# POSH: A Profiler-Enhanced TLS Compiler that Leverages Program Structure *

Wei Liu, James Tuck, Luis Ceze, Karin Strauss, Jose Renau† and Josep Torrellas

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

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

**Abstract**

As Thread-Level Speculation (TLS) architectures are becoming better understood, it is important to focus on the role of TLS compilers. In systems where tasks are generated in software, the compiler has a major performance impact: while it does not need to prove the independence of tasks, its choices of where and when to generate speculative tasks are key to overall TLS performance.

This paper presents POSH, a new, fully automated TLS compiler built on top of `gcc-3.5`. POSH is based on two design decisions. First, to partition the code into tasks, it relies on the subroutine and loop structure of the code. Second, it uses a profiling pass that takes into account both the parallelism and the data prefetching effects provided by the speculative tasks. With the code generated by POSH, a TLS chip multiprocessor with 4 3-issue cores delivers an average speedup of 1.28 for whole SpecInt 2000 applications. Moreover, the profiler increases its effectiveness by 17% if it considers the data prefetching effects of speculative tasks.

## 1 Introduction

Although parallelizing compilers have made significant advances [3, 11], 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.

One way to parallelize these codes is to use Thread-Level Speculation (TLS) (e.g. [1, 6, 9, 10, 12, 14, 16, 20, 21, 22, 23]). The approach is to build tasks from the code, and speculatively run them in parallel, hoping not to violate sequential semantics. As tasks execute, special support checks that no cross-task dependence is violated. If any is, the offending tasks are squashed, the polluted state is repaired, and the tasks are re-executed.

In many of the proposed systems, tasks are generated in software rather than built in hardware. In such cases, the compiler plays a major role. The compiler does not need to prove the absence of dependences across tasks. However, the compiler's choices of how to break the code into tasks and when to spawn them have a major impact on the performance of the resulting TLS system.

There are several instances of TLS compiler infrastructure in the literature [2, 5, 7, 13, 24, 25, 27]. In some of these compilers, tasks are built exclusively out of loop iterations [7, 27]. The reason is that loops are often the best source of parallelism. In other compilers [5, 13, 25], 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 codes with pointers.

In this paper we present POSH, a new, fully automated TLS compiler that we have developed. The compiler adds several 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-3.5 [18]. Building on gcc-3.5 allows us to leverage a complete compiler infrastructure. Moreover, since gcc has various front-ends for different languages and various back-ends for different architectures, POSH is very portable. At this point, POSH only accepts C programs, although it will soon be able to work with Fortran and C++ programs.

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 written by the programmer, namely subroutines and loops. This decision simplifies the compilation algorithms significantly. The second design decision is to add a 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 very small input data set.

To enhance parallelism and data prefetching, POSH also performs aggressive hoisting of task spawns. Moreover, it supports software value prediction. However, to maximize applicability, POSH assumes a simple target Chip Multiprocessor (CMP) architecture, without any architectural support for direct register-to-register data transfer.

*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.

The contributions of this paper are as follows:

- We show that a TLS compiler that, rather than forming tasks based on a data-dependence pass that tries to minimize cross-task dependences, uses instead the code structure (subroutines and loops) and a profiler, can deliver very good speedups. Specifically, a TLS CMP with 4 3-issue cores delivers an average speedup of 1.28 for whole (i.e., not only the loops) SpecInt 2000 applications.

- We show that, for higher effectiveness, the profiler has to take into account both the parallelism and the data prefetching effects provided by speculative tasks. In particular, the profiler increases its effectiveness by 17% if it considers the data prefetching effects.

- We show the performance impact of several important design decisions in the compiler. Specifically, we examine the impact of generating tasks out of only subroutine continuations, only loop iterations, or combinations of them; the impact of the profiling pass, and the effect of value prediction.

Ideally, we would have liked to compare the performance of POSH to other existing TLS compiler infrastructures in the literature. However, the sheer implementation effort required to reproduce the algorithms of another TLS compiler has prevented us from doing it in this paper.

and be beneficial for POSH.

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

## 2 Background on Thread-Level Speculation (TLS)

TLS consists of extracting tasks of work from sequential code and executing them in parallel, hoping not to violate sequential semantics (e.g. [1, 6, 9, 10, 12, 14, 16, 20, 21, 22, 23]). 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.

Memory accesses issued by a speculative task must be handled carefully. Stores generate speculative state that cannot be merged with the non-speculative state of the program. Such state is typically stored in a speculative buffer or cache local to the processor running the task. Only when the task becomes non-speculative can the state be allowed to merge with the non-speculative program state.

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 version of 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. Note that, thanks to the speculative 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 an execution environment that provides feedback to the compiler to improve task selection.

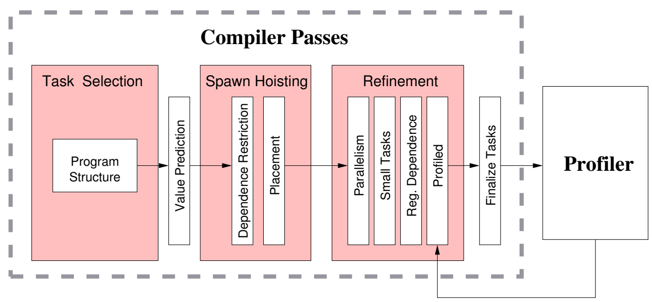


Figure 1: Flowchart of the POSH framework.

### 3.1 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. The live-ins of a task are those variables that the task uses without defining them. In particular, POSH assumes that there is no 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 is written to memory if that value may needed by any successor task. On the other hand, POSH assumes that the hardware will detect dependence violations through memory and will squash and restart tasks accordingly, as in conventional TLS architectures.

The ISA provides a *spawn* and a *commit* instruction to initiate and to successfully complete a task, respectively. The spawn instruction takes as an argument the address of the first instruction in the new task. Execution of the spawn instruction initiates a new task in an idle processor. Execution of the commit instruction indicates to the hardware that the task has completed its work. The compiler inserts 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, the compiler identifies as tasks all subroutines and all loop iterations in the code. For each task, the compiler identifies the instruction where it begins (*begin point*). The compiler inserts spawn instructions in the begin points, creating what we call *spawn points*. Because the begin point of one task is the *end point* of another, the compiler also adds commit instructions before each begin point. The output of the task selection phase is a set of begin points.

Immediately after task selection, the compiler invokes the *Value Prediction* pass. This pass predicts the values of certain kinds of variables that cross task boundaries, hoping to reduce the number of dependence violations. In POSH, we predict function return values and loop induction variables.

In the spawn hoisting phase, POSH considers each of the spawn instructions inserted, 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. Given the spawn point for a task, we hoist it as much as possible subject to two constraints. First, the spawn should be after the definition of all variables used in the task that, according to the intermediate representation, are likely to assign to the registers. The exception is when value prediction is used. Second, the spawn should be in a location that is execution equivalent[1] to the start of the task. These constraints are represented in the figure as the *Dependence Restriction* subpass.

In the refinement phase, POSH makes the final decisions about 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.

[1]We say that two basic blocks $b_1$ and $b_2$ are execution equivalent if two conditions are satisfied: (i) $b_2$ is only executed after $b_1$ is executed and before $b_1$ gets executed again, and (ii) after $b_2$ is executed, $b_2$ is not executed again before $b_1$ is executed, or vice versa. To be clear, this is a static property of the control flow graph alone.

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

In the *Finalize-Task* pass, the compiler inserts all instructions and code 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, allowing us to leverage a complete compiler infrastructure. We use the SSA tree as the high-level intermediate representation [18].

### 3.3 Profiler

The profiler provides a list of tasks that are beneficial for performance. The compiler uses this information to eliminate other non-beneficial tasks. Note that the profiler also informs the compiler of which tasks are not beneficial because the value predictions that they rely on are usually incorrect. Then the compiler also eliminates these tasks.

To perform profiling, we run the applications with the *Train* input set. The execution of the tasks is *serial*, without assuming any TLS architectural support, and modeling only some rudimentary timing. While the tasks run, the profiler collects information about each task that can be used to make a decision regarding the amount of parallelism the task has to offer, the likelihood 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 4.3. On average, a profiler run takes about 5 minutes on an Intel P4 3GHz machine.

## 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 few cross-task dependences, enough work to overcome overheads, and few live-ins. Choosing tasks that provide the optimal performance improvement is NP-hard [2].

POSH's heuristic to select good tasks is to rely on the structure that the programmer gave to the code. Specifically, POSH uses the following modules as potential tasks: sub-

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

(a)

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

(b)

```
    int i=0;
      :
loop:
    if(i>99)
      goto lend;
    <LOOP BODY>   <-- BP
    i=i+1;
    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, its continuation, and the iterations of a loop.

routines from any nesting level, their continuations, and loop iterations from one or more loops in a nest.

As an example, Figure 2 shows how POSH generates tasks out of a subroutine and its continuation (Chart (a)), as well as out of a loop iteration (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 the 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 chart) right before the backward jump. POSH identifies loops in the program by computing the set of strongly connected components (SCC) 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.

### 4.1.1 Spawn Hoisting

With spawn hoisting, we place the spawn of the task as early as possible before the begin point of the task, given the constraints indicated before. Figure 2(b) shows the code from Chart (a) after performing spawn hoisting. Note that the continuation task in Chart (a) (the one starting at *BP1*) had the live-in variable *y*. Consequently, we need to ensure that *y* is written to memory before the continuation task is invoked, and it is read from memory inside the continuation task. POSH ensures this by declaring a volatile variable *v_y* (Chart (b)). Updates to such variable will always be propagated to memory. Then, before the continuation task is spawned, POSH copies the live-in *y* to *v_y*. Inside the continuation task, *v_y* is read from memory and copied to *y*. Finally, as Chart (b) shows, the spawn for the continuation task (*Task 1*) is hoisted all the way up to after the update to *v_y* (spawn point *SP1*).

On the other hand, the spawn for the subroutine task (*Task 2*) can be hoisted all the way to the beginning of the code section, since the task has no live-ins. It will only be hoisted further up if we find a point in the code that is execution equivalent to the call to the subroutine.

Figure 2(b) also includes the commit statements for the tasks. Recall that a commit statement is placed just before each task's begin point.

Finally, Figure 2(d) shows the code from Chart (c) after performing spawn hoisting. As in Chart (b), POSH introduces a volatile variable to ensure that variable *i* is written to memory 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.

## 4.2 Prefetching Effects

While POSH targets task parallelism, it is also specifically designed to reap the benefits of prefetching in TLS. Figure 3 shows the two potential benefits of TLS: parallelism and prefetching. Given *Task 1* and *Task 2* (Chart (a)), TLS exploits parallelism by allowing the overlapped execution of the two tasks (Chart (b)). However, when violations cause tasks to be squashed and restarted, TLS can speed up the program through automatic data prefetching.

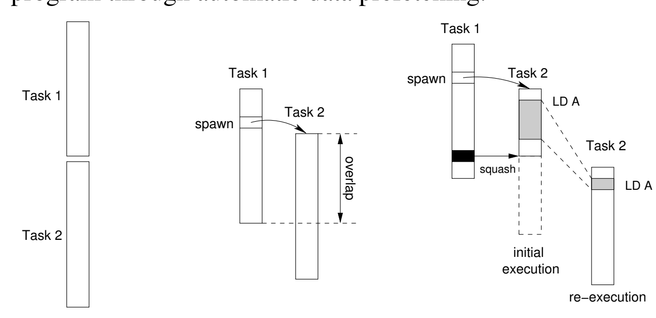


Figure 3: The two potential benefits of TLS: parallelism and prefetching.

This effect is illustrated in Figure 3(c). In its first execution, *Task 2* suffers a miss on variable *A*. After *Task 2* is squashed and restarted, its new access to *A* finds the data already in the cache. Consequently, while there is little parallelism between *Task 1* and *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 clear 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 in the re-execution.

```
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 tries to leverage prefetching through its profiler. We describe the profiler algorithms next.

### 4.3 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. We feel that constraining the profiling runs in this way makes the framework widely usable in a variety of circumstances. The profiler also models a simple L2 cache (without cycle-accurate timing model) to estimate the number of misses. The latter are used for our analysis of prefetching. Simulating a cache without modeling time introduces only very small profiling overhead. Overall, an average profiler run takes about 5 minutes.

To make the profiler as general as possible, its analysis is not tied to any number of processors. Instead, is assumes that an unlimited number of processor cores will be available. As a result, the code that our compiler eventually generates is not optimized for any specific number of processors.

#### 4.3.1 Profiler Execution

In its sequential execution of the program, the profiler estimates L2 cache misses. Moreover, it 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 that allows it to estimate cross-task dependences and squashes.

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 ($Ovhd_{spawn}$).

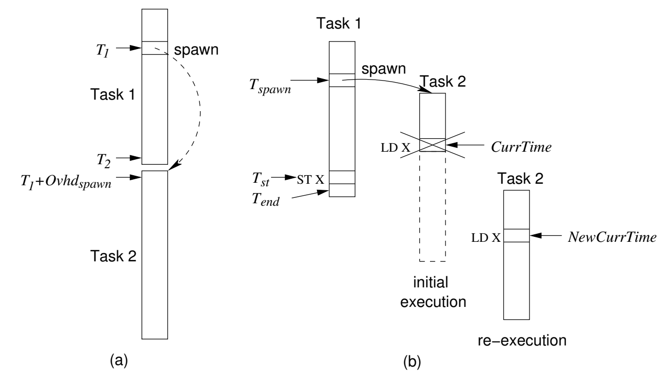


Figure 5: Example of profiler execution.

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 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 the $LDX$ happens before the $STX$ and *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 $NewCurrTime$. $NewCurrTime$ is obtained by the following formula:

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

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.

#### 4.3.2 Benefit of a Squashed Task

Based on the previous discussion, we can roughly estimate 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:

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

In the formula, $T_{end}$ and $T_{st}$ are the times when *Task 1* finishes and *ST X* executes, respectively (Figure 5-(b)).

#### 4.3.3 Task Elimination

The output of the profiler is the list of tasks that are beneficial for performance. To generate this list, the profiler runs as described, and it identifies the tasks that need to be eliminated. There are three elimination criteria: task size, hoisting distance, and squash frequency. We describe these criteria in this section.

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.

The number of instructions between the spawn point of a task and the begin point of that task is called the hoisting distance. 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 $Th_{max\_hd}$. Recall that one of our compiler passes eliminates tasks that have small hoisting distances. The reason why we still have the small hoisting distance threshold ($Th_{min\_hd}$) is that the compiler could only determine the hoisting distance statically.

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 the discussion in Section 4.3.2, some squashes may result in a net positive performance effect due to prefetching. Consequently, we apply a *Prefetching Correction* to this rule. Specifically, if a task to be eliminated has a performance benefit (*Benefit* as defined in Section 4.3.2) higher than a squash benefit threshold $Th_{sb}$, the task is not eliminated.

### 4.4 Software Value Predictor

A general approach for deciding when value prediction should be used is difficult for a compiler, but there are some specific locations where value prediction has been shown profitable in previous studies (e.g., [15, 19, 26]). In POSH, we use value prediction in three cases: function return variables, loop induction variables, and cross-iteration dependences on variables that have a behavior similar to induction variables.

For these cases, POSH uses a software value prediction scheme similar to the one in [7]. Such scheme leverages the TLS dependence tracking hardware to squash a task that used a wrong prediction.

## 5 Methodology

### 5.1 Simulated Architecture

A cycle accurate execution-driven simulator is used to evaluate POSH. The simulator models out-of-order superscalar processors and memory subsystems in detail. The TLS architecture configuration 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 [14]. There is no special hardware for communicating register values between cores. Since the L1 caches need to manage speculative data, we set their access time to a higher value: 3 cycles.

| | | | |
|---|---|---|---|
| Frequency | 4 GHz | ROB | 126 |
| Fetch width | 6 | 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 w/ TLS | 3 cycles | Latency | 12 cycles |
| Latency w/o TLS | 2 cycles | Memory: | |
| Lat. to remote L1 | at least 8 cycles | Latency | 500 cycles |
| | | Bandwidth | 10GB/s |

Table 1: Architecture configuration. All cycle counts are in processor cycles. In our comparison, we use *different* L1 cache access times for TLS and non-TLS.

In our evaluation, we report the speedups of this TLS CMP architecture over the execution of the original (*non-TLS*) application binaries on a single-processor *non-TLS* architecture. Such *non-TLS* architecture has one aggressive 3-issue core, one L1 cache, and one L2 cache like the ones in Table 1. One difference is that the L1 cache has the shorter access time of 2 cycles because it does not have to manage speculative data.

### 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 a 3-issue 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 | $Th_{min\_hd}$ | 120 instructions |
| | | $Th_{max\_hd}$ | 5M instructions |
| $Ovhd_{spawn}$ | 12 cycles | $Th_{sq}$ | 0.75 |
| $Ovhd_{squash}$ | 20 cycles | $Th_{sb}$ | 0 |

Table 2: Profiler parameters.

In the rightmost columns of Table 2, we show the threshold values used to guide our profiling algorithms. $Th_{sz}$ is set to 30 to prevent selecting tasks too small to overcome the overhead of spawning a thread. 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 0.75, which means that a task squashed more than 3 times out of 4 commits will typically be eliminated. Finally, to reap benefits from squashing, we set $Th_{sb} = 0$, which means that a task will not be eliminated if there is any benefit from squashing at all.

### 5.3 Applications Evaluated

The simulated architectures are evaluated with the SpecInt 2000 applications running 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 non-TLS binary we compare against is generated by the same compiler, *gcc-3.5*, with -O2 optimization enabled. Note that there are no TLS or other additional instructions added to the baseline binary. 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 the SGI's source-to-source optimizer (copt from MIPSPro) on the SpecInt code. This pass performs PRE, loop unrolling, inlining, and other optimizations.

To accurately compare the performance of the different binaries, simply timing a fixed number of instructions cannot be used. Instead, "simulation markers" are inserted in the code, and simulations are run for a given number of markers. After skipping the initialization (typically 1-6 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: different task selection algorithms, task characteristics, effectiveness of the profiler, and effectiveness of value prediction. In the evaluation, we select subroutine continuations and loop iterations as tasks.

### 6.1 Different Task Selection Algorithms

To evaluate the performance provided by selecting as tasks only particular code structures, we conducted three experiments in which (1) we only selected the subroutine continuations (*Subr*), (2) we only selected the loop iterations (*Loop*), and (3) we selected a combination of both (*Subr+Loop*). Figure 6 shows the speedup obtained by these three selection algorithms over the *non-TLS* architecture. In all three experiments, we used the profiling pass and enabled value prediction.

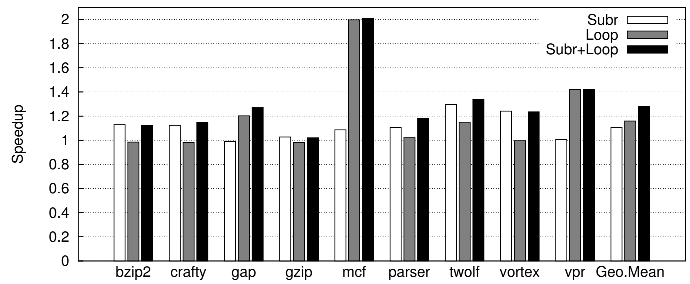


Figure 6: Comparison of different task selection algorithms: subroutine continuations only (*Subr*), loop iterations only (*Loop*), and the combination of both (*Subr+Loop*).

As shown in Figure 6, *Subr+Loop* delivers speedups that reach 2.02 in *mcf*, and have a geometric mean of 1.28. The latter is 15.3% more than in *Subr* and 10.3% more than in *Loop*. For six of the applications (bzip2, crafty, gzip, parser, twolf, and vortex), the *Subr* selection algorithm performs better than the *Loop* one. For the other three benchmarks, gap, mcf and vpr, the *Loop* selection algorithm performs better. Due to the irregular code structure, selecting only either subroutine continuations or loop iterations is not enough to get the best speedup. Instead, using both types of tasks is a simple and effective way to select tasks.

These significant speedups make POSH an attractive TLS compiler infrastructure, given that POSH is a fully-automated compiler that speculatively parallelizes irregular SpecInt programs.

### 6.2 Task Characteristics

Table 3 shows the characteristics of the tasks selected by POSH after all the passes, including the profiler. The second column shows the static number of subroutine continuation tasks, while the third column shows the static number of loops whose iterations will be given out as tasks. The average figures for these parameters are 27.0 and 7.4, respectively. Their relative value is not surprising, given that SpecInt applications usually have many subroutine calls, and loops do not dominate the program execution time. *vpr* is an interesting case, with only two static loops, yet yielding a speedup of 1.40 (as shown in Figure 6). Finally, the last column shows that the average dynamic task size ranges from 54 instructions in *mcf* to 1851 in *vortex*.

### 6.3 Effectiveness of the Profiler

The profiler plays an important role in POSH. According to our design philosophy, the compiler aggressively selects

| Appli-cation | #Static Subroutine Continuation Tasks | #Static Loops with Tasks | #Dynamic Insts per Task (Avg.) |
|---|---|---|---|
| bzip2 | 6 | 10 | 998 |
| crafty | 38 | 5 | 887 |
| gap | 5 | 6 | 288 |
| gzip | 11 | 3 | 661 |
| mcf | 2 | 2 | 54 |
| parser | 147 | 33 | 294 |
| twolf | 13 | 4 | 320 |
| vortex | 21 | 2 | 1851 |
| vpr | 0 | 2 | 454 |
| Average | 27.0 | 7.4 | 645 |

Table 3: Task characteristics.

tasks based on code structure, and lets the profiler eliminate tasks that are detrimental to performance.

Figure 7 shows the effectiveness of the profiler. We conduct three experiments: (1) no profiler is used (*NoProfiler*), (2) we use the profiler without the Prefetching Correction described in Section 4.3.3 (*Profiler_w/o_Prefetch*), and (3) we use the complete profiler (*Profiler_w/_Prefetch*). The only difference in the latter two experiments is the inclusion of prefetching awareness. In both cases, the profiler applies the other elimination rules, namely elimination of small tasks, tasks with too-short or too-long hoist distance, and tasks with frequent squashes. In all three experiments, we select both subroutine continuations and loop iterations, and have the value prediction turned on. The figure shows speedups over the *non-TLS* architecture.

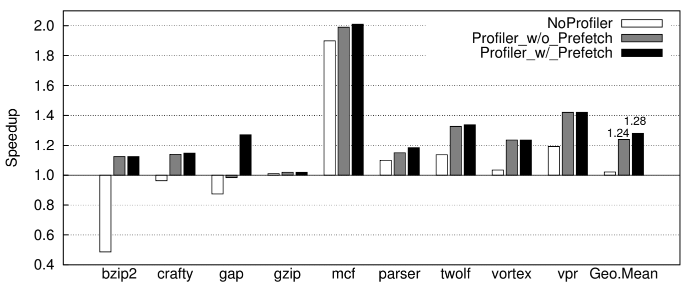


Figure 7: Speedup of POSH over the *non-TLS* architecture in three cases: no profiler, profiler without the prefetch correction, and full profiler.

As shown in Figure 7, without profiler support we obtain a negligible average speedup of 1.02. After applying the profiling pass and without considering prefetching effects, the average speedup increases to 1.24. Finally, when we include prefetching-awareness, we reach the final average speedup of 1.28. From these results, we see that adding prefetch-awareness to the profiler is important for boosting the performance: on average, the profiler increases its effectiveness by 17%.

An especially remarkable case is that of *gap*. If it uses only *Profiler_w/o_Prefetch*, it ends up 2% slower than the sequential run. The profiler eliminates tasks with obvious data dependences, thus losing the opportunity to leverage prefetching. By considering the prefetching factor in the profiler, *gap* boosts its speedup to 1.27.

#### 6.3.1 Characterization of Task Profiling

Table 4 characterizes our task profiling. The second column shows the total static number of subroutine continuations that are tasks plus the loops that contain tasks[2] before profiling. The last column shows the same data after profiling. We can see that a large number of tasks are eliminated by the profiler. On average, 139.7 tasks are selected by the compiler and only 35.4 tasks survive the elimination process, or around 75% of tasks are eliminated on average.

Columns 3-6 of Table 4 show the number of static tasks eliminated due to each of the reasons discussed in Section 4.3.3. Specifically, on average 14.8 tasks are eliminated because of their small size (Column 3), 31.9 tasks because of a small hoisting distance (Column 4), 1.8 tasks because of a large hoisting distance (Column 5), and 55.8 tasks because of frequent squashes (Column 6). The latter effect dominates.

Column 7 of Table 4 shows the number of static tasks remaining after profiling that were retained only due to the *Prefetching Correction* of Section 4.3.3. We can see that, on average, 2.1 tasks were retained because of their prefetching capabilities. While 2.1 tasks is a small fraction of the total 35.4 tasks that are remaining, they have a performance impact, as discussed in Section 6.3.

*gap* benefits the most from prefetching, with 7 prefetch tasks out of a total of 12 selected tasks. The 7 prefetch tasks help to improve the speedup from 0.98 to 1.27 (Section 6.3). Some applications, such as *bzip2, vortex* and *vpr* have no prefetch task selected. In these three applications, this type of prefetching offers no benefits.

### 6.4 Effectiveness of Value Prediction

Figure 8 shows the effectiveness of our value prediction technique. We compare the application speedups with and without the value prediction. In both runs, we use the profiler.

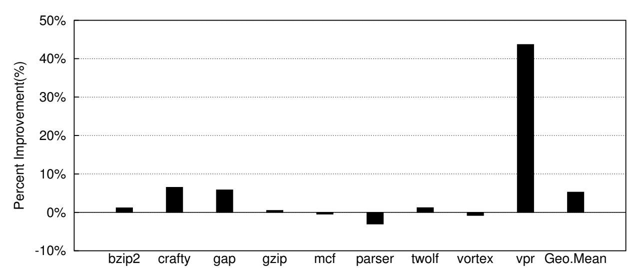


Figure 8: Improvement of the speedups with value prediction over without value prediction.

On average, 5% more speedup is delivered by POSH when value prediction is enabled. In particular, *vpr* gains 43% more speedup. According to Table 3, there are only two loop-based static tasks selected for *vpr*. The induction variables of these two loops are highly predictable and the loops show very good parallelism. Prediction is needed in these two

[2]Note that we are counting each static loop once, irrespective of the number of iterations that it has.

| App. | #Tasks Before Profiling | #Tasks Eliminated Due to Task Size | #Tasks Eliminated Due to Small Hoisting | #Tasks Eliminated Due to Large Hoisting | #Tasks Eliminated Due to Squashes | #Tasks Saved Due to Prefetch | #Tasks After Profiling |
|---|---|---|---|---|---|---|---|
| bzip2 | 115 | 2 | 44 | 1 | 51 | 0 | 17 |
| crafy | 376 | 70 | 99 | 3 | 160 | 1 | 44 |
| gap | 36 | 0 | 11 | 2 | 11 | 7 | 12 |
| gzip | 55 | 1 | 9 | 0 | 30 | 2 | 15 |
| mcf | 17 | 2 | 6 | 0 | 4 | 1 | 5 |
| parser | 464 | 47 | 68 | 10 | 158 | 6 | 181 |
| twolf | 75 | 0 | 30 | 0 | 27 | 2 | 18 |
| vortex | 98 | 11 | 20 | 0 | 43 | 0 | 24 |
| vpr | 21 | 0 | 0 | 0 | 18 | 0 | 3 |
| Average | 139.7 | 14.8 | 31.9 | 1.8 | 55.8 | 2.1 | 35.4 |

Table 4: Characterization of task profiling. Note that the static tasks after the profiling pass (last column) are always one more than the sum of the static subroutine continuation tasks and the loops with tasks in Table 3. The reason is that there is always one initial task in the program execution.

cases because the induction variable updates occur within an if-then-else statement (a solution like that in Figure 2(c) is not feasible).

Some benchmarks are hurt by value prediction. *parser* loses around 3% speedup with value prediction. The overhead of inserting extra instructions to support value prediction is not compensated by the low gains in this application. Since *parser* has frequent squashes, we are left to conclude that the dependences between tasks in parser have lower predictability than anticipated by our profiler.

## 7 Related Work

Several compiler infrastructures for TLS have been proposed but differ significantly in their scope. The Multiscalar compiler [25] 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.* [13]. 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 and, therefore, do not specify when a thread should be launched.

A number of compilers focus only on loops [7, 8, 24, 27]. In SPSM [8], 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 [7] recently presented 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 [2] built a framework for speculative multithreading on the SUIF-MachSUIF platform. Within this framework they considered dependence-based task selection algorithms and, like our work, considered a spawn instruction and looked at thread spawning strategies. Like Multiscalar, they focus on compiling the whole program for speculation but allowed the compiler to specify a spawn location as in SPSM.

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. In addition, we use profiling information to eliminate some tasks after the compiler has identified the tasks. Moreover, the profiler is prefetching-aware.

Some work has used dynamic selection of tasks for TLS [4, 17]. Jrpm [4] 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 [17] 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.

Many other works have looked at optimizations for speculative threads. Chen *et al.* [5] calculated 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.* [27] were concerned with task selection but primarily for replacing dependences with synchronization and alleviating the associated synchronization overheads. Oplinger *et al.* [19] looked for the best places within an application to speculate. One important contribution was the use of value prediction to speculate past function calls. We have incorporated some of the techniques from [27] to move data dependences as far apart in time as possible, and we have exploited the benefits of return value prediction as reported in [19].

## 8 Conclusions

This paper presented POSH, a new TLS compiler built on top of `gcc-3.5`. The paper made three contributions. First, it showed that a TLS compiler that uses the code structure (subroutines and loops) and a profiler, can deliver very good speedups. Specifically, a TLS CMP with 4 3-issue cores delivers an average speedup of 1.28 for whole SpecInt 2000 applications.

Second, the paper showed that, for higher effectiveness, the profiler has to take into account both the parallelism and the data prefetching effects provided by speculative tasks. In particular, the profiler increases its effectiveness by 17% if it considers the data prefetching effects.

Finally, the paper showed the impact of several important design decisions in the compiler, including task types, design parameters in the profiler, and value prediction.

Our future work involves comparing the effectiveness of POSH and other existing TLS compilers. We plan to identify the strengths and weaknesses of different approaches. We are also improving the models used by the profiler. In particular, we are improving the cache models, which should make it possible to gain more from prefetching.

## References


[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, November 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] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

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

[7] 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 *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[8] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, 1995.

[9] M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Technical report, MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001.

[10] 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.

[11] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[12] 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.

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

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

[15] 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.

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

[17] P. Marcuello and A. Gonzalez. Thread-Spawning Schemes for Speculative Multithreading. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, February 2002.

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

[19] J. T. Oplinger, D. L. Heine, and M. 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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