallelism allows MergeNext more opportunities to occur per task. The MergeLast event occurs when resources need to be reclaimed and when a task needs to kill its successors. For most of the benchmarks, this event happens rarely, and for a number of benchmarks, it never occurred in our simulations. Finally, restart signals occur more frequently than MergeLast because restarts are necessary on any dependence violation. Even though a task is restarted half the time it is spawned, on average, it typically happens early in the task execution. This keeps the ratio of useful to busy cycles high.

7.3 Architecture Sensitivity Analysis

In this final section, we examine design variations. While we would like to assess the impact of each of our microarchitectural mechanisms separately, we cannot disable the IS list or the timestamp intervals because they are directly needed to support out-of-order spawn. However, we can measure the effect of disabling dynamic task merging. We also measure the impact of never running out of timestamps and of using an advanced task scheduling algorithm.

Figure 6 compares the speedup of our proposed *TLS2*, to *TLS2* with the individual changes mentioned. In all cases, the binary used is *OutOrder*. As usual, all the bars show speedups relative to *BaseApp* running on *4issue*. In the following, we consider each case in turn.

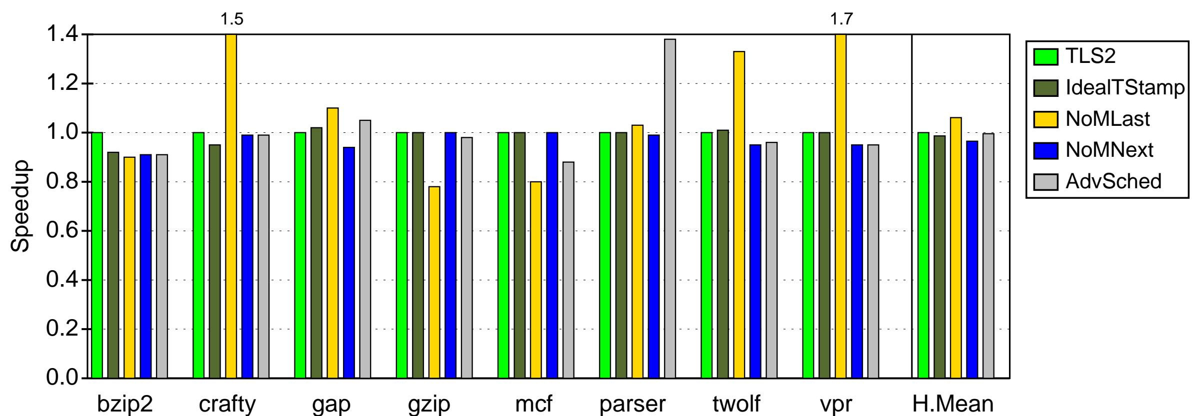

Figure 6: Comparing our proposed *TLS2* to *TLS2* with different architectural or software changes. In all cases, the binary used is *OutOrder*.

7.3.1 Disabling Dynamic Task Merging

The *NoMNext* and *NoMLast* bars in Figure 6 correspond to *TLS2* without MergeNext and without MergeLast, respectively. With *NoMNext*, the speedups are reduced by 4% on average. Disabling MergeNext benefits none of our simulated benchmarks. The main cause is the larger amount of work wasted due to squashes. Indeed, without MergeNext, tasks spawn more children. To assign on-chip contexts to all the less-speculative tasks, we have to squash more-speculative tasks frequently. As a result, *NoMNext* wastes more cycles on squashed tasks than *TLS2*. Consequently, we recommend supporting MergeNext.

With *NoMLast*, several programs, *gzip* for example, run much slower. The reason is that, typically, there are many highly-speculative tasks that use up all the task contexts, therefore preventing the non-speculative task from spawning. As a result, the CMP is using the resources poorly. A few benchmarks do gain from *NoMLast* because their average task size is increased and because the most speculative tasks are left running to either contribute to useful work or to provide prefetching for the less speculative threads. Overall, to unclog the CMP when necessary, we recommend supporting MergeLast.

7.3.2 Never Running out of Timestamps

In *TLS2*, when a task runs out of timestamps, it kills all its successors (Section 4.2). We would like to assess the overhead of these events. For that, bar *IdealTStamp* in Figure 6 corresponds to an ideal system with unlimited-sized timestamps and no additional overhead. Overall, the figure shows that *IdealTStamp* offers no performance advantage. The reason is that, in *TLS2*, running out of timestamps is relatively rare. In addition, *IdealTStamp* may perform worse than *TLS2* because restarts are occasionally beneficial to performance.

7.3.3 Supporting Advanced Task Scheduling

TLS2 uses a simple, two-priority algorithm to schedule tasks to CPUs (Section 4.3). In bar *AdvSched* of Figure 6, we change it to support the advanced scheduling outlined in Section 4.3: tasks are

strictly prioritized and preempt each other based on how speculative they are. Moreover, we set the scheduling overhead to zero.

We see that *AdvSched* delivers no advantage. The main reason is that, in *TLS2*, MergeNext regulates the number of ready tasks effectively. As a result, starvation of tasks by more speculative ones occurs with tolerable frequency.

8 Related Work

There are three pieces of work on environments that need out-of-order spawning.

Hammond *et al.* [8] have proposed a TLS CMP that supports out-of-order spawn with both subroutine and loop-iteration tasks. Their scheme is very different than ours. Each processor has a co-processor that controls TLS mechanisms by running software handlers. There are 2 broadcast busses. Co-processors are informed of what task is running on what processor. Messages are snooped from the broadcast busses and, based on their source, the co-processors can tell the relative ordering. Since caches contain state from a single task, no task ID is necessary. Squash signals are also broadcast. Commits require access to a centralized software data structure in shared memory. The most speculative task is killed if there is no space in the CMP. Overall, this is a broadcast-based, relatively centralized architecture. Moreover, the authors conclude that their scheme has too much control software overhead to support subroutine tasks. Their finding motivates our work.

There are several high-level performance-evaluation studies of environments that need out-of-order spawning [12, 13, 21, 22]. They often assume some ideal architectural feature, such as an infinite number of processors or perfect value prediction, and compare the performance to more realistic environments. Of those, [12, 13] examine a variety of sources of parallelism, including iterations from multiple loop levels, any-nesting subroutines, and full loops. [21, 22] examine subroutine-level nested parallelism in detail. None of these papers has attempted to describe the design of micro-architectural structures to support the tasks used. Consequently, they have not addressed the problems we cover. Our paper is the first detailed microarchitectural design of high-speed out-of-order tasking on a CMP and its evaluation. Many of the problems we solve do not even appear in these previous studies (e.g. task ID limitations or fast access to immediate successor).

DMT is a centralized, SMT-like processor whose hardware can extract out-of-order tasks from unmodified binaries [1]. The design is very different than ours because, being an SMT, it uses centralized structures that are *unusable* in a CMP. Specifically, DMT has a centralized hardware tree that records which tasks are successors of which. To determine the order of two tasks, the hardware walks the tree when: (1) there is a collision in the centralized LD/ST queue, or (ii) a task commits and needs to verify the register predictions for successor tasks. This centralization means that DMT does not need our proposed IS list and timestamp intervals. DMT kills the most speculative task if there is no space in the processor. However, it requires no analysis of matching task-end and spawn instruction like us because the binary is unmodified. Moreover, DMT does not support MergeNext, which is our new way of dynamically managing the resources in the system.

9 Conclusion

For CMPs with TLS to deliver on their promise, they must support dynamic environments where tasks are spawned out-of-order and unpredictably. Given the decentralized hardware of a CMP, this is challenging. This paper has been the first one to identify and design micro-architectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. We proposed three simple primitives for correct and fast task ordering and resource allocation. Task ordering is enabled with Splitting Timestamp Intervals for low-overhead order management, and with the Immediate Successor List for efficient task squash and commit. Fast and efficient resource allocation is enabled with Dynamic Task Merging, which directs speculative parallelism to the most beneficial code sections. To evaluate these primitives, we developed a TLS compiler with out-of-order spawn. With our mechanisms, a CMP with 2 4-issue processors increases the average speedup of SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (out-of-order spawn). Moreover, the resulting CMP outperforms an 8-issue superscalar: with the same clock frequency, the CMP delivers an average speedup of 1.14 over the 8-issue processor.

Overall, with our micro-architectural mechanisms, a CMP can leverage more sources of parallelism and boost TLS speedups. Note that, for applications that can exploit more PEs in the CMP (such as numerical applications), our technology is more enabling. The reason is that the codes can use our new hardware better. Finally, we expect that, as we improve our compiler algorithms, TLS speedups will improve.

References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, December 1998.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[3] M Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[4] 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.

[5] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. In *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 25–36, June 2003.

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

[7] 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.

[8] 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.

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

[10] X. F. Li, Z. H. Dui, Q. Y. Zhao, and T. F. Ngai. Software Value Prediction for Speculative Parallel Threaded Computations. In *First Value Prediction Workshop*, pages 18–25, June 2003.

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

[12] P. Marcuello and A. Gonzalez. A Quantitative Assessment of Thread-level Speculation Techniques. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 595–604, 2000.

[13] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] J. Y. Tsai, Z. Jiang, and P. C. Yew. Compiler Techniques for the Superthreaded Architecture. In *International Journal of Parallel Programming*, pages 27(1):1–19, 1999.

[20] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[21] F. Warg and P.Stenstrom. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[22] F. Warg and P. Stenström. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[23] 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.

[24] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 65–77, November 2002.