The Jrpm System for Dynamically Parallelizing Sequential Java Programs

AS INSTRUCTION-LEVEL PARALLELISM WITH A SINGLE THREAD OF CONTROL APPROACHES ITS PERFORMANCE LIMITS, DESIGNERS MUST FIND OTHER ARCHITECTURAL IMPROVEMENTS TO SPEED UP PROGRAM EXECUTION. THE JRPM SYSTEM TAKES ADVANTAGE OF RECENT DEVELOPMENTS TO ENABLE A NEW APPROACH TO AUTOMATIC PARALLELIZATION. JRPM CAN EXPLOIT THREAD-LEVEL PARALLELISM WITH MINIMAL PROGRAMMER EFFORT.

Michael K. Chen
Kunle Olukotun
Stanford University

The quest to automatically parallelize general-purpose programs is a long-standing problem in the microarchitecture community. Solving this problem is critical to continued architectural improvements in microprocessor performance, as instruction-level parallelism with a single thread of control approaches its performance limits. We've developed a solution that relies on four key technologies: chip multiprocessors, thread-level speculation, dynamic compilation, and hardware-based profiling. We can combine these technologies and manage them inside a Java virtual machine (JVM). The resulting system, the Java runtime parallelizing machine (Jrpm), can parallelize a wide range of integer, multimedia, and floating-point Java benchmarks with excellent performance.

Jrpm is a complete system for parallelizing sequential programs automatically for thread-level parallelism. This dynamic parallelization system overcomes the limitations of conventional parallelizing compiler and multiprocessor technology.

The parallelization problem

In wide-issue superscalar and very-long-instruction-word processors, instruction-level parallelism faces diminishing returns (from one to 10s of instructions). Processor architects are exploring coarser granularities, such as fine-grained thread-level parallelism (from 10 to thousands of instructions) to speed up sequential-program execution. Multiprocessor architectures and parallelizing compilers have both been around for some time now, but neither is effective at exploiting thread-level parallelism automatically.

Traditional small-scale multiprocessors (from two to 16 processors) have effectively exploited very coarse-grained parallelism (greater than tens of thousands of instructions). Unfortunately, current multiprocessor architectures must communicate dependencies throughout the multiple layers of memory hierarchy. This generates interprocessor communication latencies greater than hundreds of CPU cycles that completely eliminate potential speedups from finer-grained parallel tasks.

Published by the IEEE Computer Society

Traditional multiprocessors must also handle synchronization overheads for mutual exclusion and event synchronization. Conservative synchronization preserves program correctness, but too much synchronization can greatly degrade multiprocessor performance. This can be a serious problem for automatic program parallelization using parallelizing compilers.

Traditional parallelizing compilers use array data-dependence analysis.[1] Data dependence analysis determines dependence relationships for pairs of array references in a program. A compiler can use these results to reorder the program to exploit coarse-grained parallelism on a multiprocessor while correctly generating the same results as the original program. Such compilers have successfully parallelized Fortran-like numerical applications (which have considerable regular, well-structured parallelism) on traditional multiprocessors.

Unfortunately, data dependence analysis is often complex and expensive. Furthermore, general integer programs have characteristics such as complex control flow, irregularly structured loops, and significant pointer use that make them unsuitable for automatic compiler parallelization. These characteristics ultimately cause dependence analysis to return imprecise dependency information for reference pairs, forcing the insertion of conservative, performance-degrading synchronization into the generated code to safely handle potential dependencies.

Jrpm approach

Jrpm is a dynamic parallelization system that overcomes the difficulties of applying current technologies and approaches for the automatic parallelization of general programs. Jrpm parallelizes programs with almost no input from the user or programmer. A custom runtime system with special hardware support analyzes dynamic execution for parallelism and correctly handles dynamic dependencies. Figure 1 shows the system's key components:

- *Chip multiprocessor.* Jrpm is based on the Hydra chip multiprocessor (http://www-hydra.stanford.edu).[2] Decreasing feature size and increasing transistor counts make chip multiprocessors possible.[3-6] Chip multiprocessors combine several processors onto one die with a tightly coupled memory interface. In this configuration, interprocessor sharing and communication costs are low, making fine-grained thread-level parallelism plausible.
- *Thread-level speculation.* Hydra supports TLS,[2,7-9] which allows arbitrary division of a sequential program into threads for parallel execution while still ensuring that memory accesses between threads maintain the original sequential program order. Thus, TLS enables aggressive parallelization relative to traditional multiprocessors.
- *Hardware profiler.* Static parallelizing compilers have insufficient information to analyze dynamic dependencies effectively. Dynamic analysis to find parallelism complements a TLS processor's ability to parallelize optimistically and to use hardware to guarantee correctness. TEST (Tracer for Extracting Speculative Threads) support analyzes sequential-program execution in real-time to find the best regions to parallelize with minimum hardware support.[10]
- *Virtual machine.* Virtual machines such as Sun's JVM and Microsoft's .NET VM have become commercially popular for supporting platform-independent applications. In our system, the JVM acts as an abstraction layer that hides the dynamic analysis framework and thread-level speculation from the program, letting us seamlessly support a new execution model without modifying the source binaries.

Following Figure 1, the compiler derives a control flow graph (CFG) from program bytecodes and analyzes it to identify potential thread decompositions.[11] A single Hydra processor executes, as a sequential program, a program that has been dynamically compiled with instructions annotating local variables and possible thread decompositions. Trace hardware collects statistics in real time for the prospective decompositions. Once this hardware has collected sufficient data, the dynamic compiler recompiles into speculative threads those regions predicted to have the largest speedup and most coverage.

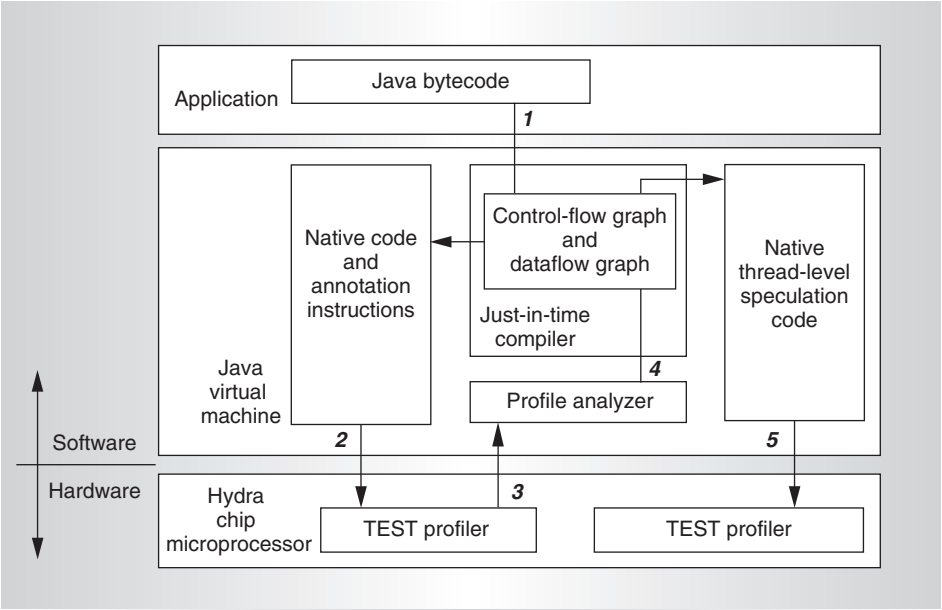

Figure 1. Overview of the Jrpm system, including hardware and software components. Programs running on Jrpm execute the following steps: 1. Identify thread decompositions by analyzing bytecodes, and compile natively with annotation instructions. 2. Run annotated program sequentially, collecting TEST (Tracer for Extracting Speculative Threads) profile statistics on potential thread decompositions. 3. Postprocess profile statistics and choose thread decompositions that provide the best speedups. 4. Recompile code with thread-level speculation (TLS) instructions for selected thread decompositions. 5. Run native TLS code.

Although the primary goal of our dynamic parallelization system is to automatically speed up program execution, the system also benefits from additional properties that are attractive to both programmers and system designers:

- *Reduced programmer effort.* Manually identifying fine-grained parallel decompositions can be time-consuming, especially for programs without obvious critical sections. Because Jrpm automatically selects and guarantees the correct behavior of executing parallel threads, programmers can focus on performance debugging instead of the usual complexities of parallel programming.
- *Portability.* Jrpm works with unmodified sequential-program bytecodes. Because the system doesn't modify the binaries explicitly for TLS, the code retains its platform independence.
- *Retargetability.* Because parallel decompositions are not explicitly coded, Jrpm can dynamically adapt decompositions at runtime for future chip multiprocessors with more processors, larger speculative buffers, or different cache configurations.
- *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 from both floating-point and difficult-to-analyze integer applications.

Chip multiprocessor with TLS support

Hydra,[2] shown in Figure 2, is a chip multiprocessor consisting of four single-issue, pipelined MIPS processors, each with private L1 data and instruction caches. High-speed, low-latency read and write buses make thread-level parallelism practical, even with substantial interprocessor sharing. An integrated, on-chip, shared L2 cache minimizes cache misses when processors work on shared data.

TLS allows the division of a sequential program into threads for parallel execution. Although speculative threads can include

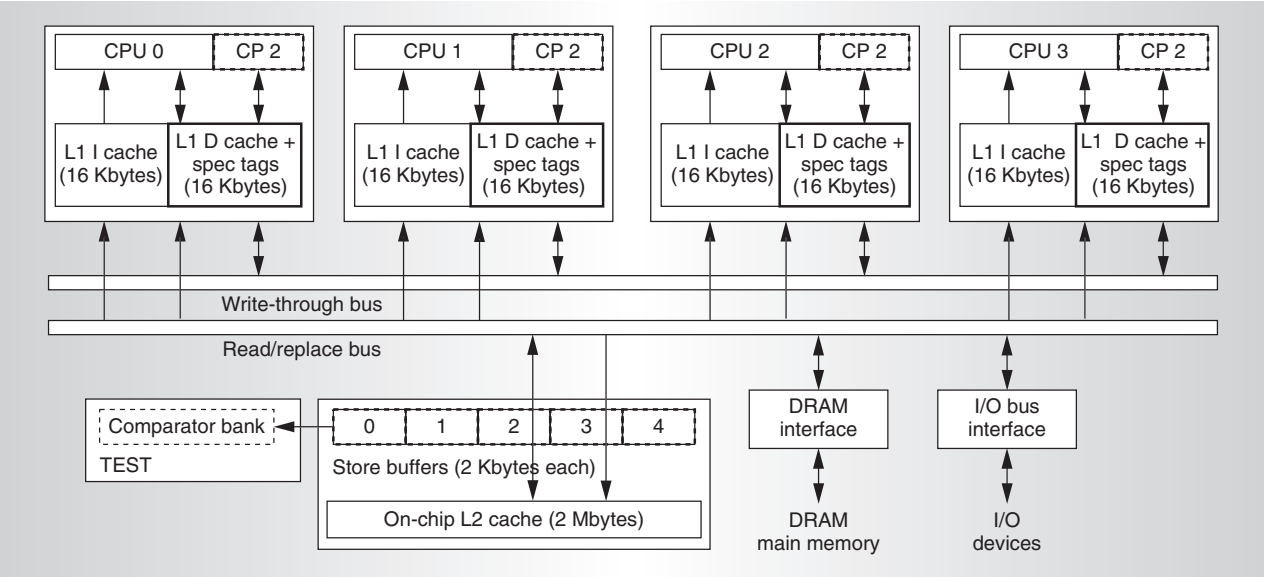

Figure 2. Hydra chip multiprocessor. Bold lines indicate additions for TLS support; dotted lines indicate additions for TEST profiling; CP means coprocessor.

loops, method call returns, or general regions,[2] most research has focused on decompositions based on loops. As Figure 3 (next page) shows, a speculative thread loop (STL) is a loop decomposed into threads, with one loop iteration assigned to each thread. Data speculation hardware ensures that the threads maintain the original sequential program's memory ordering.

The STL model designates the CPU executing the logically earliest thread as the nonspeculative head CPU. The model assigns logically later iterations, in order, to the other CPUs for speculative execution. Once the head CPU completes its iteration, it commits its state and starts speculatively executing the next unassigned iterations; the CPU executing the logically next iteration then becomes the head CPU.

Speculative-thread support in Hydra consists of

- a coprocessor (CP) in each CPU with extra registers, logic, and instruction support to control thread speculation;
- extra speculative tag bits added to the processor L1 data caches to detect interthread data dependency violations; and
- store buffers attached to the secondary cache to hold speculative data until a speculative thread can either safely commit them to the secondary cache or discard them.[2]

A running application controls TLS through instruction-set-architecture extensions and special stores issued onto the write bus. For a loop that has been transformed into speculative threads (as in Figure 3), overheads occur at the start and end of speculation, at the end of every iteration, and on dynamic read-after-write (RAW) violations.

Tracer for extracting speculative threads

TLS simplifies many automatic parallelization challenges, but we had to consider certain constraints when selecting regions for this execution model. With the Hydra chip multiprocessor, the major constraints are as follows:

- True interthread data dependencies, or read-after-write hazards, always limit speedup from parallel execution of speculative threads.
- Speculative read and write states buffered by the hardware cannot be discarded during speculative execution and must fit into the on-chip hardware structures. Attempts to drop an L1 cache line with speculative state or to write to a full store buffer will force a speculatively executing

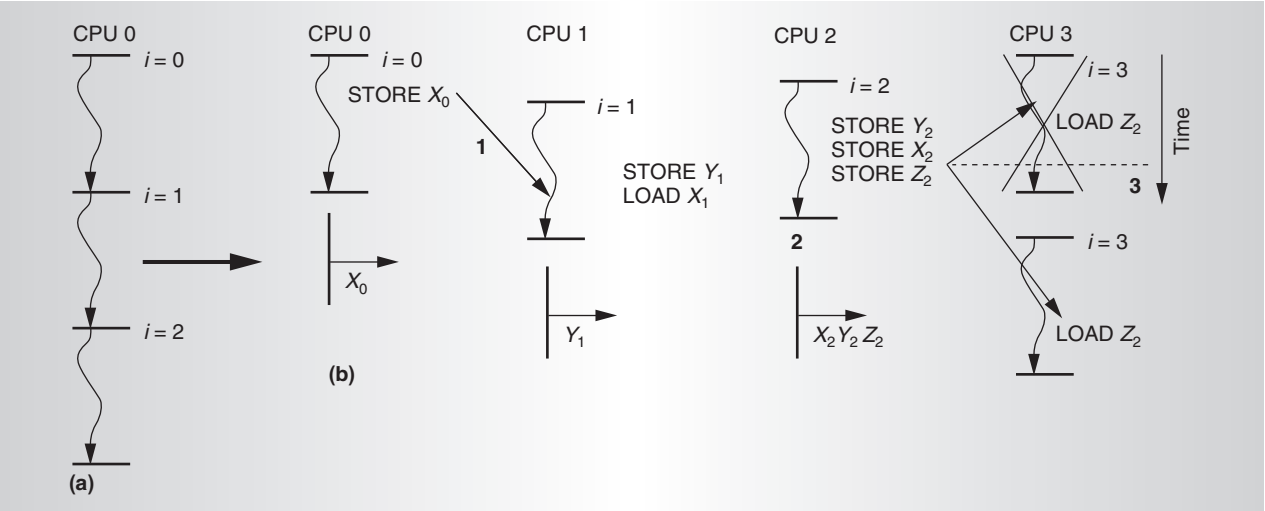

Figure 3. Sequential loop (a) transformed into a speculative thread loop (b) that uses speculative hardware to maintain the following memory orderings: 1. For variable *X*, buffered writes are available only to sequentially later threads, preventing write-after-read violations. 2. For variable *Y*, committing buffered speculative writes in threads according to the original program order maintains write-after-write ordering. 3. For variable *Z*, forwarding data to sequentially later threads, and forcing a thread to restart when a speculative thread reads a value too early and violates sequential memory ordering, guarantees correct read-after-write ordering.

thread to stall until the thread becomes the nonspeculative head thread and safe execution of a load or store is possible.
- Only one thread decomposition (for example, one loop in a loop nest) can be active at a time.
- Compiled speculative thread code introduces sequential overheads from speculative-thread-management routines and forced communication of interthread-dependent local variables, limiting speedups under TLS for very small threads (say, with less than 10 instructions).[2,9]

These constraints impose conflicting requirements for selecting thread decompositions. Speculating on small loops limits parallel coverage and suffers from higher speculative-thread overheads relative to the work performed. Speculating on large loops increases the probability of speculation buffer overflows and could incur higher relative dependency-violation penalties.

Dynamic analysis to identify STLs complements a TLS processor's ability to parallelize optimistically and to use hardware to guarantee correctness. The primary goal here, unlike with traditional parallelizing-compiler analysis, is to identify where parallelism usually exists rather than where it is guaranteed to exist. Profiling can provide accurate statistics on dynamic dependency behavior, thread size, and buffer requirements for most types of programs.

Analysis overview

The compiler examines a method's CFG to identify all natural loops that could be a potential STL.[11] Two types of trace analyses characterize an STL's potential: load dependency and speculative-state overflow.

By examining executing loads and stores, load dependency analysis looks for interthread dependencies for an STL. TEST records the time stamp when a memory or local-variable store occurs; on subsequent loads to the same address, TEST retrieves this time stamp. By comparing this value with the thread-start time stamp, it is possible to detect the frequency of interthread dependency arcs and identify critical arcs. (A critical arc is the shortest dependency arc that limits parallelism between a given pair of threads.)

Speculative-state-overflow analysis checks that the speculative state generated by an iteration of an STL will fit within the limits of the L1 caches and store buffers. TEST maintains a history of cache lines accessed by loads and stores. Later, we can determine if subse-

quent accesses to the same cache line will require allocating a new buffer state to the current speculative thread. By maintaining counters tracing these requirements, we can estimate how frequent a given STL will overflow its speculative buffer limits.

Once TEST has collected enough profiling data (for example, at least thousands of iterations of an STL under analysis), it computes the estimated speedup for an STL from the dependency arc frequencies, thread sizes, critical arc lengths, overflow frequencies, and speculative overheads. Using statistics from the two analyses and the computed speedup, Jrpm considers for recompilation into speculative threads only those loops that have average iterations per entry far greater than 1, speculative buffer overflow frequency far less than 1, predicted speedup greater than 1.2, and coverage greater than 0.5 percent during sequential execution. It is often possible to choose multiple decompositions in a loop nest. In this case, Jrpm selects the best STL by comparing the estimated execution time for the different STL decompositions.

Hardware-software support for TEST

The hardware we designed to minimize profiling overheads and improve accuracy analyzes a sequentially executing program and works when speculation is disabled.

Annotation instructions that the dynamic compiler inserts into native code mark important events relevant to trace analyses. Annotations mark a potential STL's entry, exit, and iteration end. TEST uses explicit annotations to track local variables in the same calling context as a potential STL that could cause dependencies. This simplifies the tracking of these variables in optimized compiled code. A processor automatically communicates memory load and store events to the tracing hardware when tracing is enabled. At the end of an STL (for example, an exit from a loop), special routines read the collected statistics from TEST for use by the runtime system.

The annotation instructions communicate events to the comparator banks. The comparator banks carry out the bulk of the dependency and overflow trace analyses. One comparator bank tracks the progress for a given STL. Each bank, primarily comprising comparators and counters, analyzes and collects statistics on incoming loads and stores. An array of comparator banks allows the tracing of multiple potential STLs that execute concurrently, as in nested loops. Our calculations suggest that an implementation of the TEST hardware with eight comparator banks would add less than 1 percent to the transistor count of the Hydra chip multiprocessor with TLS support.

The speculative store buffers, which are idle during sequential nonspeculative execution, hold a history of previous time stamp events during profiling. The buffers retrieve an address' time stamp on an annotating memory or local variable instruction for use in the comparator banks. The store buffers, organized as first-in, first-out (FIFO) buffers during tracing, effectively hold a limited history of memory and local-variable accesses.

Compiling selected regions into speculative threads

Jrpm's Java runtime system is based on the open-source Kaffe virtual machine (http://kaffe.org), but we used our own just-in-time compiler, microJIT, and a garbage collector to make up for the original virtual machine's performance limitations. We augmented the microJIT compiler to generate speculative thread code. The dynamic compiler inserts speculative-thread-control routines into the STLs chosen by TEST analysis. In addition to the fixed speculative-handler overheads, additional overheads are possible in certain circumstances. The master processor must communicate STL initialization values to the slave processors by saving them to the runtime stack. Certain optimizations must insert cleanup code at the entry and exit of STLs. Furthermore, the compiler must force local variables that could cause interthread (loop-carried) dependencies in an STL to communicate through loads and stores in a runtime stack shared between all speculative processors.

When possible, Jrpm's dynamic compiler automatically applies optimizations to improve speculative performance for selected STLs. Table 1 summarizes these compiler optimizations.

Parallelizing real programs using Jrpm

Table 2 summarizes the characteristics of the STLs automatically chosen from TEST analysis. Overall, we found significant diver-

Table 1. Summary of low-level TLS compiler optimizations.

Optimization	Function	Benefit	Cost
Loop-invariant register allocation	Register allocates memory load that always returns the same value	Eliminates redundant memory load per iteration	Load of value into the register at init and restart
Noncommunicating loop inductor	Locally computes loop inductor value for a thread	Eliminates frequent RAW violations for loop inductors incremented at end of iteration	Computation of inductor value at init and restart
Resetable loop inductor	Locally computes loop inductor-like value for a thread	Eliminates frequent RAW violations for loop inductor-like values	Computation of loop inductor-like value at init and restart
Reduction	Computes associative operations locally	Eliminates dependencies for associative operations	Merge locally computed values for final reduction value at exit
Synchronizing lock	Protects loop-carried dependencies from spurious RAW violations	Eliminates RAW violations from frequent dependencies	Wait and signal overhead of the lock for every thread
Multilevel decompositions	Switches selected STLs between an outer and inner loop in a nested loop	Improves load balancing for irregularly structured nested loops	Init and exit overhead for switching between STL decompositions

sity in the coverage of selected STLs. Although 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 than those shown in the table, but the omitted decompositions do not have any significant coverage. The mp3, db, jess, and DeltaBlue benchmarks have significant sections of serial execution that are not covered by any potential STLs, limiting the total speedup for these applications.

These benchmarks come from the jBYTEmark (http://www.byte.com), SPECjvm98 (http://www.specbench.org/jvm98/), and Java Grande (http://www.epcc.ed.ac.uk/javagrande/) suites, as well as real applications found on the Internet.

TLS can simplify program parallelization, but not all programs can benefit from it. Some integer benchmarks evaluated using TEST show no potential for speedup using speculation. Programs with system calls in critical code do not speed up on Jrpm, because our implementation of TLS cannot handle system calls speculatively. Several other integer programs contain only loops that consistently overflowed the speculative state, executed too few iterations for speculation, or contained an unoptimizable serializing dependency.

The larger programs contain so many loops that manual identification of STLs would have been too time-consuming. A visual analysis of the source code revealed that a traditional parallelizing compiler could analyze less than half the benchmarks.

Performance results

We ran each benchmark as a sequential annotated program on Jrpm with the TEST profiling system enabled. The dynamic compiler then recompiled the benchmark and executed it using speculative threads with the STLs selected by TEST. Figure 5 shows slowdown during profiling, the predicted TLS execution time from TEST analysis, and actual TLS performance. Figure 6 compares total program speedup (adding compilation, garbage collection, profiling, and recompilation overheads) normalized with respect to normal serial execution (including compilation and garbage collection overheads) for a given benchmark run.

During profiling, most benchmarks experience no more than a 10 percent slowdown, and only two applications have slowdowns approaching 25 percent, as Figure 5 shows. These slowdowns are reasonable, especially considering the relatively short period of time that most programs must spend on profiling to select an STL.

Simulations of this system show that our approach has significant potential for automatically exploiting thread-level parallelism. From our wide set of Java benchmarks, we can exploit thread-parallelism in integer, floating-point, and multimedia benchmarks. The best speedups, approaching 4×, occur with the floating-point applications. The speedups achieved on multimedia and integer programs are also significant, between 1.5× and 3×, but vary widely and are generally less than those achieved for floating-point applications.

Overall, TLS execution characteristics such as average thread size and number of threads per loop entry (see Table 2) vary widely from program to program. Despite this, the average thread size for most benchmarks is at least 100 cycles. We conducted our experiments using single-issue MIPS cores. The average thread size appears large enough to suggest that programs could benefit further from superscalar cores that exploit instruction-level parallelism relatively independent of the coarse-grained parallelism that TLS targets.

The overheads for profiling and dynamic recompilation are small, even for the shorter-running benchmarks. Contributing factors include the low-overhead profiling system, the limited profiling information required to make reliable STL choices, and the small amount of code that must be recompiled to transform a loop. In our benchmarks, selected STLs vary little with the amount of profiling information collected, once TEST collects enough data to overcome local variations in RAW violations, buffer overflows, and thread sizes. The reason for this stability is that most selected STLs are invariant to the input data set. For benchmarks with STLs sensitive to the input data set, the input data sets remain stable for the duration of the benchmark. In real-world cases, in which the input data sets can change during runtime, Jrpm could trigger reprofiling and recompilation when a selected STL sensitive to the input data set consistently experiences unexpected behavior.

We also measured the effect of optimizations and improvements that impact all STLs. We found that the reduction in overheads improves speculative performance more than 5 percent on 10 applications. Loop-invariant register allocation improves performance only 2 percent to 4 percent for five applications. In addition, without the noncommunicating loop inductor, performance generally suffers far too much to make meaningful comparisons.

We had to resolve several correctness or performance bottleneck issues in interfacing the SVM and the Hydra CMP with TLS support. These modifications have a more significant effect on benchmark performance than specialized compiler optimizations. Parallelizing memory allocator access and removing synchronized object locks during speculation significantly affects performance on six integer benchmarks. In general, the opportunities to apply specialized compiler optimizations are limited to specific STLs in integer programs, but the cumulative impact of the optimizations is significant.

Table 2. Description and characteristics of integer benchmarks evaluated on the Jrpm system.

Benchmark	Description	Loop count	No. of selected loops
Assignment	Resource allocation	32	11
BitOps	Bit array operations	4	2
compress	Compression	28	4
db	Database	37	6
deltaBlue	Constraint solver	22	5
EmFloatPnt	Floating-point emulation	7	1
Huffman	Compression	14	6
IDEA	Encryption	2	1
jess	Expert system	134	4
JLex	Lexical analyzer generator	128	7
MipsSimulator	CPU simulator	19	2
monteCarlo	Monte Carlo simulation	15	5
NumHeapSort	Heap sort	5	2
raytrace	Ray tracer	14	1
euler	Fluid dynamics	32	13
fft	Fast Fourier transform	5	2
FourierTest	Fourier coefficients	2	1
LuFactor	LU factorization	13	7
moldyn	Molecular dynamics	8	1
NeuralNet	Neural net	19	8
shallow	Shallow water simulation	11	3
decJpeg	Image decoder	61	21
encJpeg	Image compression	62	9
h263dec	Video decoder	54	3
mpegVideo	Video decoder	69	9
mp3	Mp3 decoder	98	17

* Average loop height is the number of nested loops between a selected loop and the innermost loop.

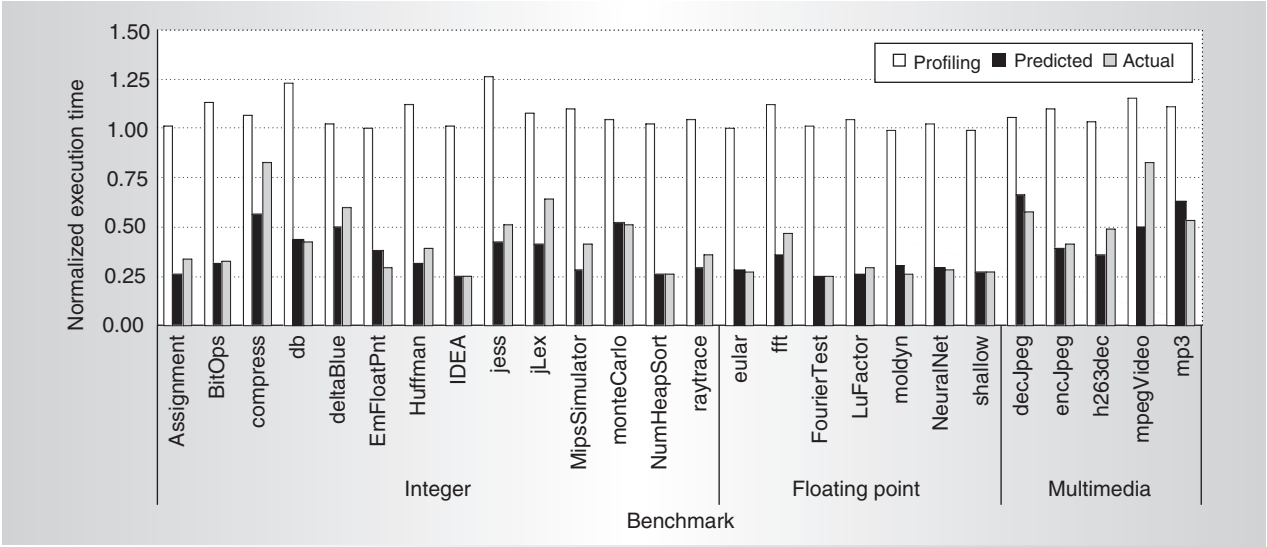

Figure 5. Simulation results of slowdown during profiling, predicted TLS performance, and actual TLS performance on Hydra (four CPUs) normalized to the original sequential program's execution time.

Future work on Jrpm will focus on three areas: dynamic reoptimization of running applications, performance enhancements on applications that currently perform below expectations using TLS, and scaling of the system to work on larger multiprocessor systems (more than four CPUs). We are also looking at running additional applications on Jrpm to further demonstrate the system's ability to speed up a wide variety of programs. MICRO

Acknowledgments

We thank Lance Hammond for his valuable feedback and simulator support, and Tim Wilkinson for his support of the Kaffe virtual machine. US Defense Advanced Research Projects Agency Air Force contract F29601-01-2-0085 and National Science Foundation grant CCR-0220138 provided research funding.

References

1. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, Morgan Kaufmann, 2001.
2. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS VIII), ACM Press, 1998, pp. 58-69.
3. "SiByte SOCs: BCM1125H and BCM1250," Broadcom Corp., http://sibyte.broadcom.com/public/chips/index.html#bcm1250.
4. J. Emer, "Ev8: The Post-Ultimate Alpha," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 01), IEEE CS Press, 2001, keynote address.
5. J. Kahle, "Power4: A Dual-CPU Processor Chip," *Proc. Microprocessor Forum*, In-Stat MDR, 1999.
6. M. Tremblay, "MAJC: Microprocessor Architecture for Java Computing," *Proc. 11th Ann. Int'l Symp. High-Performance Chips* (Hot Chips XI), IEEE CS Press, 1999.
7. M. Cintra, J.F. Martinez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 27), ACM Press, 2000, pp. 13-24.
8. P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," *Proc. ACM Int'l Conf. Supercomputing* (ICS 99), ACM Press, 1999, pp. 365-372.
9. J.G. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 27), ACM Press, 2000, pp. 1-12.
10. M. Chen and K. Olukotun, "TEST: A Tracer for Extracting Speculative Threads," *Proc. Int'l Symp. Code Generations and Optimization* (CGO 03), IEEE CS Press,

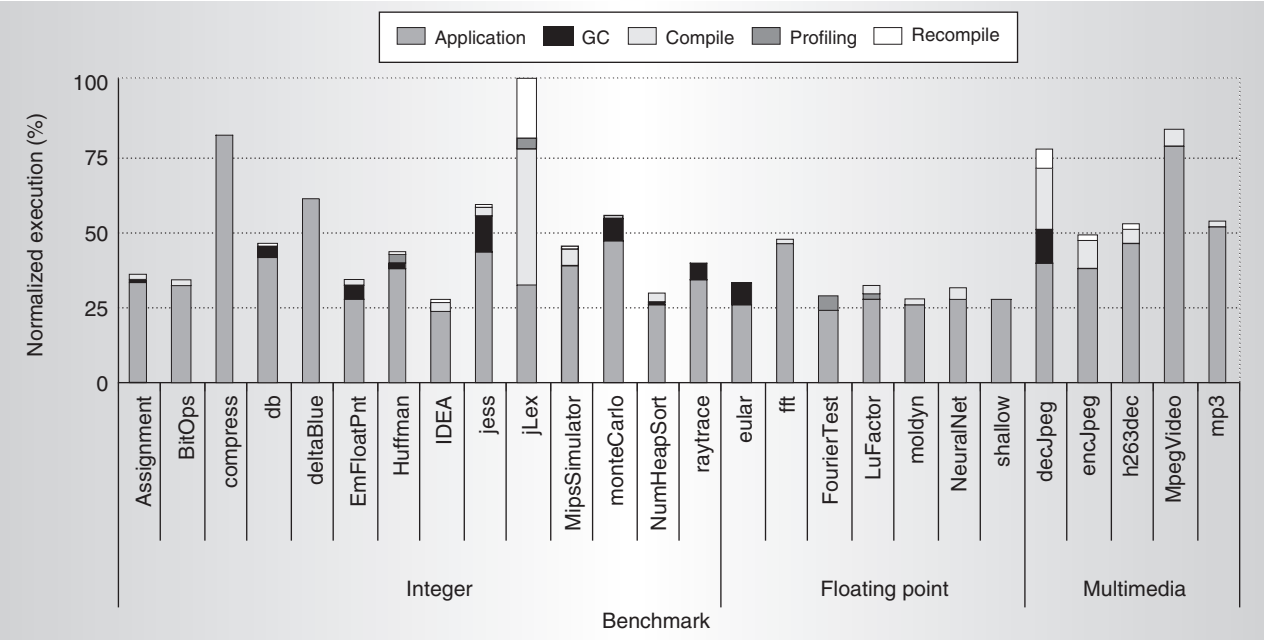

Figure 6. Total program speedup with compilation, garbage collection, profiling, and recompilation overheads using default benchmark data sets.

2003, pp. 301-312.

11. S. Muchnick, *Advanced Compiler Design Implementation*, Morgan Kaufmann, 1997.

Michael K. Chen is a PhD candidate in electrical engineering at Stanford University. His research interests include high-performance and parallel computer architectures, and optimizations for improving the performance of modern programming language features, such as garbage collection and dynamic compilation. Chen has a BS and MS in electrical engineering, and an MS in industrial engineering, all from Stanford University. He is a member of the IEEE and the ACM..

Kunle Olukotun is an associate professor of electrical engineering at Stanford University, where he leads the Hydra project. His research interests include the design, analysis, and verification of computer systems using ideas from computer architecture and CAD. Olukotun has a BS in computer engineering from Calvin College and an MS and PhD in computer engineering from the University of Michigan. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Michael K. Chen, Stanford University, Gates Bldg. 3A-316, Stanford, CA, 94305-9030; mikey@hydra.stanford.edu.