The Jrpm System for Dynamically Parallelizing Java Programs

Michael K. Chen
Stanford University
mikey@hydra.stanford.edu

Kunle Olukotun
Stanford University
kunle@ogun.stanford.edu

Abstract

We describe the Java runtime parallelizing machine (Jrpm), a complete system for parallelizing sequential programs automatically. Jrpm is based on a chip multiprocessor (CMP) with thread-level speculation (TLS) support. CMPs have low sharing and communication costs relative to traditional multiprocessors, and thread-level speculation (TLS) simplifies program parallelization by allowing us to parallelize optimistically without violating correct sequential program behavior. Using a Java virtual machine with dynamic compilation support coupled with a hardware profiler, speculative buffer requirements and inter-thread dependencies of prospective speculative thread loops (STLs) are analyzed in real-time to identify the best loops to parallelize. Once sufficient data has been collected to make a reasonable decision, selected loops are dynamically recompiled to run in parallel.

Experimental results demonstrate that Jrpm can exploit thread-level parallelism with minimal effort from the programmer. On four processors, we achieved speedups of 3 to 4 for floating point applications, 2 to 3 on multimedia applications, and between 1.5 and 2.5 on integer applications. Performance was achieved by automatic selection of thread decompositions by the hardware profiler, intra-procedural optimizations on code compiled dynamically into speculative threads, and some minor programmer transformations for exposing parallelism that cannot be performed automatically.

1. Introduction and Overview

As the limits of instruction-level parallelism (1 – 10s of instructions) with a single thread of control are approached [44], we must look elsewhere for architectural improvements that can speedup sequential program execution. Coarser grained parallelism, like fine-grained thread-level parallelism (10 – 1,000s of instructions), is a potential area for exploration. This type of parallelism is not exploited today due to limitations of current multiprocessor architectures and parallelizing compilers.

Traditional symmetric multiprocessors (SMPs) [13][22] have been most effective at exploiting coarse grained parallelism (>1,000s of instructions). The high cost of inter-processor communication in these systems, generally >100s of cycles, eliminates any potential speedups for fine-grained parallel tasks that either have true dependencies or closely shared cache lines.

Modern compilers that perform array dependence analysis can parallelize Fortran-like numerical applications on traditional multiprocessors [2][5][20][38]. Unfortunately, numerous challenges have made automatic compiler parallelization of general integer programs difficult. Analyzing pointer aliasing, control flow, irregular array accesses, and dynamic loop limits as well as handling inter-procedural analysis complicate static dependence analysis [3]. These difficulties introduce imprecision into dependence relations, limit the accuracy of parallelism estimates, and force conservative synchronization to safely handle potential dependencies.

Our paper describes the Java runtime parallelizing machine (Jrpm), a complete system that automatically parallelizes sequential programs using thread-level parallelism. This system takes advantage of recent developments that now enable a different approach to automatic parallelization for small multiprocessors (2 – 8 CPUs). The key components of this system are: a chip multiprocessor that provides low-latency inter-processor communication, thread speculation support that allows us to parallelize optimistically, a hardware profiler for identifying parallel loops from program execution, and a virtual machine environment where dynamic optimizations can be performed without modifying source binaries.

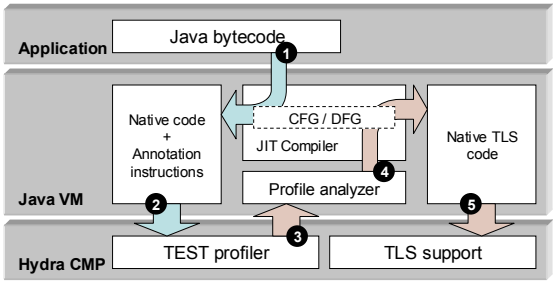
- **Chip multiprocessor –** Jrpm is based on the Hydra chip multiprocessor (CMP) [32]. Decreasing feature size and increasing transistor counts now allow chip multiprocessors to be a reality [6][14][24][42]. Chip multiprocessors combine several CPUs onto one die with a tightly coupled memory interface. In this configuration, inter-processor sharing and communication costs are significantly less than in traditional multiprocessors. The reduced communication costs make it possible to take advantage of fine-grained thread-level parallelism.
- **Thread-level speculation –** Hydra includes support for thread-level speculation (TLS) [10][21][29][40]. TLS allows a sequential program to be divided into arbitrary chunks of code, or threads, to be executed in parallel. TLS hardware ensures memory accesses between threads maintain the original sequential program order.

Traditional multiprocessors must synchronize conservatively to preserve correctness when static analysis cannot determine with complete certainty that a dependency does not exist. For non floating-point applications, this makes it difficult to find regions that can be parallelized safely and still achieve good speedups. With TLS, it is possible to parallelize

aggressively rather than conservatively. Since sequential ordering is guaranteed for parallel execution of any program decomposition under TLS, the problem of compilation is reduced to finding regions of program execution with significant parallelism.

• **Hardware profiler –** Static parallelizing compilers often have insufficient information to analyze dynamic dependencies effectively. Dynamic analysis to find parallelism complements a TLS processor's ability to parallelize optimistically and use hardware to guarantee correctness. Tracer for Extracting Speculative Threads (TEST) [9] is hardware support in Jrpm that analyzes sequential program execution in real-time to find the best regions to parallelize. This system provides accurate estimates of dynamic dependency behavior, thread size, and buffering requirements that are needed for selecting good decompositions and that would be difficult to derive statically. Estimates show the tracer requires minimal hardware additions to our CMP and causes only minor slowdowns to programs during analysis.

• **Virtual machine –** The Java virtual machine (JVM) [28] acts as an abstraction layer to hide dynamic analysis and thread-level speculation from the program. Virtual machines like the JVM and Microsoft's .NET VM have become commercially popular recently for supporting platform independent applications. Binaries are distributed as portable, platform independent *bytecodes* [28] that are dynamically compiled into native instructions of the underlying architecture and run within the protected environment of the VM. This virtualization allows us to seamlessly support a new execution model without modifying the source binaries.



1. Identify possible thread decompositions by analyzing bytecodes and compile natively with annotation instructions.
2. Run annotated program sequentially, collecting TEST profile statistics on potential thread decompositions.
3. Post-process profile statistics and choose thread decompositions that provide the best speedups.
4. Recompile code with TLS instructions for selected thread decompositions.
5. Run native TLS code.

Figure 1 – Block diagram of Jrpm.

These basic components are used to create a system that can automatically compile programs to exploit thread-level parallelism. A block diagram of Jrpm outlining how the various software and hardware components work together is shown in Figure 1. The compiler derives a control-flow graph from program *bytecodes* and analyzes it to identify potential thread decompositions. A program that has been dynamically compiled with instructions annotating local variables and thread decompositions is executed as a sequential program on a single Hydra processor. The trace hardware collects statistics in real-time for the prospective decompositions. Once sufficient data has been collected by the trace hardware, regions predicted to have the largest speedup and most coverage are dynamically recompiled into speculative threads.

This dynamic parallelization system has other potential benefits:

• **Reduced programmer effort –** Explicit coding of parallelism is better suited for coarser grained and distinct tasks. Manually identifying fine-grained parallel decompositions can be a time-consuming effort for programs without obvious critical sections.

• **Portability –** The code maintains platform independence because the binaries are not modified explicitly for thread speculation.

• **Retargetability –** Decompositions can be tailored dynamically for specific hardware or data. Different decompositions may be chosen for future CMPs with more CPUs, larger speculative buffers, and different cache configurations. In some applications, decompositions can also be optimized for specific data set sizes.

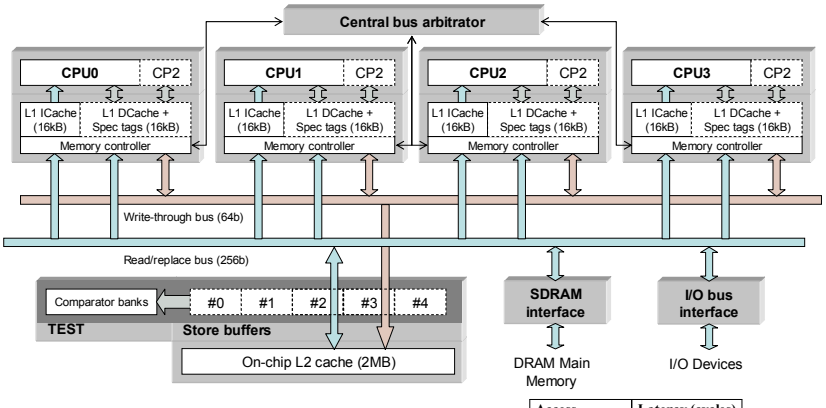
• **Simplified analysis** – Compared to traditional parallelizing compilers, the Jrpm system relies on more hardware for TLS and profiling support, but reduces the complexity of the analysis required to extract exposed thread-level parallelism.

Simulations demonstrate Jrpm's ability to exploit thread-level parallelism automatically with minimal effort from the programmer. On our CMP with four CPUs, we achieve speedups of 3 to 4 on floating point applications, 2 to 3 on multimedia applications, and between 1.5 and 2.5 on integer applications. Selection of appropriate thread decompositions by the hardware profiler, low-level optimizations on code dynamically compiled into speculative threads, and some manual programmer transforms for exposing parallelism when it cannot be found automatically are all key to getting good performance.

The remainder of this paper details the major components of Jrpm and provides results from simulation. Section 2 introduces the Hydra CMP with speculation support. Section 3 describes the hardware profiling system used to select good speculative decompositions. Section 4 shows compiler optimizations performed on selected speculative thread loops. Section 5 explains performance and correctness issues for running thread speculation within a JVM. Section 6 presents the results of running real programs on Jrpm. Section 7 compares related work and Section 8 summarizes our findings.

2. CMP with TLS Support

Hydra [32], shown in Figure 2, is a chip multiprocessor consisting of 4 single-issue, pipelined MIPS processors, each with private L1 data and instruction caches. High-speed read and write buses enable low-latency inter-processor communication that



Buffers	Per-thread limit	Associativity
Load buffer	16kB (512 lines x 32B)	4-way
Store buffer	2kB (64 lines x 32B)	Fully

Access	Latency (cycles)
L2	5
Interprocessor	10
Main memory	50

Figure 2 – Block diagram of our CMP. TLS support blocks shown in dotted lines. TEST profile hardware blocks shown in dark blocks.

makes it possible to take advantage of thread-level parallelism even when there is substantial inter-processor sharing. An integrated on-chip shared L2 cache minimizes cache misses to lower memory when the processors work on shared data.

Table 1 – Thread-level speculation overheads.

TLS Operation	Overhead (cycles) Old	New	Work performed
`STL_STARTUP` *(Master CPU only)*	41	23	Clear store buffers Set speculative event handlers Store $fp & $gp Wake up slave CPUs Enable TLS
`STL_SHUTDOWN` *(Master CPU only)*	46	16	Wait to become head CPU Disable TLS Kill slave CPUs
`STL_EOI` End-of-iteration	14	5	Wait to become head CPU Commit store buffer Clear cache speculation tag bits Start new thread
`STL_RESTART` Violation and restart	13	6	Clear store buffers Clear cache speculation tag bits Restore $fp & $gp

Thread-level speculation (TLS) allows a sequential program to be divided into threads to be executed in parallel. Data speculation hardware ensures memory ordering of the original sequential program is maintained by the threads. Read-after-write (RAW) dependencies are guaranteed by forwarding data to sequentially later threads, and forcing restarts of threads when a speculative thread reads a value too early and sequential memory ordering is violated. By buffering and committing speculative writes in threads according to the original program order, write-after-write (WAW) ordering is maintained. Finally, buffered writes are only available to sequentially later threads, preventing write-after-read (WAR) violations.

Speculative thread support in Hydra consists of coprocessors for each CPU that control thread speculation, extra speculative tag bits added to the processor L1 data caches to detect inter-thread data dependency violations, and a set of secondary cache store buffers to hold speculative data until it can either be safely committed to the secondary cache or discarded [21]. The physical limits of buffered speculative state in Hydra are given in Figure 2.

TLS is controlled in an application through special instructions that access the coprocessor and through special stores issued onto the write bus. For a loop transformed into speculative threads, there are overheads for starting and shutting down speculation, at the end of every iteration, and on dynamic RAW violations, as shown in Table 1. This table shows improvements in the software handlers (*New*) compared to overheads reported previously for our runtime system (*Old*). The significant reduction in overheads results from improved coding of the speculative control routines and more efficient speculation control instructions.

3. Selecting Thread Decompositions

A speculative thread loop (STL) is *a loop decomposed into threads, where each loop iteration is one thread*. Because hardware guarantees correct parallel execution of speculative threads, compiling for this execution model involves finding regions in programs where thread-level parallelism can be exploited within our hardware constraints:

- Inter-thread data dependencies, or RAW hazards, limit parallel execution of speculative threads.
- Speculative read and write state buffered by the hardware can overflow and force speculative execution to stall until reads or writes can be performed safely (e.g. when the thread becomes the non-speculative "head" thread).
- Code compiled as speculative threads introduces overheads from speculation handlers and communication of inter-thread (loop-carried) dependent local variables.
- Only one STL (e.g. one loop level in a loop nest) may be active at a given time.

These constraints impose conflicting objectives when selecting STLs. Speculating on small loops limits coverage, precludes speculation on outer loops, and suffers from proportionally higher speculative thread overheads. Speculating on large loops can increase speculation buffer overflows and incur high penalties for late RAW violations.

As discussed earlier, detecting parallelism with static compilers in large, general programs is challenging [3]. Jrpm relies on dynamic analysis to find parallelism, which complements a TLS processor's ability to parallelize optimistically and use hardware to guarantee correctness. The Tracer for Extracting Speculative Threads (TEST) [9] profiles sequential program execution and collects dependency timing, thread length, and speculative buffer usage estimates to find the best loops to transform into STLs. TEST support requires

little hardware support and incurs only minimal slowdown to programs during analysis. Section 3.1 is an overview of the analysis used to select STLs and Section 3.2 briefly shows how the analysis is implemented in hardware and software.

3.1 Selecting Good Decompositions

TEST analysis is applied to any loop which does not have any obvious serializing dependency. TEST analysis relies on comparing event *timestamps* from different events during sequential execution to derive statistics for a potential STL.

Two primary analysis are performed by TEST to characterize the effects of real constrains on a potential STL. The *load dependency analysis* looks for inter-thread dependencies that limit parallelism. *Timestamps* recorded on previous memory or local variable stores are retrieved during a load to the same address and compared to *thread start timestamps* to determine if an inter-thread dependency arc exists. The dependency with the smallest *arc distance arc* is also recorded as the *critical arc* that limits parallelism between the threads.

The *speculative state overflow analysis* ensures that speculative state for a STL can fit within the L1 caches and store buffers, preventing potential TLS stalls due to speculative state overflows. In this analysis, a *cache line timestamp* and *cache line tag* is recorded for the cache line a heap load or store would have hit. Subsequent memory accesses check for a previously recorded *cache line timestamp* with a matching *cache line tag*. If no *timestamp* exists or if it is less than the current *thread start timestamp* of a STL, a counter, either the *load counter* for speculatively read cache lines or *store counter* for store buffer entries, is incremented to reflect new buffer state required by the current thread. The *overflow counter* is incremented if either counter for the current thread exceeds hardware buffer limits.

The results of the *load dependency analysis* and the *buffer overflow analysis* are accumulated over time and used to predict the performance of a potential STL. Two heuristics signal that sufficient data has been collected for a potential STL: at least 1000 iterations have been executed or outer potential STLs are predicted to consistently overflow speculative state.

Once enough profiling data has been collected, the estimated speedup for a STL is computed using average dependency arc frequencies, thread sizes, critical arc lengths, overflow frequencies and speculative overheads. Only loops that have average iterations per entry >> 1, speculative buffer overflow frequency << 1, and predicted speedups > 1.2 are recompiled into speculative threads. In a loop nest, where we are limited to speculating at one loop level at a time, we select the best STL from those possible by comparing the estimated execution times of speculating at different levels in the loop nest.

3.2 Hardware Profiling Support

Without hardware support, our simulations indicate execution slows over 100x when applying this analysis with software alone. This magnitude of slowdown is unacceptable in a real dynamic compilation system, and the sizeable software overheads introduce significant imprecision into the collected statistics.

Table 2. Annotating instructions and associated operations.

Instruction	Normal operation	Annotation (when enabled)
`lw|lb|lbu|lh lhu|lwc1 addr`	Load	Load
`sw|sb|sh| swc1 addr`	Store	Store
`lwl vn`	none	Local variable load
`swl vn`	"	Local variable store
`sloop n`	"	Start loop
`eoi`	"	Loop end of iteration
`eloop n`	"	End loop

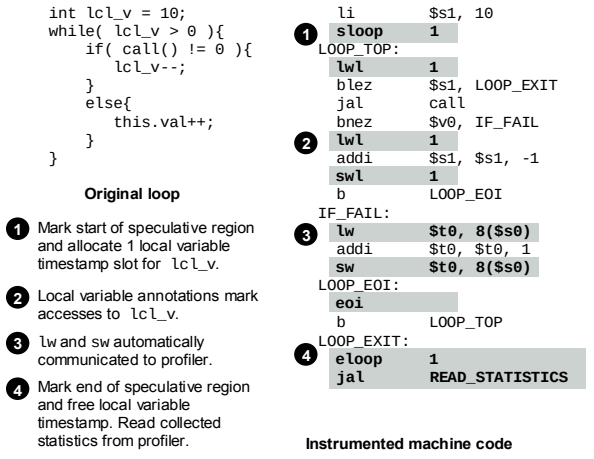


Figure 3 – Example of an annotated loop.

TEST hardware works when speculation is disabled and requires only minor hardware additions to our CMP design. The additions are new annotation instructions to be inserted by the dynamic compiler and hardware comparator banks that perform the trace analysis. The speculative store buffers, which are idle during non-speculative execution, hold *timestamps* for the profiler. Estimates indicate the hardware support adds less than 1% to the total transistor count of Hydra.

Annotating instructions, listed in Table 2, are automatically inserted by the JIT compiler. The compiler derives a control-flow graph (CFG) from Java *bytecodes*. All natural loops [31] identified from the CFG are marked as prospective STLs and denoted with special annotations, as illustrated in Figure 3. After compiler optimizations to eliminate unnecessary annotations, benchmarks experience an average 7.8% slowdown during profiling, and only 2 applications have slowdowns approaching 25%, as shown later in Figure 8. Higher profiling overheads are seen in some programs due to the cost of annotations on potential STLs with small loop bodies.

Store buffers that normally hold writes during speculative execution hold *timestamps* during TEST profiling. The store buffers are statically partitioned so that three buffers hold heap access *store timestamps*, one holds *cache line timestamps*, and one holds *store*

timestamps to local variables. An address' *timestamp* is returned when requested by a memory access or local variable annotation.

The comparator banks carry out the bulk of the real-time dependency and overflow analyses. Memory load and store instructions and the special annotating instructions are automatically communicated through the memory system to the banks during profiling. One comparator bank tracks the statistics for one potential STL by analyzing *timestamps* from the executing memory and local variable loads and stores. An array of eight comparator banks allows us to analyze multiple potential STLs executing simultaneously in loop nests. Comparators compare *thread start timestamps* against incoming *cache line timestamps* to check for speculative state overflow and against incoming *store timestamps* to identify *critical arcs*. At the end of each thread of a potential STL, critical arc lengths, critical arc counts, and buffer overflows are accumulated into counters. The accumulated statistics are read from TEST upon exit from a potential STL.

4. Compiling Thread Decompositions

4.1 microJIT Dynamic Compiler

Our Java runtime system is based on the open-source Kaffe VM (http://www.kaffe.org/), but we used our own JIT compiler and garbage collector to address performance limitations of the original VM. We developed microJIT [8], a fast, dataflow compiler that can generate code 30% faster than the Sun-client dynamic compiler [17], which performs only limited optimizations, and over 10x faster than Sun-server compiler [12], an optimizing static single assignment (SSA) dynamic compiler [31]. Code generation speed was achieved without sacrificing code quality by interleaving compilation stages and minimizing compiler passes. Code optimizations performed by our compiler include common sub-expression elimination, copy propagation, constant propagation, loop invariant code motion, inlining and global register allocation. The resulting code generated by microJIT is competitive with commercial JIT compilers. Our experiments that show the generated code is 20% faster than Sun-client generated code and is within 25% of the performance of Sun-server generated code [8].

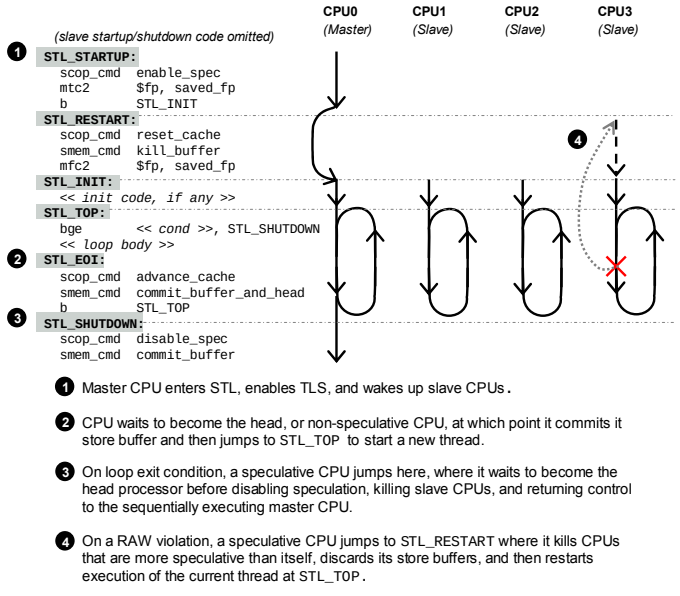


Figure 4 – Illustration of STL code executed by the processors.

For the Jrpm system, the microJIT compiler was augmented to generate speculative thread code. Speculative thread control routines from Table 1 are inserted into STLs chosen by TEST analysis, as illustrated in Figure 4. In addition to the fixed overheads of the speculative handlers, additional overheads may be introduced in specific circumstances. STL initialization values must be saved to the runtime stack by the master processor and then loaded by the slave processor. Cleanup code must be inserted at STL_STARTUP and STL_SHUTDOWN by some of the optimizations below. Local variables that may cause inter-thread (loop carried) dependencies in a STL must be communicated through forced loads and stores in the runtime stack.

4.2 Optimizations for Improving Speculative Performance

Optimizations that improve speculative performance, like register allocating loop invariants, using non-violating loop inductors, inserting synchronization locks, and identifying reduction operations, are applied automatically to selected STLs when possible. These compiler optimizations are described below.

4.2.1 Loop invariant register allocation. Traditionally, register allocating loop invariants eliminates the need to reload a read-only value every time it is used in a sequential loop. Register allocating loop invariants in STLs provides similar benefits, but is a little trickier to implement. On a RAW violation, the register state of a speculative thread is unknown because it was interrupted in mid-execution. Registers must be restored to their original thread start value before the thread can be restarted. One possible way of accomplishing this is by maintaining a shadow register file that holds the thread start register state so that it can be quickly restored on restart. Unfortunately, this approach requires a significant amount of extra hardware to mirror all the architectural registers and complex logic to restore register state in a minimal number of cycles. We accomplish similar functionality with software only. Register allocated loop invariants are saved to the runtime stack during TLS startup. On a RAW violation, only relevant registers are restored by a custom STL_RESTART handler, as illustrated in Figure 5.

4.2.2 Non-communicating loop inductors. Loop induction variables are always incremented the same amount in every loop iteration. Loop

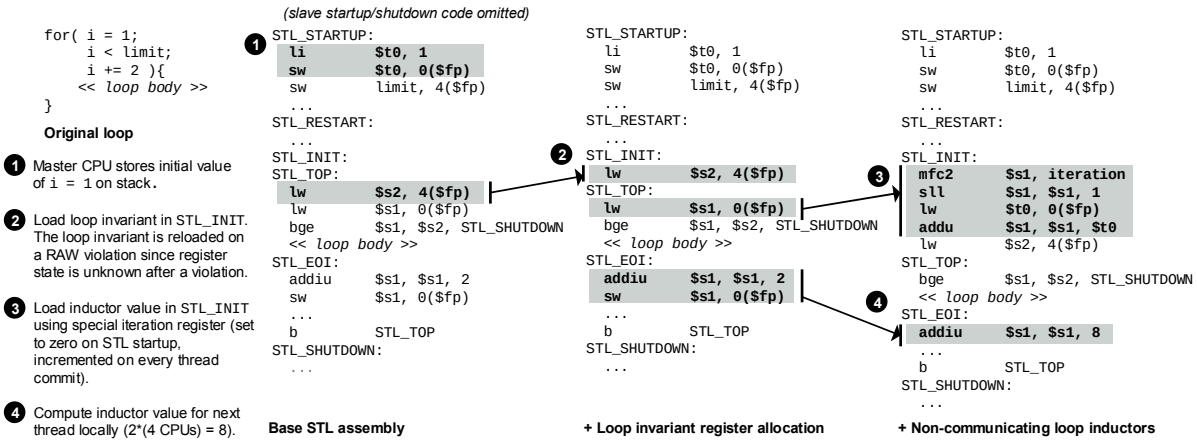


Figure 5 – Illustration of two optimizations. Arrows show how code is moved by the optimizations.

inductors can be a performance problem for thread speculation on loops because they result in a inter-thread (loop carried) dependency. Communication of inductors introduces serialization between loop iterations and can cause RAW violations. Previous proposals [11][41][45] have used value prediction or scheduling to eliminate these negative effects.

We use software and a simple hardware iteration counter to completely eliminate communication of loop inductors. To keep software and hardware requirements low, our runtime system distributes loop iterations of a STL to processors in a simple round robin order [33]. For four CPUs, this implies that CPU0 always handles iterations 0, 4, 8, …, CPU1 always handles iterations 1, 5, 9, …, and so forth. This configuration allows us to easily compute the next value of the loop inductor locally for the next speculative thread that a processor handles, as shown in Figure 5. Like loop invariant register allocation, the value of register allocated inductors must be properly recovered after a thread restart. To achieve this, the `STL_RESTART` handler can compute the correct current value of the inductor using a simple counter tracking the current speculative thread iteration being executed by a CPU.

4.2.3 Reset-able non-communicating loop inductors. Non-communicating loop inductors can also be extended to handle variables that look like loop inductors, but occasionally have an unpredictable value at an unknown loop iteration. The code to handle this is similar to the code to handle normal loop inductors, except that when the rare condition is encountered, the correct current value is written and a violation and restart is forced to later speculative threads. Within the STL_RESTART handler, the restarted speculative threads can then load the correct, updated value.

4.2.4 Thread synchronizing lock. Previous studies have shown that minimizing violation frequency can improve TLS performance [11][41]. For a frequent loop carried dependency, it is often better to wait for the correct value than to get an incorrect one, violate, and then have to restart. We can protect such a variable using a synchronization lock that stalls speculative threads until the variable has been properly updated, as shown in Figure 6. This lock uses a special instruction, `lwnv` (load word, non-violating), that acts like a normal `lw` to return the most recently written value, even if it is speculative, but can never cause a violation and restart. Jrpm automatically inserts a synchronizing lock into compiled code when the TEST profile result for a STL shows a frequently occurring inter-thread dependency (e.g. > 80%) that has an *average dependency arc length* much less than the *average thread size.*

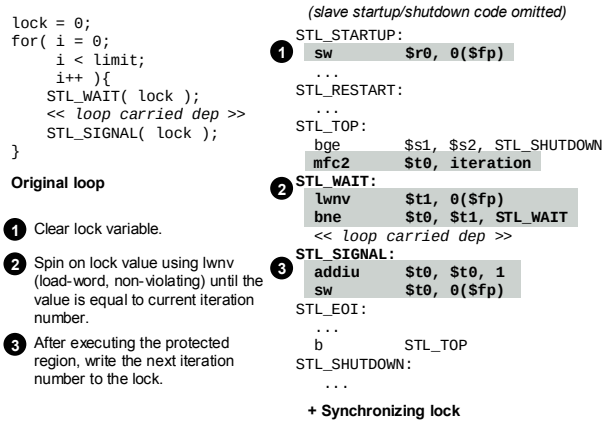


Figure 6 – Illustration of a thread synchronizing lock.

4.2.5 Reduction operator optimization. Reduction operations [3], like computing sums or finding min/max values, are loop carried dependencies that can be easily transformed so they do not result in inter-thread dependencies. Reductions can be computed locally for the speculative threads that a given processor is handling. The locally computed value from each processor can be merged at the end of a STL to get the correct final value.

```
STL_STARTUP
for( ){ // outer loop
        << outer loop body >>
        if( rare condition ){
                STL_SWITCH_STARTUP
                for( ){ // inner loop
                        << inner loop body >>
                        STL_EOI
                }
                STL_SWITCH_SHUTDOWN
        }
        STL_EOI
}
STL_SHUTDOWN
```

Figure 7 – STL assembly for multilevel STL decompositions.

4.2.6 Multilevel STL decompositions. Multilevel STL decompositions are useful in irregularly structured nested loops, like the one shown in Figure 7. The two loops here both contain significant parallelism, but the inner loop, contained within a conditional block, is executed infrequently. We would normally choose to speculate on the outer loop, but speculating on the outer loop while executing the infrequent inner loop could lead to load imbalance between iterations that degrade speculative performance.

Our compiler can insert low-overhead handlers so that the selected STL can be switched to the inner loop and then restored to the outer loop upon completion of the inner loop. Multilevel STL decompositions can be applied automatically when TEST profiling shows that the inner loop is executed infrequently (e.g. the loop iteration count of the outer loop is much larger than the loop entry count of the inner loop).

4.2.7 Hoisted startup and shutdown routines. If the number of iterations per entry into a STL is low, then the overhead of the STL_STARTUP and STL_SHUDOWN handlers can be proportionally high. More than half of this handler code wakes up the other CPUs (which are idle or interrupted from another process) and initializes the speculation hardware. This code does not need to be re-executed in regions of code that may have several STLs or when a STL is entered multiple times, like a selected STL within a loop nest. In these cases, startup and shutdown overheads can be reduced by hoisting some of the STL startup and shutdown handler code to the entry and exit of a given method, or to the outer-most loop.

5. Virtual Machine Considerations

5.1 Exception Handling

One of the nice properties of thread-level speculation is that it preserves the sequential execution behavior even for unexpected events like exceptions. Threads may generate false exceptions during TLS because computations are based on speculative memory references (e.g. a thread with a null pointer exception may be restarted to execute with different memory values). Consequently, a speculative thread with an exception must wait until it becomes the non-speculative "head" thread [21]. Only when this happens is the exception real and must be handled. In Java, hardware and software exceptions are "thrown" up the call stack until it is caught by a `catch` handler [16]. Nothing special needs to happen if an exception thrown during speculation is caught within an executing STL by a `catch` handler. If an exception thrown during speculation is not caught within a STL, exception handling code must properly terminate speculative execution before jumping to an exception handler external to the STL.

5.2 Garbage Collection

Originally, memory allocation during speculation interfered with performance in several benchmarks. Our JVM uses a fast concurrent mark-and-sweep garbage collector [23]. Unallocated objects are stored in a linked free list. Allocating objects on every speculative thread of a STL caused a serializing dependency on this linked list. To deal with this, we parallelized access to the allocator during speculation. This was achieved by maintaining several free lists that can be accessed privately by each processor during speculative execution. Similar techniques could be applied to prevent serializing dependencies in JVMs that use copying garbage collectors [23].

5.3 Synchronized Objects

In Java, method invocations can be explicitly synchronized to an object (using the `synchronized` keyword [16]) to prevent concurrent accesses from multiple explicit Java threads. A simple object lock will cause an inter-thread dependency if it is locked and unlocked on every iteration during speculation. Such a lock is unnecessary during speculative execution because the correct sequential ordering of accesses by concurrently executing speculative threads is guaranteed. Unfortunately, object locks cannot be omitted from compiled code because they may be compiled into methods called from a STL during speculation, but the method may also be called during normal non-speculative execution. We re-implemented the lock routine to prevent object synchronization from causing serialization during speculation, while maintaining the correct behavior during normal execution.

6. Performance Results

Table 3 lists benchmarks we ran on Jrpm, including applications from the jBYTEmark (http://www.byte.com/), SPECjvm98 (http://www.specbench.org/), Java Grande (http://www.epcc.ed.ac.uk/javagrande/javag.html) benchmarks suites, as well as real applications found on the internet.

6.1 Selected Speculative Thread Loops

Table 3 summarize the characteristics of the STLs chosen from analysis. Overall, we found significant diversity in the coverage of selected STLs. While many programs have critical sections, Assignment, NeuralNet, euler and mp3 have many STLs that contribute equally to total execution time. Several programs have more

selected STLs (column e) than are shown in the table, but the omitted decompositions do not have any significant coverage (< 0.5% of total execution time). mp3, db, jess, and DeltaBlue have significant sections of serial execution (column i) not covered by any potential STLs that limits the total speedup for these applications.

TLS can simplify program parallelization, but not all programs can benefit from it. Some integer benchmarks evaluated using TEST showed no potential for speedup using speculation. Programs with system calls in critical code, like I/O intensive programs, currently cannot benefit from TLS because system calls cannot be handled speculatively in our system. Several programs had loops with characteristics that suggested little or no thread-level parallelism. Pizza, a Java source to bytecode compiler, and jack, a parser generator, contained loops that were both too large and executed too few iterations for speculation. Richards, a task scheduler, contained a serializing dependency within its critical STL that could not be optimized. Of two different raytracer programs we looked at, one had a parallel loop that was predicted to consistently overflow the speculative buffers, while the other had a parallel loop that fit within the speculative buffers. This last example suggests that the way a program is written can strongly affect how well it can be automatically parallelized using TLS.

The larger programs contain significant numbers of loops (column c) that would have made manual identification of STLs a time consuming task. A visual analysis of the source code identified that less than half of the benchmarks can be analyzed by a traditional parallelizing compiler (column a) (e.g. they resemble floating point code w/ regular array accesses, no dynamic pointers, bounded loops, and little control flow). The average height of selected STLs (column f) have granularities larger than the inner-most loop in a loop nest. The maximum loop nesting depth column (column d) indicates that eight comparator banks are sufficient to analyze most of the nested loop structures in the benchmark programs. When deeply nested loops with a depth greater than eight must be analyzed, banks are initially allocated to the outer-most loops, but are freed to be used by inner loops when the outer loops eventually predict speculative buffer overflow.

Our experiments suggest that our dynamic parallelization system can make STL selections that account for input data set sizes. We noticed several applications where the best decompositions can change according to input data sizes (column b). In these benchmarks, multiple levels of parallelism exist in key loops. Assignment, NeuralNet, LUFactor, euler, and shallow use a nested loop to traverse a 2-dimensional data array. For these programs, loops lower in a loop nest must be chosen with larger data sets because the number of inner loop iterations will rise, increasing the probability of overflowing speculative state when speculating higher in a loop nest.

6.2 Speculative Thread Performance

Each benchmark was run as a sequential annotated program on Jrpm with the TEST profiling system enabled. The benchmark was then recompiled and executed using speculative threads with the STLs selected by TEST. Figure 8 displays the slowdown during profiling, the predicted TLS execution time from TEST analysis, and actual TLS performance. Figure 9 shows total program speedup (adding compilation, garbage collection, profiling, and recompilation overheads) compared to serial execution for a given

Table 3 – Characteristics and performance statistics for benchmarks evaluated with STLs selected by TEST.

			Characteristics				TEST		TLS Statistics					Speedups from TLS Optimizations									
			(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)	(q)	(r)	(s)	(t)	(u)
Category / Benchmark	Description	Data set	Analyzable	Data set sensitive	Loop count	Loop depth	Selected loops	Avg. loop depth	Threads / STL entry	Thread size (cycles)	Serial exection	Avg. load buffer sz.	Avg. store buffer sz.	Opt - Overheads	Opt - Loop inv.	Opt - Reduction	Opt - Sync	Opt - Resetable ind.	Opt - Hoisting	Opt - Multilevel	JVM - Allocation	JVM – Java lock	Manual
Integer																							
Assignment	Resource allocation	51x51	N	Y	32	5	11	2.0	29	199	1%	1.2	0.2	11%	2%					3%			
BitOps	Bit array operations		N	N	4	2	2	1.0	7646	29	0%	1.1	0.8	73%	10%			128%					
compress	Compression		N	N	28	4	4	1.8	93755	546	0%	4.1	3.1	2%	1%								42%
db	Database	5000.	N	Y	37	5	6	1.7	23142	510	3%	7.8	2.9	2%	2%		5%				29%	31%	44%
deltaBlue	Constraint solver		N	N	22	4	5	2.6	82	501	22%	6.5	2.9	1%									1%
EmFloatPnt	FP emulation		N	N	7	3	1	2.0	255	20127	0%	11.0	9.5								16%		
Huffman	Compression		N	N	14	3	6	1.3	502	108	0%	2.0	0.8	18%			2%						65%
IDEA	Encryption		Y	N	2	2	1	2.0	242	6307	0%	1.9	2.2										
jess	Expert system		N	N	134	11	4	5.3	166	339	27%	6.6	2.8	1%	1%								
jLex	Lexical analyzer gen		N	N	128	6	7	3.0	71	2699	7%	4.1	2.4				1%				11%	8%	
MipsSimulator	CPU simulator		N	N	19	4	2	3.5	51931	1313	0%	11.3	4.4	1%								79%	17%
monteCarlo	Monte carlo sim.		N	N	15	3	5	1.4	942	119	5%	2.4	0.8	3%		27%	18%						74%
NumHeapSort	Heap sort		N	N	5	3	2	2.0	6081	555	0%	3.4	3.5	2%	1%								245%
raytrace	Raytracer		N	N	14	4	1	1.0	65	158	9%	0.0	0.0	6%	1%								
Floating point																							
euler	Fluid dynamics	33x9	Y	Y	32	2	13	1.1	66	304	1%	4.8	1.6	5%							88%		
fft	Fast fourier trans.	1024.	Y	Y	5	3	2	2.0	187	231	0%	1.4	3.8	2%			8%						
FourierTest	Fourier coefficients		Y	N	2	2	1	2.0	100	167802	0%	4.3	3.5										
LuFactor	LU factorization	101x101	Y	Y	13	3	7	1.6	64	455	0%	1.9	0.6	3%	2%								
moldyn	Molecular dynamics		Y	N	8	2	1	1.0	1026	96	2%	2.0	0.0	10%									
NeuralNet	Neural net	35x8x8	Y	Y	19	4	8	1.9	9	617	1%	1.7	2.5	6%	3%	2%							
shallow	Shallow water sim.	256x256	Y	Y	11	3	3	1.0	257	1420	0%	7.8	6.2	1%									
Multimedia																							
decJpeg	Image decoder		N	N	61	5	21	2.2	34	124	13%	2.5	1.0										
encJpeg	Image compression		N	N	62	8	9	1.6	54	121	1%	2.8	1.6	10%	4%								
h263dec	Video decoder		N	N	54	5	3	3.0	165	212	10%	4.0	2.2	3%									
mpegVideo	Video decoder		N	N	69	8	9	1.4	23	701	47%	4.1	3.7										
mp3	mp3 decoder		N	N	98	6	17	2.3	55	181	14%	3.1	1.4	6%			2%			8%			

benchmark run.

The results show relatively good speculative performance for selected STLs. Overall, TLS execution characteristics like average thread size, number of threads per loop entry, and speculative buffers required (columns g, h, j & k) vary widely from program to program. The average thread size for most benchmarks is at least a hundred or more cycles. Given that our experiments were conducted using single-issue MIPS cores, the average size of threads is sufficiently large to suggest that programs may further benefit from superscalar cores that can exploit instruction-level parallelism relatively independent of the coarser-grained parallelism being targeted by TLS.

We found the overheads for profiling and dynamic recompilation small, even for the shorter running benchmarks. This is possible due to the low-overhead profiling system and relatively small amount of collected profiling information required to make reliable STL choices. In our benchmarks, selected STLs varied little with the amount of profiling information collected once enough data was collected to overcome local variations in RAW violations, buffer overflows and thread sizes. This is due to the fact that most selected STLs were invariant to the input data set and the input data sets were stable for the duration of the benchmark. For benchmarks with STLs sensitive to input data set sizes, changing in mid-execution to a larger input data set could cause frequent speculative buffer overflows. For these benchmarks, a potential solution would be trigger reprofiling and recompilation when a selected STL consistently experiences unexpected buffer overflows.

Table 3 shows the effect of optimizations and improvements that impact all STLs. Reduced overheads improve speculative performance more than 5% on 10 applications. Loop invariant register allocation only improves performance 2-4% for five applications. The effect of the non-communicating loop inductor is not shown because without this critical optimization, performance suffers far too much to make a meaningful comparison. While similar benefits of optimizing inductor variables in speculative loops were shown by [11][41][45], our approach has advantages. Our scheme suffers no overhead for communicating inductors because they are computed locally, and is simpler because it does not require significant hardware support or instruction scheduling.

Table 3 shows the effect of specialized compiler optimizations and VM modifications on the benchmarks. VM modifications appear to have a more significant impact. Parallelizing memory allocator access and removing synchronized object locks during speculation significantly affects performance on six benchmarks.

In general, the opportunities to apply compiler optimizations are limited to specific STLs in integer programs, but the sum impact is significant. The resetable non-communicating loop inductor dramatically improves BitOps due to the removal of a loop-carried dependency from the relatively small threads in its critical STL. Multilevel STL decompositions improve mp3 and, to a much lesser degree, Assignment. Thread synchronizing lock prevents performance-degrading violations on monteCarlo and db. The opportunity to apply this optimization is less than we hoped for, but the limited impact seems consistent with other studies that have looked at techniques for minimizing violations [11][41]. The only compiler optimization that seems to have little effect is hoisting startup and shutdown handlers. This optimization would likely be most useful in a nested loop with small number of iterations per entry into the loop. The two loops in NeuralNet that use this optimization do not have the ideal characteristics, and only benefit slightly from it.

Also included in Table 3 is the effect of manual transformations to improve speculative performance. For six of the integer benchmarks, some programmer assistance is needed to expose parallelism. The transformations were achieved with the assistance of TEST feedback. TEST profiling results that summarized critical potential STLs and associated *dependency arc* frequencies and lengths facilitated quick identification of performance bottlenecks in the source code. The resulting transformations, listed in Table 4, significantly improve performance and do not slowdown the original sequential execution. Only three of these benchmarks require significant manual transforms, while the other three need only trivial modifications. Examination of the program sources suggest most of these modifications can

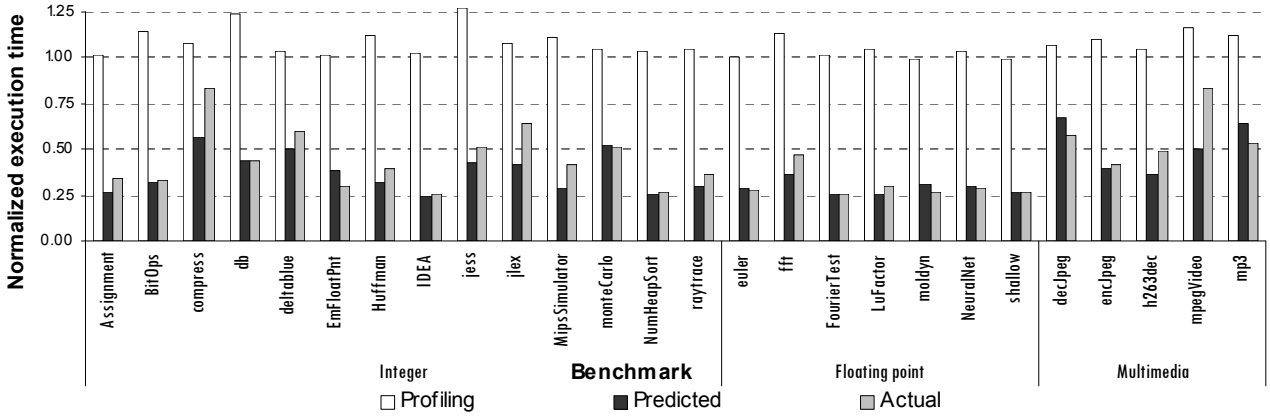


Figure 8 – Shows simulation results of slowdown during profiling, predicted TLS performance, and actual TLS performance on the Hydra (4 CPUs) normalized to execution time of the original sequential program.

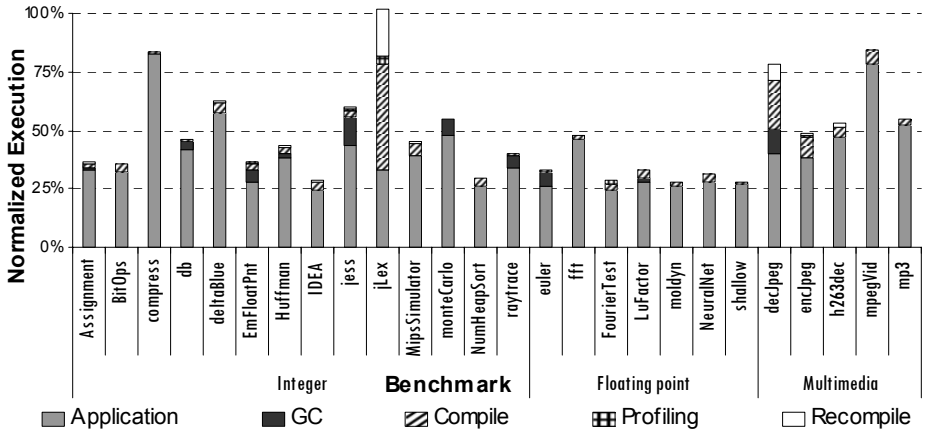


Figure 9 – Total program speedup with compilation, garbage collection, profiling, and recompilation overheads.

not be performed automatically because they require high-level understanding of the program.

Table 4 – Difficulty and potential for compiler automation of manual transformations performed that improve speculative performance.

Benchmark	Difficulty	Compiler optimizable	Lines modified	Modified operations
NumHeapSort	Low	N	7	Remove loop carried dependency at top of sorted heap
Huffman	Med	N	22	Merge independent streams to prevent sub-word dependencies during compression. Guess next offset when uncompressing data stream.
MipsSimulator	Med	N	70	Minimize dependencies for forwarding load delay slot value
db	Low	Y	4	Schedule loop carried dependency
compress	Low	N	13	Guess next offset when compressing/uncompressing data
monteCarlo	Med	N	39	Schedule loop carried dependency

Overall, the results show TEST analysis does a reasonable job of predicting speculative performance and can effectively choose STLs that result in good speedups, but differences between predicted and actual speculative execution were found. To further understand the discrepancy between predicted and actual TLS performance from Figure 8, we looked at time spent in various states during speculative execution, shown in Figure 10. The primary differences appear to be from effects that cannot be accounted for when predicting speculative performance.

In EmFloatPnt, fft, jLex, and MipsSimulator, significant processor time is spent in *wait-used state*. For fft, this comes from buffer overflow stalls for large STL iterations. While TEST analysis properly accounts for overflow, its effects are predicted to be smaller because the accumulated statistics make iterations that overflow appear to be smaller. For EmFloatPnt, jLex, and MipsSimulator, *wait-used state* is due to load imbalance. Since speculative thread iterations must commit in order, load imbalance forces speculative processors to wait if iterations ahead of it have still not completed. Predicted speculative performance does not account for *wait-used state* because of the averaging effects of accumulating counters.

Over ten applications, particularly compress, Huffman, and mpegVideo, have significant amounts of *run-violated* and *wait-violated state* that significantly degraded performance. Since we derive predicted speculative performance by computing parallelism from ideal scheduling of inter-thread dependencies, our analysis does not account for the cost of violations. In actual speculative execution, violations near the end of a speculative thread can cause a thread to restart and re-execute all its work, thereby losing any parallelism that may have existed. Examinations of program sources suggest that thread violations in these benchmarks are truly dynamic and cannot be mitigated by thread synchronization or value prediction.

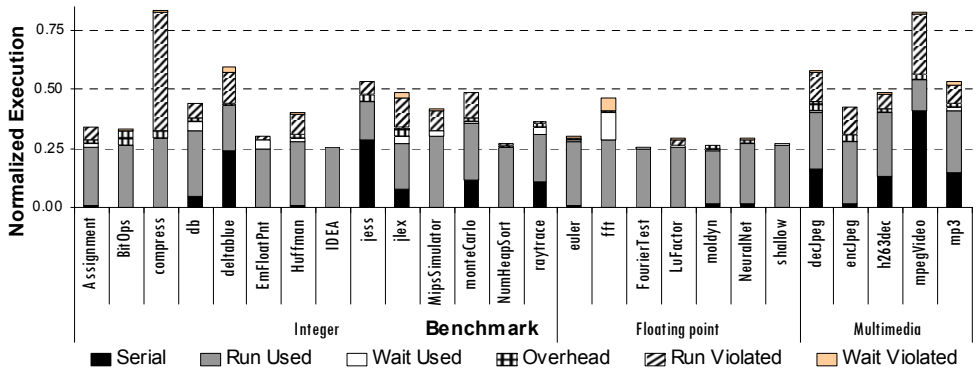
7. Related Work

The Multiscalar paradigm [15] was the first complete description and evaluation of an architecture with TLS support. Several other architectures for TLS using CMPs have been proposed [10][21][29][40][42]. These implementations have mostly targeted coarser grains of granularity than the Multiscalar architecture. In a similar vein, software-based dynamic dependence detection has been proposed for traditional multiprocessor systems as a way to preserve correctness for loops executed in parallel that may have complex dependency patterns [18][36][37][39].

There are numerous commercial and research compilers based on array dependence analysis for parallelizing Fortran programs [2][5][20][38]. Several studies have looked at how these compilers might be applied to general programs [4][25][35].

There is a growing body of related work on TLS compilation addressing different architectures and aspects of the problem. The Multiscalar [43] compiler focuses on compile-time heuristics to increase intra-procedural task sizes and intra-task dependency scheduling to increase task parallelism. These optimizations are sufficient for the small threads targeted by Multiscalar, but do not address the memory disambiguation and decomposition selection difficulties of compiling for coarser-grained TLS systems.

A study of TLS limits by Oplinger et al. [34] and



State	Meaning
Serial	Time spent not running speculatively
Run-used	Committed speculative CPU time used for application computation
Wait-used	Committed speculative CPU time waiting to become head or stalling due to buffer overflow
Overhead	TLS startup, eoi, restart and shutdown handler overheads
Run-violated	Speculative CPU time discarded due to RAW violation used for application computation
Wait-violated	Speculative CPU time discarded due to RAW violation spent waiting to become head or stalling due to buffer overflow

Figure 10 – Breakdown of actual speculative execution by time spent in various states.

Steffan and Mowry's work on the Stampede TLS processor [40] have, like us, used simulation and profiling to identify good thread decompositions. These studies used cycle-accurate CPU simulators that analyze dependencies of all executing memory references to deduce performance estimates of potential STLs. Neither study specifically addressed how their technique could be used for automatic speculative compilation or how their analysis could have been performed without significant simulation and profiling overheads.

There has also been research on improving the performance of STLs once they are selected. Several studies [11][33][41] showed that minimizing inter-thread violations using hardware synchronization and value prediction can significantly improve speculative performance. Zhai et al. [45] found that compiler scheduling to increase distances between inter-thread dependencies is most beneficial for minimizing synchronization stall time of induction variables, but scheduling of other inter-thread dependencies did not improve performance significantly.

Tremblay et al. previously proposed a JVM system with speculative thread support [7][42], but there are significant differences from Jrpm. Their CMP, the MAJC processor, has only two processors and only supports TLS with inefficient software routines. They also do not propose how speculation can be applied automatically to programs.

There has been other related research that has considered how feedback can be used to dynamically parallelize programs. Ko et al. [26] identified optimal decompositions through brute-force incremental execution of all possible decompositions in multilevel parallel programs. Numerous systems have also been designed to manually tune parallel performance on traditional multiprocessors [1][19][27][30], but they have relied on off-line, not real-time, dependence analysis of memory traces.

8. Conclusions

Our paper describes the Java runtime parallelizing machine (Jrpm), a complete system that automatically parallelizes sequential programs using thread-level parallelism. This system uses a chip multiprocessor, thread-level speculation, hardware profiling, and dynamic compilation together to enable a different approach to automatic parallelization for small-scale chip multiprocessors. Simulation results demonstrate Jrpm's ability to automatically select and optimize appropriate thread decompositions with minimal effort from the programmer. On our CMP with four processors, we achieve speedups of 3 to 4 on floating point applications, 2 to 3 on multimedia applications, and between 1.5 and 2.5 on integer applications.

Future work will involve trying more applications out on Jrpm and further investigation on how much profiling is needed before recompilation into speculative threads. We are also looking into new techniques to improve performance on applications with significant dynamic violation frequencies that cannot be minimized by thread synchronization or value prediction.

9. Acknowledgements

This work was supported by DARPA Air Force Contract F29601-01-2-0085 and NSF Grant CCR-0220138.

10. References

[1] Adve, V. S. et al. An integrated compilation and performance analysis environment for data parallel programs. In SC'95, San Diego, CA, November 1995.

[2] Adve, V. S. et al. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In SC'98, Orlando, FL, November 1998.

[3] Allen, R and Kennedy, K. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[4] Artigas, P. et al. Automatic Loop Transformations and Parallelization for Java. In ICS'2000, Santa Fe, NM, May 2000.

[5] Blume, W. et al. Polaris: Improving the Effectiveness of Parallelizing Compilers. In 7th International Workshop on Languages and Compilers for Parallel Computing. Ithaca, NY, August 1994.

[6] Broadcom Corporation. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[7] Chaudhry, S. et al. Space-Time Dimensional Computing for Java

Programs on the MAJC Architecture. In Java Microarchitectures, Kluwer Academic Publishers, Boston, MA, April 2002.

[8] Chen, M. and Olukotun, K. Targeting Dynamic Compilation for Embedded Environments. JVM'02, San Francisco, CA, August 2002.

[9] Chen, M. and Olukotun, K. TEST: A Tracer for Extracting Speculative Threads. In CGO'03, San Francisco, CA, March 2003.

[10] Cintra, M., Martinez, J. F., and Torrellas, J. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In ISCA-27, Vancouver, BC, June 2000.

[11] Cintra, M. and Torrellas, J. Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In HPCA-8, Anaheim, CA, February 2002.

[12] Click, C. High-Performance Computing with the Server Compiler for the Java HotSpot Virtual Machine. In JavaOne 2001, San Francisco, CA, June 2001.

[13] Culler, D.E. et al. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, San Mateo, CA, 1998.

[14] Emer, J. Ev8: The post-ultimate alpha (keynote address). In PACT'01, Barcelona, Spain, September 2001.

[15] Gopal, S. et al. Speculative Versioning Cache. In HPCA-4, Las Vegas, NV, February 1998.

[16] Gosling, J., Joy, B., and Steele, G. The Java Language Specification. Addison Wesley, Reading, MA, 1996.

[17] Griessemer, R. and Mitrovic, S. The Java HotSpot Virtual Machine Client Compiler: Technology and Application. In JavaOne 2001, San Francisco, CA, June 2001.

[18] Gupta, M and Nim, R. Techniques for Speculative Run-Time Parallelization of Loops. In SC'98, November 1998.

[19] Hall, M. W. et al. Experiences using the ParaScope Editor: an interactive parallel programming tool. In PPoPP'93, pages 33-43, May 1993.

[20] Hall, M. W. et al. Maximizing Multiprocessor Performance with the SUIF Compiler. In IEEE Computer, December 1996.

[21] Hammond, L., Willey, M., and Olukotun, K. Data Speculation Support for a Chip Multiprocessor. In ASPLOS-VIII, San Jose, CA, October 1998.

[22] Hennessy, J. L. and Patterson, D. A. Computer Architecture: A Quantitative Approach, Third edition. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 2002.

[23] Jones, R. and Lins, R. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, Chichester, UK, 1996.

[24] Kahle, J. Power4: A Dual-CPU Processor Chip. In Microprocessor Forum '99, October 1999.

[25] Knobe, K. and Sarkar, V. Array SSA form and its use in Parallelization. In PoPL'98, San Diego, CA, January 1998.

[26] Ko, W. et al. Effective Cross-Platform, Multilevel Parallelism via Dynamic Adaptive Execution. In 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments. Ft. Lauderdale, FL, April 2002.

[27] Liao, S. W. et al. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In PPoPP'99, Atlanta, GA, May 1999.

[28] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison Wesley, Reading, MA, 1997.

[29] Marcuello, P. and Gonzalez, A. Clustered Speculative Multithreaded Processors. In ICS'99, Rhodes, Greece, June 1999.

[30] Miller, B. P. et al. The Paradyn Parallel Performance Measurement Tools. In IEEE Computer, 28(11):37-46, November 1995.

[31] Muchnick, S. Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[32] Olukotun, K. et al. The case for a single chip multiprocessor. In ASPLOS-VII, Cambridge, MA, October 1996.

[33] Olukotun, K., Hammond, L., and Willey, M. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In ICS'99, Rhodes, Greece, June 1999.

[34] Oplinger, J. T., Heine, D. L., and Lam, M. S. In Search of Speculative Thread-Level Parallelism. In PACT'99, Newport Beach, CA, October 1999.

[35] Wu, P. and Padua, D. Containers on the Parallelization of General-purpose Java Programs. In PACT'99, Newport Beach, CA, October 1999.

[36] Rauchwerger, L. and Padua, D. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In PLDI'95, La Jolla, CA, June 1995.

[37] Saltz, J., Mirchandaney, R., and Crowley, K. Runtime parallelization and scheduling of loops. In IEEE Transaction on Computers, 40(5):603-612, May 1991.

[38] Sarkar, V. The PTRAN Parallel Programming System. In Parallel Functional Programming Languages and Compilers, ACM Press Frontier Series, pages 309-391, 1991.

[39] So, B., Moon, S., and Hall, M. W. Measuring the Effectiveness of Automatic Parallelization in SUIF. In ICS'98, Melbourne, Australia, July 1998.

[40] Steffan, J. G. et al. A Scalable Approach to Thread-Level Speculation. In ISCA-27, Vancouver, BC, June 2000.

[41] Steffan, J. G. et al. Improving Value Communication for Thread-Level Speculation. In HPCA-8, Cambridge, MA, February 2002.

[42] Tremblay, M. MAJC: Microprocessor Architecture for Java Computing. In HotChips'99, Stanford, CA, August 1999.

[43] Vijaykumar, T. N. and Sohi, G. S. Task Selection for a Multiscalar Processor. In MICRO'98, Chicago, IL, August 1998.

[44] Wall, D. Limits of Instruction-Level Parallelism. DEC Western Research Lab WRL-TN-15, December 1990.

[45] Zhai, A. et al. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In ASPLOS-X, San Jose, CA, October, 2002.