# POSH: A TLS Compiler that Exploits Program Structure *

Wei Liu James Tuck Luis Ceze Wonsun Ahn Karin Strauss Jose Renau† Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
{liuwei,jtuck,luisceze,dahn2,kstrauss,torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

†Computer Engineering Department
University of California, Santa Cruz
renau@soe.ucsc.edu
http://masc.soe.ucsc.edu

## Abstract

As multi-core architectures with Thread-Level Speculation (TLS) are becoming better understood, it is important to focus on TLS compilation. TLS compilers are interesting in that, while they do not need to fully prove the independence of concurrent tasks, they make choices of where and when to generate speculative tasks that are crucial to overall TLS performance.

This paper presents POSH, a new, fully automated TLS compiler built on top of gcc. POSH is based on two design decisions. First, to partition the code into tasks, it leverages the code structures created by the programmer, namely subroutines and loops. Second, it uses a simple profiling pass to discard ineffective tasks. With the code generated by POSH, a simulated TLS chip multiprocessor with 4 superscalar cores delivers an average speedup of 1.30 for the SPECint 2000 applications. Moreover, an estimated 26% of this speedup is a result of the implicit data prefetching provided by squashed tasks.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; C.1.4 [*Processor Architectures*]: Parallel Architectures

***General Terms*** Algorithms, Design, Measurement, Performance

***Keywords*** Thread-level speculation, TLS compiler, profiling, multi-core architecture, prefetching

## 1. Introduction

Although parallelizing compilers have made significant advances, they still fail to parallelize many codes. Examples of hard-to-parallelize codes are those with accesses through pointers or subscripted subscripts, possible interprocedural dependences, or input-dependent access patterns.

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

*PPoPP'06* March 29–31, 2006, New York, New York, USA.
Copyright © 2006 ACM 1-59593-189-9/06/0003. . . $5.00.

One potential way to execute these codes in parallel is to use multi-core architectures with Thread-Level Speculation (TLS) (e.g., [6, 8, 16, 17, 18, 19]). The general approach is to build tasks from the code, and speculatively run them in parallel, hoping not to violate sequential semantics. As tasks execute, special architectural support checks that no cross-task dependence is violated. If any is, the offending tasks are automatically squashed, the polluted state is repaired, and the tasks are re-executed.

Key to the acceptance of TLS architectures is the development of TLS compilers. Such compilers are unique in that they do not need to fully prove the absence of dependences across concurrent tasks — the hardware will ultimately guarantee it. However, their choices on how to break the code into speculative tasks and when to spawn them have a crucial impact on the performance of the resulting TLS system.

There are several instances of TLS compiler infrastructure in the literature (e.g., [1, 3, 4, 7, 13, 20, 21, 24]). In some of these compilers, tasks are built exclusively out of loop iterations [4, 24]. The reason is that loops are often the best source of parallelism. In other compilers [3, 7, 21], a dependence analysis pass identifies the most likely data dependences in the code and partitions the code into tasks to minimize cross-task dependences. In general, identifying likely dependences, often interprocedurally, is hard in irregular codes.

In this paper we present POSH, a new, fully automated TLS compiler infrastructure that we have developed. POSH adds several TLS passes to gcc-3.5, which is an early version of the latest gcc-4.0. These TLS passes operate on a static single assignment (SSA) tree used as the high-level intermediate representation in gcc [11]. Building on gcc allows us to leverage a complete compiler infrastructure and makes POSH very portable.

In the design of POSH, we have made two main design decisions. First, to partition the code into tasks, we rely on the code structures created by the programmer, namely subroutines and loops. This decision simplifies the TLS algorithms significantly. The second design decision is to add a simple profiling pass that takes into account both the parallelism and the data prefetching effects provided by the speculative tasks. The profiling pass prunes some tasks if it estimates that they are not beneficial. This profiling pass is invoked with a small input data set.

To enhance parallelism and data prefetching, POSH performs aggressive hoisting of task spawns. Moreover, it supports software value prediction. Finally, to maximize applicability, POSH targets a Chip Multiprocessor (CMP) architecture with relatively simple

TLS hardware. In particular, it assumes that processors can only communicate through shared memory.

Overall, the contributions of this paper are as follows:

- We present POSH, a complete TLS compiler infrastructure. Two main characteristics of POSH are that it leverages the code structure (loop iterations and subroutines of any nesting level) to generate tasks, and that it uses a profiling pass to discard ineffective tasks.
- We show that, through speculative parallelization, POSH can significantly speed-up applications that are very hard to analyze. Specifically, whole SPECint 2000 applications running on a simulated TLS CMP with 4 superscalar cores are sped up by 1.30 on average.
- We perform a detailed characterization of speedup sources and task behavior. We find that, for best performance, both subroutine and loop iteration parallelism should be exploited. Moreover, an estimated 26% of the TLS speedup is a result of the implicit data prefetching provided by squashed tasks. Finally, both task profiling and value prediction contribute to the speedups.

This paper is organized as follows. Section 2 gives some background on TLS; Section 3 gives an overview of POSH; Section 4 describes the main algorithms and design issues in POSH; Section 5 and Section 6 evaluate POSH; Section 7 discusses related work, and Section 8 concludes.

## 2. Background on TLS

A TLS compiler breaks a hard-to-analyze sequential code into tasks, and speculatively executes them in parallel, hoping not to violate sequential semantics (e.g., [1, 3, 4, 7, 13, 20, 21, 24]). The control flow of the sequential code imposes a control dependence relation between the tasks. This relation establishes an order of the tasks, and we can use the terms predecessor and successor to express this order. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is *speculative* when it may perform or may have performed operations that violate data or control dependences with its predecessor tasks. When a non-speculative task finishes execution, it is ready to *commit*. The role of commit is to inform the rest of the system that the data generated by the task are now part of the safe, non-speculative program state. Among other operations, committing always involves passing the non-speculative status to a successor task. Tasks must commit in strict order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware support that tracks, for each individual task, the data written and the data read without first writing it. A data dependence violation is flagged when a task modifies a datum that may have been loaded earlier by a successor task. At this point, the consumer task is *squashed* and all the state that it has produced is discarded. Its successor tasks are also squashed. Then, the task is re-executed.

TLS architectures can discard the state produced by a speculative task and re-start the task thanks to special hardware that buffers all speculative modifications, and a checkpointing mechanism that enables rollback. Discussion of such hardware (e.g. [6, 8, 16, 17, 18, 19]) is beyond this paper's scope. Note that, thanks to these buffers, anti and output dependences across tasks do not cause squashes.

## 3. Overview of POSH

The POSH framework is composed of two parts closely tied together: a compiler and a profiler (Figure 1). The compiler performs task selection, inserts task spawn points, and generates the code. The profiler is a simple software module that provides feedback to the compiler to improve task selection.

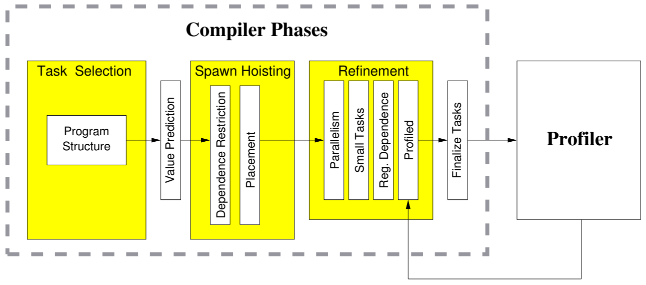


**Figure 1.** Structure of the POSH framework.

### 3.1 Target TLS Hardware Assumptions

POSH makes several assumptions on the target TLS hardware, including how live-ins are passed to tasks, how dependences are enforced between tasks, and how tasks are created and terminated. First, POSH does not assume any special hardware support to transfer registers between tasks — all live-ins to a task must be passed through memory. This model corresponds to a standard CMP, where the different cores only communicate through memory. Consequently, it is the responsibility of POSH to guarantee that any value in a register that may be needed by any successor task is written to memory. Second, POSH assumes that the hardware will detect data dependence violations through memory, and will squash and restart tasks accordingly, as is the norm in most TLS architectures proposed.

Finally, POSH assumes an ISA with *spawn* and *commit* instructions to initiate and to successfully complete a task, respectively. The spawn instruction takes as an argument the address of the first instruction in the child task. Execution of the spawn instruction initiates the child task in an idle processor. Execution of the commit instruction indicates to the hardware that the task has completed its work. It is the job of the compiler to insert the spawn and commit instructions.

### 3.2 Compiler Phases

There are three main compiler phases: *Task Selection*, *Spawn Hoisting*, and *Task Refinement* (Figure 1). In the task selection phase, POSH identifies as tasks many subroutines, subroutine continuations, loop iterations, and loop continuations. By subroutine or loop continuation, we mean the code that follows the subroutine or loop, respectively. All these programmer-generated structures are taken as hints to delineate code sections with a relatively independent and sizable amount of work. For each task, POSH identifies the instruction where it begins (*begin point*). Because the begin point of one task is the *end point* of another, POSH then adds commit instructions right before each begin point. The output of the task selection phase is a set of begin points.

Immediately after task selection, POSH invokes the *Value Prediction* pass (Figure 1). This pass predicts the values of certain kinds of variables that cross task boundaries, hoping to reduce the number of data dependence violations. Specifically, POSH predicts the values of function return variables and variables that are or behave like loop induction variables.

In the spawn hoisting phase, POSH inserts task spawn instructions at the begin points of all tasks, creating what we call *spawn points*. Then, POSH considers each of the spawn instructions and tries to hoist them as much as possible in the intermediate representation of the program. The goal of hoisting the spawn points is to enhance parallelism and prefetching as much as possible. There are, however, three constraints on how far can POSH hoist a spawn instruction. These constraints are discussed in Section 4.2 and are represented in the figure as the *Dependence Restriction* box.

In the task refinement phase, POSH makes the final decisions on which tasks will make it into the final binary. This phase is composed of a number of passes, whose goal is to improve the quality of the final set of tasks chosen for execution. From the perspective of the compiler, the profiler is part of this task refinement process.

The refinement phase includes the *Parallelism*, *Small Tasks*, *Register Dependence* and *Profiled* steps. The first three steps eliminate tasks that have certain characteristics, namely they are not spawned farther than some threshold number of instructions from their begin point, are smaller than certain threshold static task size, or have too many live-ins, respectively. The last step (*Profiled*) accepts input from the profiler and uses it to eliminate a final set of tasks.

In the *Finalize-Tasks* step, the compiler inserts all instructions needed to correctly spawn, execute, and commit tasks, as well as to perform value prediction. The final code generation varies depending on whether we plan to profile or not. If we do, then extra information (e.g., task id) is encoded into each task to allow the profiler to communicate back to the compiler.

We built these phases as a part of gcc-3.5, which was later merged into gcc-4.0. This allows us to leverage a complete compiler infrastructure. We perform our transformations in the tree SSA high-level intermediate representation [11].

### 3.3 Profiler

The profiler provides a list of tasks that are beneficial for performance. The compiler uses this information to eliminate the non-beneficial tasks. The profiler also informs the compiler of the effectiveness of value prediction.

When the profiler runs, it collects information about each task. The information can be used to estimate the amount of parallelism the task enables, the likelihood that the task is squashed, and whether the task may offer benefits due to prefetching. A more detailed explanation of the profiler algorithms is given in Section 4.4. On average, a profiler run takes about 5 minutes on an Intel 3GHz desktop.

## 4. Algorithms and Design Issues

### 4.1 Task Selection

Task selection is easier for TLS compilers than for conventional parallelizing compilers. The reason is that dependences are allowed to remain across tasks, since the hardware ultimately guarantees correct execution. In practice, a variety of heuristics can be used to choose tasks. The resulting tasks should ideally have no cross-task dependences, enough work to overcome overheads, and few live-ins. Choosing tasks that provide the optimal performance improvement is NP-hard [1].

POSH's heuristic to select tasks is to rely on the structure that the programmer gave to the code. Specifically, POSH uses the following modules as potential tasks: (i) subroutines at any nesting level, (ii) their continuations, (iii) loop iterations at any nesting level, and (iv) loop continuations. We refer to the first two types of tasks as "subroutines" and the last two types as "loop iterations".

As an example, Figure 2 shows how POSH generates tasks out of a subroutine and its continuation (Chart (a)), or out of loop iterations (Chart (c)). Chart (a) shows a code segment with a call to subroutine *S1*. POSH identifies two tasks: the call to *S1* and its continuation code (the code that follows the call). Consequently, it inserts begin points *BP2* and *BP1*, respectively.

Chart (c) shows a loop as it is typically represented in the intermediate representation of gcc-3.5. The representation typically places the update of the induction variable (*i* in the example) right before the backward jump. POSH identifies loops in the program by computing the set of strongly connected nodes in the control flow graph. Then, it tries to identify the update to the induction variable, and it places the task begin point for iteration *n* (*BP* in the figure), right before the update of the induction variable in iteration *n-1*. With this approach, induction variables neither need to be predicted nor cause dependence violations. In the cases where gcc-3.5 does not follow this pattern, POSH does predict the values of induction variables.

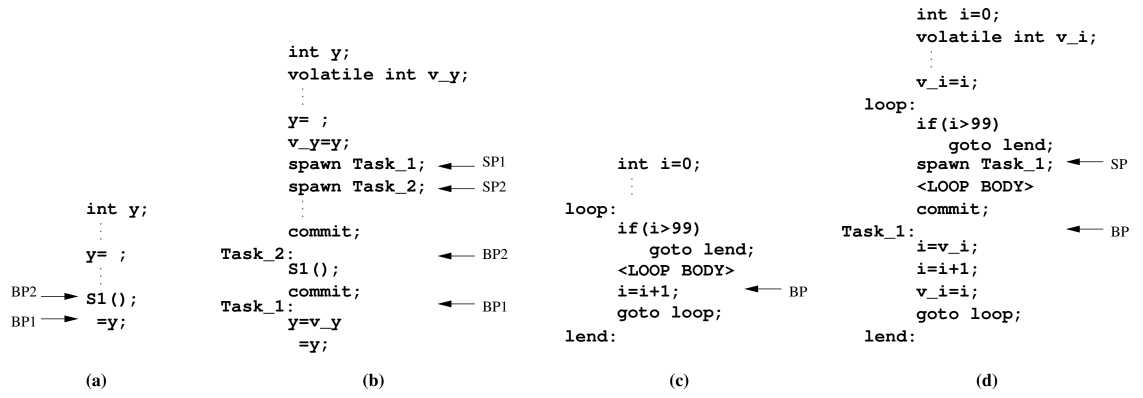
### 4.2 Spawn Hoisting

In the spawn hoisting phase, POSH places a spawn instruction at the begin point of every task, and then tries to hoist it as much as possible in the intermediate representation of the program. The amount of hoisting is limited by three constraints. First, the spawn instruction should not be before the definition of any variable used or very likely used in the task — except if value prediction is used. Second, the spawn should be in a location that is execution equivalent to the start of the task[1]. This constraint ensures that a task is spawned if and only if it needs to be executed.

Finally, if a task spawns multiple children tasks, the spawn instructions should be placed so that children tasks are spawned in strict reverse sequential order. This means that the child task that is latest in sequential order should be spawned first, and similarly for the other children tasks. As explained in [15], the reason for this convention is to simplify the CMP microarchitecture, while still enabling a high degree of task spawn flexibility and high performance.

Figure 2(b) shows the code from Chart (a) after transformation. The continuation task in Chart (a) (the one starting at *BP1*) has the live-in variable *y*. Consequently, we need to ensure that *y* is written to memory before the continuation task is invoked, and that *y* is read from memory inside the continuation task. POSH ensures this by declaring a volatile variable *v_y* (Chart (b)). Updates to such a

[1] We say that location $l_1$ in the control flow graph is execution equivalent to location $l_2$ of the start of the task when the instruction in $l_2$ executes if and only if the instruction in $l_1$ has executed, and both instructions are executed the same number of times and in an interleaved manner.



```
int y;
  :
y= ;
  :
S1();      <- BP2
 =y;       <- BP1
```
(a)

```
         int y;
         volatile int v_y;
           :
         y= ;
         v_y=y;
         spawn Task_1;   <- SP1
         spawn Task_2;   <- SP2
           :
         commit;
Task_2:                  <- BP2
         S1();
         commit;
Task_1:                  <- BP1
         y=v_y
          =y;
```
(b)

```
     int i=0;
       :
loop:
     if(i>99)
        goto lend;
     <LOOP BODY>
     i=i+1;          <- BP
     goto loop;
lend:
```
(c)

```
           int i=0;
           volatile int v_i;
             :
           v_i=i;
     loop:
           if(i>99)
              goto lend;
           spawn Task_1;   <- SP
           <LOOP BODY>
           commit;
   Task_1:                 <- BP
           i=v_i;
           i=i+1;
           v_i=i;
           goto loop;
     lend:
```
(d)

**Figure 2.** Generating tasks out of a subroutine and its continuation, and out of loop iterations. In Charts (c) and (d), the loop body is assumed to contain potential cross-iteration dependences.

variable will always be propagated to memory. Then, before the continuation task is spawned, POSH copies *y* to *v_y*. Inside the continuation task, *v_y* is read from memory and copied to *y*.

As Chart (b) shows, the spawn for the continuation task is hoisted all the way up to after the update to *v_y* (spawn point *SP1*). It cannot be hoisted further due to the first constraint above. The spawn for the subroutine task (*Task_2*) is hoisted up to right after spawn point *SP1*. POSH does not hoist it earlier because of the third constraint described above: since *Task_2* is earlier than *Task_1* in sequential order and both tasks are spawned from the same task, *Task_2* should be spawned after *Task_1* is spawned. A valid alternative but possibly worse for parallelism is for *Task_2* to spawn earlier and for *Task_1* to spawn from *Task_2*.

Figure 2(b) also includes the commit statements for the tasks. Recall that a commit statement is placed at the end of each task, which is right before the begin point of the next task.

Finally, Figure 2(d) shows the code from Chart (c) after transformation. As in Chart (b), POSH introduces a volatile variable to ensure that variable *i* is written to memory at every iteration and read from memory by the successor iteration. Note that the spawn for *Task_1* can be hoisted only up to the beginning of the loop body because of the execution equivalence constraint. POSH also inserts the commit statement.

## 4.3 Prefetching Effects

The speedup of TLS comes from two effects: task parallelism and data prefetching. Figure 3 shows how two TLS tasks, *Task 1* and *Task 2* (Chart (a)) can benefit from parallelism and prefetching. TLS benefits from parallelism when the two tasks run concurrently (Chart (b)). TLS benefits from data prefetching when a task suffers a cache miss on datum *A*, the task is then squashed, and later a second task that will not be squashed obtains *A* from the cache. This second task can be the re-execution of the squashed task or a different task.

Figure 3(c) illustrates this effect when the task that benefits from prefetching is the one that was squashed. In its first execution, *Task 2* suffers a miss on variable *A*. Later, due to a dependence violation, *Task 2* is squashed and restarted. In the re-execution, when *Task 2* accesses *A* again, it finds the data already in the cache. Consequently, while there is little parallelism between *Task 1* and

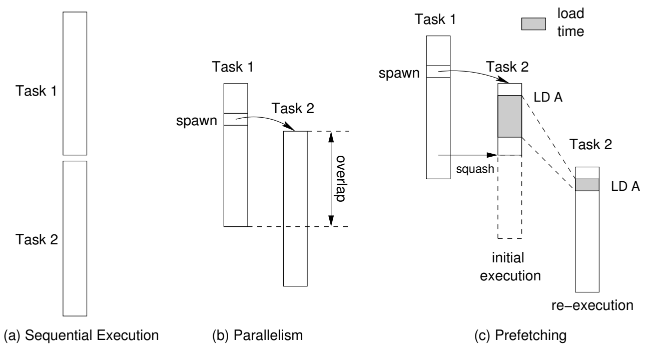


**Figure 3.** The two potential benefits of TLS: parallelism and prefetching.

*Task 2* in Figure 3(c), TLS speeds up the program because *Task 2* benefits from automatic data prefetching.

Figure 4 shows a code snippet from the SPECint 2000 *gap* application that illustrates prefetching. The while loop has loop-carried dependences in *hdP*, *hdL*, and *i*. Consequently, existing TLS compilers are unlikely to parallelize this loop. However, parallelizing this loop yields significant performance gains due to prefetching. Specifically, *ProdInt()* calculates the product of two integer numbers. The numbers are stored in memory in a tree data structure. As a result, *ProdInt()* has poor locality and suffers many L2 misses. Fortunately, the squashed tasks bring in lines into the cache that are very likely to be needed later.

```
i = HD_TO_INT(hdR);
while ( i != 0 ) {
  if ( i % 2 == 1 )  hdP = ProdInt( hdP, hdL );
  if ( i     >  1 )  hdL = ProdInt( hdL, hdL );
  i = i / 2;
}
```

**Figure 4.** Code snippet from the SPECint 2000 *gap* application that illustrates prefetching.

POSH exploits prefetching implicitly. In addition, the profiler tries to expose its effect. We examine the profiler next.

## 4.4 Profiler

The profiler runs the applications with the *Train* input set. The execution of the tasks is *serial*, does not assume any TLS architectural support, and models only some rudimentary timing. In addition, the profiling analysis is not tied to any number of processors; instead, it assumes that an unlimited number of processor cores will be available.

With so few constraints, the profiling framework is widely usable in a variety of circumstances. The profiler also models a simple cache (without modeling time) to estimate the number of L2 cache misses. The latter are used for prefetching analysis. Simulating a cache without modeling time introduces only a small overhead. Overall, an average profiler run takes about 5 minutes on a desktop.

### 4.4.1 Profiler Execution

The profiler assumes that every instruction executed takes $C_I$ cycles, except for loads and stores that miss in the L2 cache, which take $C_{L2Miss}$ cycles. It also assumes some constant overhead for squashing a task and its successors and restarting the task ($Ovhd_{squash}$), and for spawning a task ($Ovhd_{spawn}$). With all this information, the profiler can build a rudimentary model of the TLS execution.

Let us consider an example (Figure 5-(a)). Although the profiler executes the code sequentially, it assigns a time to each instruction as if the tasks were executed in parallel. Specifically, when the profiler executes the first instruction of *Task 2*, it rewinds the time back to when the task would be spawned ($T_1$) plus the spawn overhead ($Ovhd_{spawn}$).

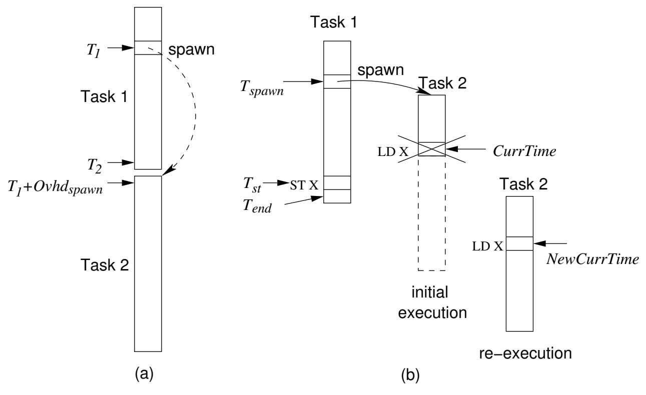


**Figure 5.** Example of profiler models.

For each spawn instruction, the profiler records the time and the target task. For each store, it records the time and the address stored to. When the profiler encounters a load to an address, it checks the table of recorded stores to find the latest store that wrote to that address. If the time of the load is less than the time of the store, the profiler has detected a potential dependence violation. At this point, the profiler conceptually squashes the consumer task and updates the times of its instructions.

An example is shown in Figure 5-(b). In the figure, the profiler executed the $STX$ in *Task 1* and assigned time $T_{st}$ to it. Later, the profiler encounters the $LDX$ in *Task 2* at a time that we call $CurrTime$. Since $CurrTime < T_{st}$, it means that *Task 2* needs to be squashed. As a result, the profiler updates the times of all instructions in *Task 2*. In particular, the new $LDX$ time is $NewCurrTime$:

$$\begin{aligned} NewCurrTime =& T_{st} + Ovhd_{squash} \\ & + CurrTime - T_{spawn} - Ovhd_{spawn} \\ & - N_{L2Miss} \times (C_{L2Miss} - C_I) \end{aligned}$$

In this formula, $T_{spawn}$ is the time associated with the initial spawn of *Task 2*, and $N_{L2Miss}$ is the number of L2 misses suffered by the first execution of *Task 2* until it reached $LDX$. With this method, the profiler models the squash and re-execution with a single sequential run.

### 4.4.2 Benefit of a Squashed Task

Based on the previous discussion, the profiler estimates the expected performance benefit of a squashed task. The benefit is a combination of (i) any task overlap remaining after re-execution, and (ii) prefetching effects, as follows:

$$\begin{aligned} Benefit =& Overlap + Prefetch \\ =& (T_{end} - T_{st} - Ovhd_{squash}) \\ & + (C_{L2Miss} - C_I) \times M_{L2Miss} \end{aligned}$$

In the formula, $T_{end}$ and $T_{st}$ are the times when *Task 1* finishes and executes *ST X*, respectively (Figure 5-(b)). $M_{L2Miss}$ is the number of misses suffered by the first execution of *Task 2* until *Task 1* executed *ST X* and squashed *Task 2*. This simple model assumes that all these missing data will be re-accessed by *Task 2* or another non-squashed task.

### 4.4.3 Task Elimination

The profiler uses three criteria to identify the tasks that need to be eliminated: task size, hoisting distance, and squash frequency. First, due to the overhead of task spawning, small tasks are unlikely to provide much benefit. Consequently, we eliminate a task if its size is smaller than threshold $Th_{sz}$ and it spawns no other task. We treat small tasks that spawn other tasks with care. The reason is that if such small tasks can be hoisted significantly (see next criteria), their callees would benefit substantially.

Second, we examine the hoisting distance, namely the number of instructions between the spawn point of a task and the begin point of that task. Short hoisting distances do not expose much overlap between tasks, while long hoisting distances are likely to introduce too many data dependences. Consequently, we eliminate the tasks that have a hoisting distance smaller than $Th_{min\_hd}$ or larger than $Th_{max\_hd}$. This dynamic algorithm complements one of our compiler steps that eliminated tasks that had small *static* hoisting distances.

Finally, task squashes are very expensive. Consequently, we eliminate tasks with an average number of squashes per task commit that is higher than a squash threshold $Th_{sq}$. However, based on Section 4.4.2, we know that some squashes may result in a net positive performance effect due to partial overlap or prefetching. Consequently, we apply a correction to this rule. Specifically, if a task to be eliminated due to squashes has a performance benefit (*Benefit* as defined in Section 4.4.2) higher than a squash benefit threshold $Th_{sb}$, the task is not eliminated.

### 4.5 Software Value Predictor

There are some specific locations in the code where value prediction has been shown profitable in previous studies (e.g., [9, 12, 22]). In POSH, we use value prediction in three cases: some function return variables, loop induction variables, and some variables in loops that have a behavior similar to induction variables. For these cases, POSH uses a software value prediction scheme similar to the one in [9]. Such scheme leverages the TLS dependence tracking hardware to squash a task that used a wrong prediction.

## 5. Methodology

### 5.1 Simulated Architecture

Since there is no hardware platform that supports TLS, we target POSH to SESC, a cycle-accurate execution-driven simulator [14]. The simulator models out-of-order superscalar processors and memory subsystems in detail. The TLS architecture modeled is shown in Table 1. It is a four-processor CMP with TLS support. Each processor is a 3-issue core and has a private L1 cache that buffers the speculative data. 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 speculative L1 caches similar to [15]. There is no special hardware for communicating register values between cores.

| | | | |
|---|---|---|---|
| Frequency | 4 GHz | ROB | 132 |
| Fetch width | 8 | I-window | 68 |
| Issue width | 3 | LD/ST queue | 48/42 |
| Retire width | 3 | Mem/Int/Fp unit | 1/2/1 |
| Branch predictor: | | Spawn Overhead | 12 cycles |
| Mispred. Penalty | 14 cycles | Squash Overhead | 20 cycles |
| BTB | 2K, 2-way | | |
| L1 Cache: | | L2 Cache: | |
| Size, assoc, line | 16KB, 4, 64B | Size, assoc, line | 1MB, 8, 64B |
| Latency | 3 cycles | Latency | 12 cycles |
| | | Memory: | |
| Lat. to remote L1 | at least 8 cycles | Latency | 500 cycles |
| | | Bandwidth | 10GB/s |

**Table 1.** Architecture simulated. All cycle counts are in processor cycles.

In our evaluation, we report the speedups of this TLS CMP architecture over the sequential execution of the *original* application binaries running on a single-processor non-TLS architecture. The non-TLS architecture has one 3-issue core, one L1 cache, and one L2 cache like those in Table 1.

### 5.2 Profiler Parameters

Table 2 shows the parameters used to configure the profiler. We assume 1 cycle per instruction and a 200-cycle execution for an instruction that misses in L2. The latter is lower than the time to get to memory because the architecture we model is an out-of-order processor that can hide some of the latency by executing independent instructions.

| | | | |
|---|---|---|---|
| $C_I$ | 1 cycle | $Th_{sz}$ | 30 instructions |
| $C_{L2Miss}$ | 200 cycles | $Th_{min\_hd}$ | 150 instructions |
| | | $Th_{max\_hd}$ | 5M instructions |
| $Ovhd_{spawn}$ | 12 cycles | $Th_{sq}$ | 0.55 |
| $Ovhd_{squash}$ | 20 cycles | $Th_{sb}$ | 0 |

**Table 2.** Profiler parameters.

In the rightmost column of Table 2, we show the threshold values used in our profiler. $Th_{sz}$ is set to 30 to prevent selecting very small tasks. The minimum and maximum spawn distance thresholds, $Th_{min\_hd}$ and $Th_{max\_hd}$, respectively, are set to conservative values. The squash threshold $Th_{sq}$ is set to 0.55, which means that a task squashed more than about once out of 2 commits will typically be eliminated. Finally, we set $Th_{sb} = 0$, so that no task is eliminated if there is any benefit at all from squashing.

### 5.3 Applications Evaluated

We run the SPECint 2000 applications using the *Ref* data set. The profiler uses the *Train* data set. All of the SPECint 2000 codes are included except three that fail our compilation pass (*gcc*, *perlbmk*, and *eon* — the latter because C++ is not currently supported).

The baseline binaries for sequential execution have no TLS or any other additional instructions. For the TLS binaries, POSH rearranges the code into tasks and adds extra instructions for spawn, commit, passing live-ins through memory, and value prediction.

In both the TLS and non-TLS compilations, we first run SGI's source-to-source optimizer (copt from MIPSPro) on the SPECint code. This pass performs partial redundancy elimination, loop unrolling, inlining, and other optimizations.

To accurately compare the performance of the different binaries, we cannot simply time a fixed number of instructions. The reason is that TLS binaries have more instructions. Instead, "simulation markers" are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization (typically over 1 billion instructions), a certain number of markers are executed, so that the baseline binary graduates from 500 million to 1 billion instructions.

## 6. Evaluation

To evaluate POSH, we examine several issues: task selection, static and dynamic task characteristics, memory behavior, prefetching, and effectiveness of the profiler and value prediction.

### 6.1 Impact of Task Selection

To evaluate the performance impact of selecting tasks based on different types of code structure, we conduct three experiments. In the experiments, we select as tasks: (1) only subroutines and subroutine continuations (*Subr*), (2) only loop iterations and loop continuations (*Loop*), or (3) all such tasks (*Subr+Loop*). Figure 6.1 shows the speedup obtained by these three selection algorithms over the sequential execution.

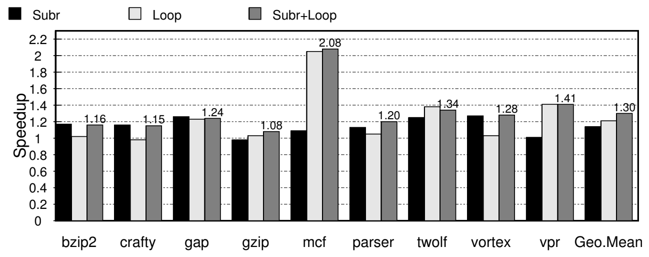


**Figure 6.** Speedup of the TLS execution over the sequential one for different task selection algorithms.

As shown in the figure, the best speedups are obtained when both types of tasks are selected (*Subr+Loop*). With this task selec-

| App. | # of *Subr* Tasks | # of Loops with *Loop* Tasks | # of Register Live-Ins | | | # of Register Live-Outs | | |
|---|---|---|---|---|---|---|---|---|
| | | | All | *Subr* | *Loop* | All | *Subr* | *Loop* |
| bzip2 | 6 | 11 | 3.1 | 3.0 | 3.1 | 4.3 | 5.6 | 4.1 |
| crafty | 38 | 4 | 6.6 | 6.6 | 8.0 | 8.0 | 8.0 | 9.0 |
| gap | 18 | 3 | 5.1 | 3.2 | 6.5 | 5.3 | 2.8 | 7.0 |
| gzip | 11 | 4 | 2.8 | 3.3 | 2.1 | 4.1 | 4.7 | 3.0 |
| mcf | 2 | 2 | 3.4 | 6.0 | 3.4 | 4.3 | 5.5 | 4.3 |
| parser | 95 | 35 | 6.7 | 9.3 | 1.6 | 8.4 | 11.7 | 2.2 |
| twolf | 15 | 4 | 3.8 | 5.0 | 2.6 | 3.8 | 5.0 | 2.6 |
| vortex | 105 | 1 | 7.0 | 7.0 | 9.0 | 5.1 | 5.1 | 9.0 |
| vpr | 0 | 2 | 12.1 | 0.0 | 12.1 | 13.8 | 0.0 | 13.8 |
| Average | 32.2 | 7.3 | 5.7 | 4.8 | 5.4 | 6.3 | 5.4 | 6.1 |

**Table 3.** Static task information.

| App. | # of Tasks (%) | | | # of Instructions (%) | | | Avg. Size (Insts) | | | Busy CPUs |
|---|---|---|---|---|---|---|---|---|---|---|
| | Success | Restart | Kill | Success | Restart | Kill | Success | Restart | Kill | |
| bzip2 | 56.7 | 14.5 | 28.8 | 93.3 | 3.8 | 2.9 | 965.8 | 153.4 | 59.1 | 1.35 |
| crafty | 47.9 | 22.6 | 29.5 | 77.4 | 8.5 | 14.1 | 1175.0 | 271.9 | 346.9 | 1.74 |
| gap | 19.1 | 29.4 | 51.5 | 59.9 | 27.7 | 12.4 | 553.8 | 166.2 | 42.3 | 2.08 |
| gzip | 32.5 | 21.9 | 45.6 | 52.8 | 18.9 | 28.3 | 202.6 | 107.3 | 77.3 | 2.14 |
| mcf | 77.1 | 13.4 | 9.5 | 84.1 | 13.0 | 2.9 | 68.1 | 60.1 | 19.1 | 2.94 |
| parser | 35.7 | 24.5 | 39.8 | 67.8 | 20.6 | 11.6 | 279.9 | 124.0 | 43.0 | 1.86 |
| twolf | 65.7 | 14.4 | 19.9 | 88.7 | 7.3 | 4.0 | 308.7 | 116.5 | 45.5 | 1.67 |
| vortex | 54.0 | 24.8 | 21.2 | 81.1 | 9.6 | 9.3 | 391.4 | 101.0 | 114.1 | 2.27 |
| vpr | 22.7 | 28.5 | 48.8 | 56.2 | 28.8 | 15.0 | 334.9 | 136.8 | 41.5 | 2.82 |
| Average | 45.7 | 21.6 | 32.7 | 73.5 | 15.3 | 11.2 | 475.6 | 137.5 | 87.7 | 2.10 |

**Table 4.** Dynamic task information.

tion algorithm, TLS delivers speedups that reach 2.08 in *mcf*, and have a geometric mean of 1.30. Of the other two algorithms, while some applications perform better with *Subr*, others perform better with *Loop*. Consequently, selecting only either *Subr* or *Loop* is not enough to get the best results. In the rest of the paper, we use both types of tasks.

These significant *Subr+Loop* speedups make POSH an attractive TLS compiler infrastructure, given that they are obtained in a fully automated manner for hard-to-analyze, pointer-based programs such as SPECint.

## 6.2 Task Characterization

### 6.2.1 Static Information

Table 3 gives static information on the tasks selected by POSH after all the passes, including the profiler. The second column shows the number of subroutine and subroutine continuation tasks, while the third column shows the number of loops whose iterations and continuations are given out as tasks. The average figures for these parameters are 32.2 and 7.3, respectively. Their relative value is not surprising, given that SPECint applications usually have many subroutine calls. *Vpr* is an interesting case, with only two static loops, yet yielding a speedup of 1.41 with *Loop* (Figure 6.1).

The last two groups of columns show the average number of register live-ins and live-outs, respectively, per task. These numbers are small. On average, a task has about 6 register live-ins and 6 register live-outs.

### 6.2.2 Dynamic Information

Figure 7 shows the cumulative distribution of the dynamic size of tasks that commit. The bulk of the committing tasks have 50-500 instructions and seldom have more than 4,000 instructions.

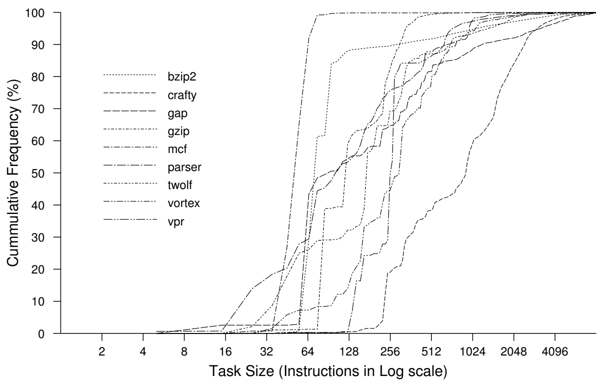


**Figure 7.** Distribution of the average size of dynamic tasks that commit, in number of instructions.

Table 4 shows information on the dynamic behavior of tasks. Tasks are grouped into those that successfully commit (*Success*), those that cause a dependence violation and have to be re-executed (*Restart*), and those that are successors of *Restart* tasks and, therefore, are killed (*Kill*). Columns 2-4 show the fraction of tasks of each type, while Columns 5-7 show the fraction of instructions belonging to each type of task, and Columns 8-10 show the average size of the tasks of each type.

The table shows that slightly over 50% of the dynamic tasks get restarted or killed. Since the restarted or killed tasks do not typically run to completion, they correspond to only around 26% of all the instructions executed. The table also shows that the average size of the tasks that commit is 476 instructions, while tasks that get restarted or killed are on average much smaller.

The last column of Table 4 shows the number of busy CPUs. On average, there are 2.10 busy CPUs. As shown in the previous columns, some of the work is wasted in restarted and killed tasks.

## 6.3 Memory System Access Characterization

To gain insight into the speedups delivered by TLS, we analyze the memory system accesses. Figure 8 breaks down the read requests according to where they are satisfied from. A read request can be satisfied by the processor's load/store queue (*Forwarded*), the local L1, a remote L1 (only for TLS execution), the L2 cache, and main memory. For each application, Figure 8 shows the number of read requests for the sequential (*S*) and TLS (*T*) executions. The bars are normalized to the sequential execution. In the TLS execution, only reads from committed tasks are considered.

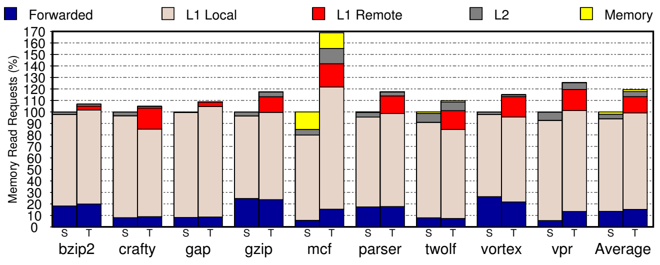


**Figure 8.** Distribution of where the reads are satisfied from. In the figure, *S* and *T* refer to sequential and TLS execution, respectively.

The figure shows that the TLS execution has more reads. On average, it has about 20% more reads. This is because the TLS binary code is less efficient, as the compiler has a hard time optimizing across task boundaries. Moreover, the TLS binary adds instructions to pass task live-ins through memory. The figure also shows that, in both sequential and TLS execution, most of the reads are satisfied by the L1. However, in TLS, some of these accesses are satisfied by a remote L1. Overall, TLS execution is less efficient in that it has more reads and, on average, they take longer to complete.

The most expensive reads in Figure 8 are those that go to main memory. Figure 9 considers such reads for the TLS execution. They are labeled *Committed* in the figure, and are normalized to the number of reads to main memory under sequential execution. The figure also stacks up the reads to main memory issued by squashed tasks. Such reads can bring useless data into the caches (*Wasted*), or can bring useful data that are reused by a committed task, either from the L2 cache or from the L1 cache (*Prefetch to L2* and *Prefetch to L1*, respectively). Interestingly, Figure 9 shows that the number of reads to main memory by squashed tasks that end up prefetching data into L2 or L1 is significant. On average, they account for slightly less than 20% of the reads to main memory. These reads largely represent the prefetching effect of squashed tasks.

Note that these measurements do not include the small prefetching effect of write misses to memory by squashed tasks. While a write miss brings data into both L1 and L2, only L1 is updated. Since the L2 line remains clean, when the task is squashed, the L2 line is not invalidated. Such line could later be reused by a committed task. In practice, however, write misses are much less frequent

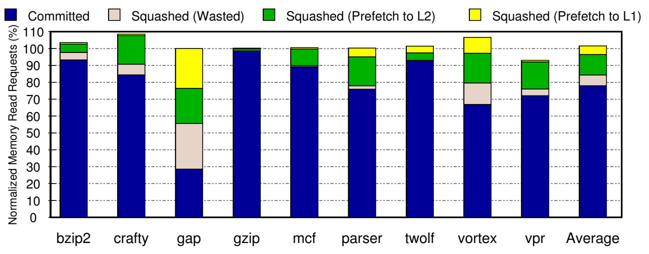


**Figure 9.** Total reads to main memory by committed and squashed tasks in TLS execution. The bars are normalized to the number of reads to main memory in sequential execution.

than read misses. Therefore, accounting for their prefetching effect into L2 would not change our analysis significantly.

## 6.4 Contribution of Prefetching to TLS Speedup

The prefetching effect of squashed tasks contributes to the speedup of TLS execution. To see its contribution, we simulated a TLS execution where the data brought in by the *Prefetch to L2* and *Prefetch to L1* reads of Figure 9 are marked as invalid. Consequently, they are not reused, and committed tasks have to re-request them from main memory. We call this execution *TLS_NoPrefetch*. Figure 10 compares the speedup of *TLS_NoPrefetch* and TLS over sequential execution. The difference between the two bars is the effect of data prefetching induced by TLS.

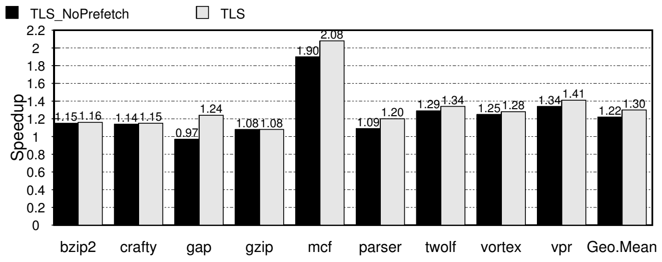


**Figure 10.** Speedup of *TLS_NoPrefetch* and TLS over sequential execution.

From the difference between the two bars, we see that most applications benefit from this type of prefetching. The application that benefits the most is *gap*. If we focus on the section of the geometric mean bars that is above 1, we see that *TLS_NoPrefetch* is about one quarter lower than TLS (i.e. $\frac{1.30-1.22}{1.30-1.00} = 0.26$). Consequently, we conclude that, roughly speaking, about one quarter of the TLS speedup comes from prefetching and the rest from parallelism.

## 6.5 Effectiveness of the Profiler

To assess the effectiveness of the profiler, we compare the TLS code generated by POSH with and without the profiling pass. Figure 11 shows the speedups of such codes over the sequential execution. The figure shows that, without the profiler, the TLS execution obtains a minor average speedup of 1.04. If we apply the profiling pass, we obtain the 1.30 average speedup already presented in Section 6.1.

Table 5 gives insight into why the profiling pass is necessary in POSH. The table shows the number of static tasks before and after

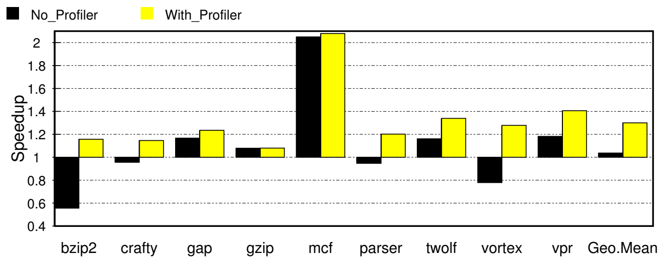


**Figure 11.** Speedup of TLS with and without the profiling pass over the sequential execution.

the profiling pass. Following Table 3, the figures in Table 5 add up the number of subroutine and subroutine continuation tasks, and the number of loops whose iterations and continuation are given out as tasks. From the table, we see that the profiler reduces the average number of such tasks from 198 to 39. Most of the tasks eliminated are small tasks, tasks that have little hoisting, and tasks that cause violations. In the current organization of POSH, we rely on the profiler to identify these tasks. For this reason, the profiling pass is needed for good performance.

| App. | #Tasks Before Profiling | #Tasks After Profiling |
|---|---|---|
| bzip2 | 120 | 17 |
| crafy | 424 | 42 |
| gap | 78 | 21 |
| gzip | 57 | 15 |
| mcf | 22 | 4 |
| parser | 587 | 130 |
| twolf | 75 | 19 |
| vortex | 396 | 106 |
| vpr | 26 | 2 |
| Average | 198.3 | 39.5 |

**Table 5.** Number of static tasks before and after the profiling pass.

### 6.6 Effectiveness of Value Prediction

Finally, we consider the effectiveness of our value prediction techniques. Figure 12 shows the speedup of TLS with and without value prediction over the sequential execution. On average, the applications run about 7% slower if POSH does not use value prediction. Consequently, we recommend its use.

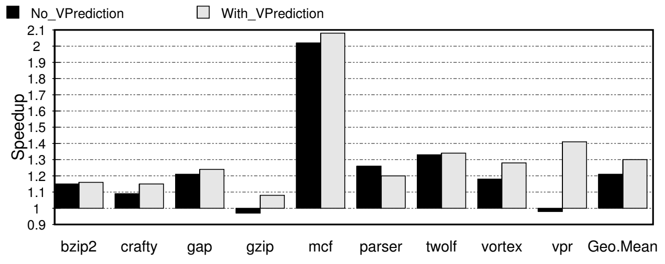


**Figure 12.** Speedup of TLS with and without value prediction over the sequential execution.

This average hides variations across applications. For example, *vpr* runs 31% slower without value prediction. According to Table 3, there are only two static loops selected to generate tasks in *vpr*. The induction variables of these two loops are highly predictable, and the loops show good parallelism. Prediction is needed in these two cases because the induction variable updates occur within an if-then-else statement — a solution like that in Figure 2(d) is not feasible.

On the other hand, some applications are hurt by value prediction. For example, *parser* runs about 5% faster without value prediction. The overhead of inserting extra instructions to support value prediction is not compensated by the performance gains in this application.

## 7. Related Work

Several compiler infrastructures for TLS have been proposed but differ significantly in their scope. The Multiscalar compiler [21] selects tasks by walking the Control Flow Graph (CFG) and accumulating basic blocks into tasks using a variety of heuristics. The task selection methodology for the Multiscalar compiler was recently revisited by Johnson *et al.* [7]. Instead of using a heuristic to collect basic blocks into tasks, the CFG is now annotated with weights and broken into tasks using a min-cut algorithm. These compilers assume special hardware for dispatching threads; they do not specify when a thread should be launched.

A number of compilers focus only on loops [4, 5, 20, 24]. In SPSM [5], loop iterations are selected by the compiler as speculative threads. An interesting part of the work is the use of the *fork* instruction, very similar to our spawn instruction, that allows the compiler to specify when tasks begin executing. In addition, SPSM recognized the potential benefits from prefetching but proposed no techniques to exploit it. Du *et al.* [4] present a cost-driven compilation framework to statically determine which loops in a program deserve speculative parallelization. They compute a cost graph from the control flow and data dependence graphs and estimate the probability that misspeculation will occur along different paths in the graph. The cost graph, in addition to a set of criteria, determine which loops in a program deserve speculation.

Bhowmik and Franklin [1] build a framework for speculative multithreading on the SUIF-MachSUIF platform. Within this framework, they consider dependence-based task selection algorithms. Like Multiscalar, they focus on compiling the whole program for speculation, but allow the compiler to specify a spawn location as in SPSM. Mitosis [13] also focuses on parallelizing hard-to-analyze applications. A feature of Mitosis is that it generates pre-computation slices to predict the live-ins of tasks. Since the compiler does not need to guarantee the correctness of the pre-computation slices, it performs aggressive optimizations to reduce their overhead.

In each of the above efforts, the compiler statically splits the program into tasks leveraging varying degrees of dependence analysis. In addition, all of these approaches use profiling to guide their task selection by collecting probabilities for common execution paths. In POSH, we use the program structure to identify tasks. We also use profiling to eliminate some tasks after the compiler has identified the tasks. Moreover, the profiler is prefetching-aware.

Some work has used dynamic information to improve selection of tasks for TLS [2, 10, 23]. Jrpm [2] decomposes a Java program into threads dynamically using a hardware profiler called TEST.

While the program runs in TEST, they identify important loops that will provide the most benefit due to speculative parallelization and recompile them with dynamic compilation support. POSH is different from Jrpm in three aspects. First, POSH does not rely on a hardware profiler. Second, POSH considers both loops and subroutine continuations. Third, POSH takes into account prefetching effects in the profiling pass. Marcuello and Gonzalez [10] use profiling to identify tasks but are primarily interested in thread-spawning policies. POSH uses the profiling pass to refine a set of tasks already selected by the compiler. Concurrently to our work, Whaley and Kozyrakis [23] describe a scheme similar to POSH's profiler. They also identify the profiler as a convenient and effective technique to improve task selection, and they show that simple profiling techniques can provide large performance gains. However, they only consider subroutine continuations as tasks, and they do not consider prefetching effects in their profiler.

Many other works have looked at optimizations for speculative threads. Chen *et al.* [3] calculate a probability for each points-to relationship that might exist for a pointer at a given point in the program. This probability can be used to determine whether a squash is likely to occur due to a memory-carried dependence. Zhai *et al.* [24] are concerned with task selection but primarily for replacing dependences with synchronization and alleviating the associated synchronization overheads. Oplinger *et al.* [12] look for the best places within an application to speculate. One important contribution is the use of value prediction to speculate past function calls. We have incorporated some of the techniques from [24] to move data dependences as far apart as possible, and we have exploited the benefits of return value prediction as reported in [12].

## 8. Conclusions

A promising approach to leverage CMPs to speed-up hard-to-analyze codes such as SPECint is to design architectures and compilers for TLS. This paper has focused on TLS compilation and made three main contributions.

First, this paper presented POSH, a new TLS compiler built on top of gcc. POSH leverages the structure of the code (loop iterations and subroutines of any nesting level) to generate tasks, and uses a profiling pass to discard ineffective tasks. Second, this paper showed that, through speculative parallelization, POSH can significantly speed-up hard-to-analyze applications. Specifically, whole SPECint 2000 applications running on a simulated TLS CMP with 4 superscalar cores are sped up by 1.30 on average. Finally, this paper performed a detailed characterization of speedup sources and task behavior. In particular, it found that, for best performance, both subroutine and loop iteration parallelism should be exploited. Moreover, an estimated 26% of the TLS speedup is a result of the implicit data prefetching provided by squashed tasks.

## References

[1] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[2] M. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *International Symposium on Computer Architecture*, June 2003.

[3] 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 *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 25–36, June 2003.

[4] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[5] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 1995.

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

[7] T. Johnson, R. Eigenmann, and T. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

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

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

[10] P. Marcuello and A. Gonzalez. Thread-Spawning Schemes for Speculative Multithreading. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2002.

[11] D. Novillo. Design and Implementation of the TreeSSA. In *Proceedings of the GCC Developer's Summit*, June 2004.

[12] J. T. Oplinger, D. L. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[13] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. In *Conference on Programming Language Design and Implementation*, pages 269–279, 2005.

[14] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[15] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation. In *International Conference on Supercomputing (ICS)*, pages 179–188, June 2005.

[16] G. Sohi, S. Breach, and T. Vijayakumar. Multiscalar Processors. In *Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

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

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

[19] 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.

[20] 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.

[21] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *International Symposium on Microarchitecture*, pages 81–92, November 1998.

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

[23] J. Whaley and C. Kozyrakis. Heuristics for Profile-Driven Method-Level Speculative Parallelism. In *International Conference on Parallel Processing*, pages 147–156, 2005.

[24] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.