TEST: A Tracer for Extracting Speculative Threads

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

Kunle Olukotun
Stanford University
kunle@ogun.stanford.edu

Abstract

Thread-level speculation (TLS) allows sequential programs to be arbitrarily decomposed into threads that can be safely executed in parallel. A key challenge for TLS processors is choosing thread decompositions that speedup the program. Current techniques for identifying decompositions have practical limitations in real systems. Traditional parallelizing compilers do not work effectively on most integer programs, and software profiling slows down program execution too much for real-time analysis.

Tracer for Extracting Speculative Threads (TEST) is hardware support that analyzes sequential program execution to estimate performance of possible thread decompositions. This hardware is used in a dynamic parallelization system that automatically transforms unmodified, sequential Java programs to run on TLS processors. In this system, the best thread decompositions found by TEST are dynamically recompiled to run speculatively. This paper describes the analysis performed by TEST and presents simulation results demonstrating its effectiveness on real programs. Estimates are also provided that show the tracer requires minimal hardware additions to our speculative chip-multiprocessor (< 1% of the total transistor count) and causes only minor slowdowns to programs during analysis (3-25%).

1. Introduction

Modern compilers that perform array datra-dependence analysis can automatically parallelize Fortran-like numerical applications on traditional multiprocessors [2][5][14][27]. 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][21]. These difficulties introduce imprecision into dependence relations, limit the accuracy of parallelism estimates, and force conservative synchronization to safely handle potential dependencies when running on a traditional multiprocessor.

Thread-level speculation (TLS) is a technique that enables parallel execution of sequential applications on tightly coupled multiprocessors without explicit synchronization. With TLS, a sequential program can be decomposed into threads to be run on the processors without regard to memory dependencies. The threads are sequenced in the order in which they would execute sequentially, but are actually executed in parallel. Data speculation hardware support ensures that true dependencies between memory accesses are honored across threads by backing up and restarting threads that execute an inter-thread dependent load too early.

We have designed Hydra, a chip multiprocessor (CMP) with thread speculation support [15]. The granularity of speculative threads targeted by Hydra ranges from 10s to 1000s of instructions. We would like decompositions to be chosen automatically because such fine-grained threads are impractical to manually identify in large programs and are smaller than threads that should be explicitly coded by a programmer.

A CMP with speculation support allows us to approach automatic parallelization differently. Chip multiprocessors combine several CPUs onto one die with a tightly coupled memory interface. The high-bandwidth, low latency interconnect and small number of processors (four) in Hydra reduces the importance of parallelizing compiler optimizations like blocking and loop reordering that minimize inter-processor communication in traditional multiprocessors. Potential write-after-read (WAR) and write-after-write (WAW) hazards, which must be identified and correctly ordered by the compiler in traditional multiprocessors, are automatically handled by the speculation hardware and never exact performance penalties.

TLS simplifies many automatic parallelization challenges, but there are significant constraints that must be considered. On Hydra, the major constraints are:

- True inter-thread data dependencies, or read-after-write (RAW) hazards, always limit speedup from parallel execution of speculative threads.
- Speculative read and write state that is buffered by the hardware may overflow for large, long-running speculative threads, forcing speculative execution to stall until a load or store can be performed safely.
- Only one thread decomposition may be active (e.g. one loop in a loop nest) at a given time.
- Compiled speculative thread code introduces sequential

overheads [22][29] from speculative thread management routines and forced communication of inter-thread dependent local variables.

These constraints impose conflicting requirements when selecting thread decompositions. While threads can be composed in many ways [23][7][15], most research has focused on decompositions based on loops because experimentally, they have yielded the most performance gains. A speculative thread loop (STL) is a loop decomposed into threads where one loop iteration is one thread. Applying the constraints to these decompositions, speculating on small loops limits parallel coverage and suffers from higher speculative thread overheads relative to work performed. Speculating on large loops increases the probability of speculation buffer overflows and may incur higher dependency violation penalties.

Previous studies of TLS processors have relied on simulation and profiling to identify appropriate STLs [23][29]. In these studies, cycle-accurate CPU simulators with dependence analysis of executing memory references were used to deduce performance estimates of potential STLs. These results were used to select the best STLs for actual speculative execution. This type of dynamic analysis is computationally expensive, slowing execution over 100x during analysis, but is effective and bypasses the difficulties of static compiler analysis.

Dynamic analysis to identify STLs complements a TLS processor's ability to parallelize optimistically and use hardware to guarantee correctness. The primary goal is to identify where parallelism *usually* exists. As stated earlier, integer program characteristics force dependency analysis using static compilers to be conservative and imprecise, making it difficult deduce program parallelism. These characteristics also make it difficult to statically estimate thread size and buffer requirements essential to predicting speculative performance. Profiling, on the other hand, can provide accurate statistics on dynamic dependency behavior, thread size, and buffer requirements for most types of programs.

In this paper, we present Tracer for Extracting Speculative Threads (TEST), hardware support in Hydra for low-overhead profiling of potential STLs. Along with a Java virtual machine (JVM), Hydra with TEST and TLS support are used together to create the Java Runtime Parallelizing Machine (Jrpm), a complete system that uses speculative threads to dynamically parallelize sequential Java programs.

Virtual machines (VMs) like the JVM and Microsoft's Rotor (for C#) have become commercially popular in recent years as a way of supporting applications without being tied to a specific operating system or processor architecture. Our system demonstrates how this virtualization can be used in a novel way to seamlessly support new execution models without modifying existing platform independent program sources. Key to making this work is the VM's platform-independent *bytecode* representation and dynamic recompilation support.

In the Jrpm system, the compiler derives a control-flow graph from program *bytecodes* and analyzes it to identify potential STLs. A program that has been dynamically compiled with instructions annotating local variables and STL boundaries is executed as a sequential program on a single processor of Hydra. The trace hardware collects statistics for the prospective STLs. The JVM takes results from TEST analysis to identify the best code sub-sections to dynamically recompile into speculative threads. By relying on dynamic execution profiles collected by TEST, a simple scalar compiler is all that is required to generate the speculative threads.

The contributions of this paper are:

- We show how potential STLs are selected from Java programs and describe a simple trace algorithm to select optimal decompositions quickly. The analysis extracts speculative thread coverage, data dependence timing, and speculative state requirements from potential STLs, and identifies the best decompositions by comparing the collected statistics. The analysis also generates dependency profiles for directing compiler optimizations and performance tuning by a programmer to improve speculative thread performance.
- We describe the hardware and software for implementing the trace algorithm on Hydra. During analysis, this implementation introduces only minimal slowdowns (3-25%) to sequentially executing Java programs. A similar software-only implementation would be unacceptable for use in a dynamic compilation system because it slows execution over 100x during analysis. Total hardware requirements for implementing TEST are minimal because it utilizes some of the hardware normally used only for speculation, and only requires addition of a small hardware comparator array (< 1% of the total CMP transistor count) and support for a few additional processor instructions.

The outline of this paper is as follows: Section 2 describes related work in TLS, dynamic dependence detection, and dynamic parallelization systems. Section 3 is a brief introduction and overview of the Jrpm system. We describe the algorithms for analyzing traces and selecting STLs in Section 4 and provide details of the TEST hardware implementation in Section 5. In Section 6, we present results from applying TEST to real programs. Finally, conclusions are discussed in Section 7.

2. Related Work

The Multiscalar paradigm [11] was the first complete description and evaluation of an architecture with TLS support. Several architectures for TLS using CMPs have been proposed [9][15][20][29]. These implementations

have mostly targeted coarser grains of granularity than the Multiscalar architecture. Similar to TLS, software-based dynamic dependence detection has been proposed for traditional multiprocessor systems to preserve correctness for loops executed in parallel that may have complex dependency patterns [12][25][26][28].

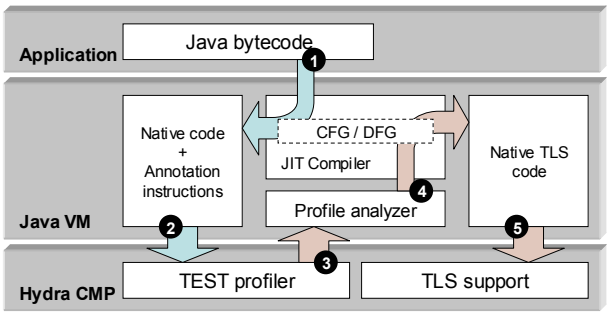
There has been some related work on selecting decompositions for TLS processors. The Multiscalar [32] compiler focused on compile-time heuristics to increase intra-procedural task sizes and intra-task dependency scheduling to increase task parallelism. These optimizations are sufficient for the smaller threads targeted by Multiscalar, but do not address the memory disambiguation and decomposition selection difficulties of compiling for coarser-grained TLS systems. Trace-driven simulation was used by Oplinger et al. to study the limits of TLS [23], and by Steffan and Mowry for manual selection of loops to be executed speculatively on the Stampede TLS machine [29]. Neither study specifically addresses how their technique can be used for automatic speculative compilation or how their analysis can be performed without significant simulation overheads. Chaudhry et al. [6] provide few details about how decompositions are selected for the MAJC TLS processor and Cintra et al. [9] restrict speculative decompositions in their TLS architecture to the inner-most loop of a loop nest.

There has also been research on improving performance of STLs once they are selected. These techniques include hardware synchronization and value prediction schemes to minimize violations [10][30], and compiler scheduling to increase distances between inter-thread dependencies [30][34].

Numerous commercial and research compilers based on array dependence analysis for parallelizing Fortran programs have been developed [2][5][14][27]. Several studies have looked at how these compilers might be applied to general programs [4][17][24]. The Jrpm system exploits parallelism exposed to the hardware analyzer with only a simple scalar compiler. Additional use of an array data-dependence compiler might further improve parallel performance of STLs chosen by TEST and uncover hidden parallelism, but this is beyond our current scope.

There has been other loosely related research that has used feedback to dynamically parallelize programs. Ko et al. [18] identified optimal decompositions through brute-force incremental execution of all possible decompositions in multi-level parallel programs. There have been numerous systems designed to tune parallel performance on traditional multiprocessors [1][13][19], but they have relied on off-line, rather than real-time, dependency analysis of memory traces.

3. Java Runtime Parallelizing Machine (Jrpm)



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

Figure 1 – Block diagram of Jrpm.

A block diagram of Jrpm outlining its various software and hardware components is shown in Figure 1. The compiler chooses possible STLs by analyzing a program control-flow graph. A program that has been dynamically compiled with instructions annotating local variables and STL boundaries is executed as a sequential program on one processor of Hydra. The trace hardware analyzes prospective STLs. The code is dynamically recompiled into speculative threads on regions that are predicted to have the largest speedup and most coverage. Section 3.1 and Section 3.2 provide a brief overview of the Hydra CMP and the microJIT dynamic compiler.

3.1. Hydra CMP with Speculative Thread Support

Hydra [15][22], shown in Figure 2, consists of four single-issue pipelined MIPS processors, each with private L1 data caches, attached to an integrated on-chip L2 cache with separate read and write buses. Speculative thread support in our CMP consists of special coprocessor instructions, extra speculative tag bits added to each line of the L1 data caches, and a set of secondary cache write buffers. The coprocessor instructions provide an interface to the thread speculation control hardware, the tag bits detect data dependency violations between threads, and the write buffers hold speculative data until it can be safely committed to the secondary cache or discarded. The physical limits of buffered speculative state are given in Table 1.

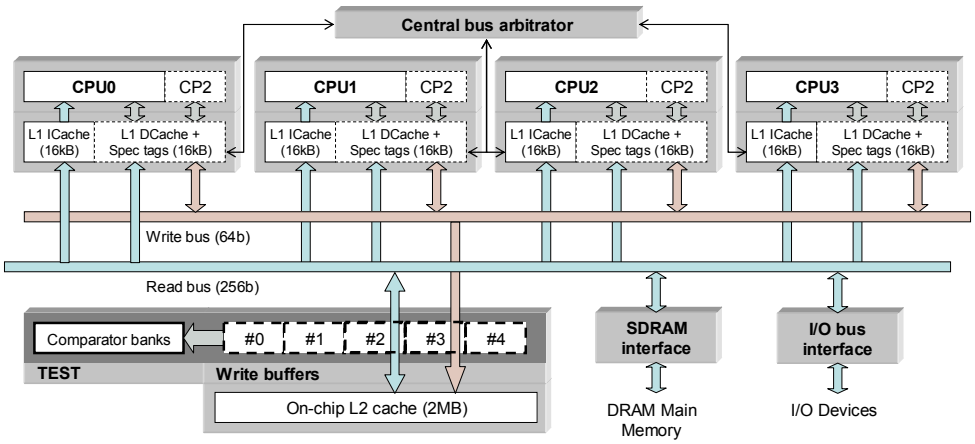


Figure 2 – Block diagram of our CMP. TLS support blocks shown in dotted lines. TEST hardware blocks shown in dark blocks.

Table 1 – Thread-level speculation buffer limits.

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

Table 2 – Thread-level speculation overheads.

TLS Operation	Overhead / delay	Additional overheads
Loop startup	25 cycles	Initialize loop local variables Load register-allocated loop invariants
Loop shutdown	25 cycles	Complete sum and min/max reductions
Loop end-of-iteration	5 cycles	Increment loop iterators
Violation and restart	5 cycles	Load register-allocated loop invariants
Store-load communication	10 cycles	

3.2. microJIT Dynamic Compiler

The open-source Kaffe virtual machine (http://www.kaffe.org/) was chosen for our Java runtime system. The microJIT compiler [8] was augmented to generate both annotated code for TEST analysis and speculative thread code. Once a STL is chosen, the compiler inserts assembly routines, shown in Table 2, that start, control, and terminate speculative threads. These routines introduce some overheads relative to the original sequential code. The compiler also performs optimizations and transformations on the selected STLs. Inter-thread local variable dependencies are globalized and local variable initializers are communicated to each thread. Optimizations that improve speculative performance, like register allocating loop invariants, using non-violating loop inductors, inserting synchronization locks, and transforming reduction operations (e.g. sum, min/max), are applied when possible. Details on these speculative compiler optimizations will be presented in a future paper.

4. TEST Algorithms

This section describes in-depth the trace analyses performed on potential thread decompositions and the decision process used to select thread decompositions from collected statistics. Section 5 describes the hardware and software support required to support the trace analyses.

4.1. Identifying Potential Decompositions

Speculative threads can be composed from loops, method call returns, and general regions [7][15][23]. The remainder of this paper will focus only on decompositions formed from loops. Our experiments so far have not found many method call return or general region decompositions that are either not covered by similar loop decompositions or have significant coverage to impact total execution time.

The compiler chooses potential STLs by examining a method's control-flow graph to identify all natural loops [21]. Loops are chosen optimistically, relying instead on TEST results to identify desirable decompositions. Any loop without obvious loop-carried dependencies that would completely eliminate speedup (e.g. end-of-loop store and start-of-loop load) is considered a potential STL. Loop inductors [21], which are dependencies that can be eliminated by the compiler, are ignored so that potentially parallel loops are not overlooked. Scalar analysis is used to identify simple dependencies, but we forgo advanced techniques that require analyzing array access patterns, control flow, or memory accesses to find dependencies.

4.2. Trace Analyses

This section describes the two trace analyses performed to characterize the potential of a STL, the *load dependency analysis* and the *speculative state overflow analysis*. TEST analysis relies on the notion of event *timestamps*, or the time when an event occurs. *Timestamps* from different events are compared against each other to compute specific statistics.

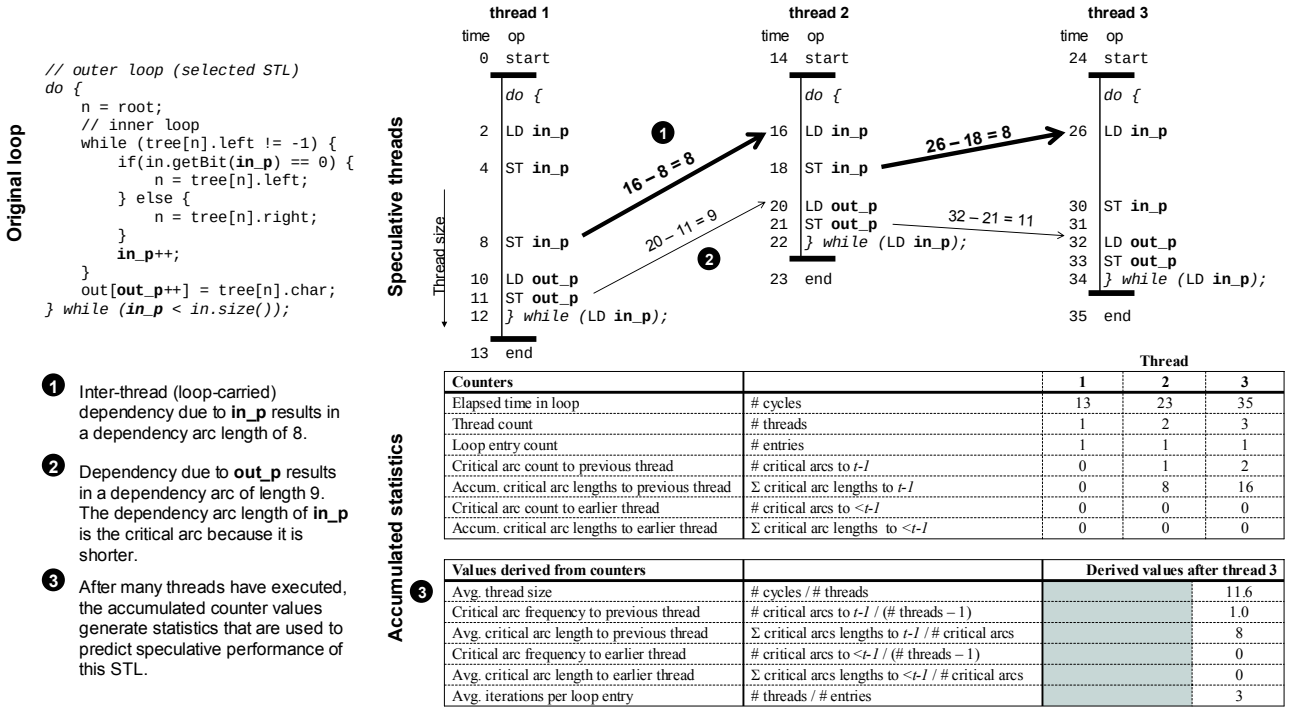


Figure 3 – Example (Huffman decode) of the *load dependency analysis*. Analysis is performed on the outer loop in this example. Loop-carried dependencies are bold in source code. Arrows represent *dependency arcs*. *Critical arcs* shown in darker arrows.

4.2.1 Load dependency analysis. The *load dependency analysis* looks for inter-thread dependencies for a STL, as illustrated in Figure 3. A *store timestamp* is recorded on a memory or local variable store, and retrieved on a subsequent load to the same address. The *store timestamp* is checked against *thread start timestamps* to determine if an inter-thread *dependency arc* exists to the previous thread (*t-1*), or an earlier thread (< *t-1*). If an inter-thread *dependency arc* is detected, the *arc length*, the difference between the current time and the *store timestamp,* is recorded. While many *dependency arcs* may exist between any two given speculative threads, we only record the *critical arc* (dark arrows in figure), which is the shortest arc and limits parallelism between the threads.

4.2.2 Speculative state overflow analysis. The *speculative state overflow analysis* checks that speculative state for a STL can fit within the L1 caches and store buffers. Each L1 data cache line includes additional tag bits to record a processor's speculative read state. Each speculative store buffer, with L1 cache line sized entries, collects all speculative heap writes made by a processor. Dropping a L1 cache line with speculative state or overflowing a store buffer forces a speculatively executing thread to stall until reads or writes can be performed safely (when the thread becomes the non-speculative "head" thread).

Time	Op	Tag (bits 31:11 for ST) (bits 31:14 for LD)	Index (bits 10:5 for ST) (bits 13:5 for LD)	(a) LD timestamp hit	(b) ST timestamp hit	(c) Old timestamp	(d) Thread start time	(e) LD/ST in current thread? is (c) > (d) ?	(f) LD counter ++ on LD if not (a) or not (e)	(g) ST counter ++ on ST if not (b) or not (e)	(h) Overflow ++ if (f) > LD limit or (g) > ST limit
0	New thread						0		0	0	0
5	LD 0x20000	0x8	0x0	N			"		1	0	0
10	ST 0x10040	0x20	0x2		N		"		1	1	0
15	LD 0x20008	0x8	0x0	Y		5	"	Y	1	1	0
20	LD 0x20040	0x8	0x2	N			"		2	1	0
25	New thread						25		0	0	0
30	LD 0x20000	0x8	0x0	Y		15	"	N	1	0	0
35	LD 0x10040	0x4	0x2	N			"		2	0	0
40	ST 0x10040	0x20	0x2		Y	10	"	N	2	1	0
45	ST 0x10048	0x20	0x2		Y	40	"	Y	2	1	0
50	LD 0x20000	0x8	0x0	Y		30	"	Y	2	1	0

Figure 4 – Illustration of the *speculative state overflow analysis*. Thread start time (column d) and counters (columns f & g) are reset for every new thread. Arrows show conditions that cause counters to be incremented.

$$\text{base\_speedup} = \sum_{t=\text{all\_prev\_threads}} \text{critical\_arc\_freq}_t \bullet \begin{cases} 4 & \left| \frac{\text{avg\_critical\_arc\_distance}_t}{\text{avg\_thread\_size}} > \frac{3}{4} \right. \\ \frac{\text{avg\_thread\_size}}{\text{avg\_thread\_size} - \text{avg\_critical\_arc\_distance}_t} & \left| \frac{\text{avg\_critical\_arc\_distance}_t}{\text{avg\_thread\_size}} <= \frac{3}{4} \right. \end{cases}$$

$$\text{spec\_time} = (\text{startup\_and\_shutdown\_overhead} \bullet \#\text{entries} + \text{orig\_time} + \text{eoi\_overhead} \bullet \#\text{threads}) \bullet \left(\text{overflow\_freq} + \frac{1 - \text{overflow\_freq}}{\text{base\_speedup}} \right)$$

$$\text{speedup} = \frac{\text{orig\_time}}{\text{spec\_time}}$$

Equation 1 – Estimated speedup for a STL.

An example of the overflow analysis on a set of STL memory references is shown in Figure 4. The *speculative state overflow analysis* relies on timestamps associated with cache lines. 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* (columns a & b). If no timestamp exists or if it is less than the current *thread start timestamp* of a STL (column e), counters tracking buffer requirements are incremented to reflect new buffer state required by the current thread. The *load counter* (column f) tracks new speculatively read cache lines, and the *store counter* (column g) tracks new store buffer entries. The *overflow counter* (column h) is incremented if either counter for the current thread exceeds buffer limits.

4.3. Selecting Thread Decompositions

The statistics shown in Figure 3 and Figure 4 are accumulated over time and analyzed to predict the performance of a STL. The estimated speedup for a STL, shown in Equation 1, is derived from average critical arc frequencies, thread sizes, critical arc lengths, overflow frequencies, and speculative overheads.

Speedup is limited to four in Hydra (or total number of processors in the CMP). Note that we expect maximal speedup if the average *critical arc length* is at least ¾ the average thread size (or *(p-1)/p* where *p* is the number of processors). This is the point at which the processors are completely utilized and the inter-thread dependencies are separated enough not to limit speedup.

There are multiple possible decompositions that can be chosen in a loop nest. For example, in Figure 3, either the outer loop or the inner loop can be transformed into speculative threads, but not both at the same time. We select the best STL by comparing the estimated execution time using speculative threads for a STL against the estimated execution time using speculative threads for any nested STL decompositions instead, plus any non-speculative serial execution, as shown in Equation 2.

Table 3 shows how this equation is applied to the example loop in Figure 3.

$$\frac{\text{time}_{\text{this\_tsd}}}{\text{speedup}_{\text{this\_tsd}}} <> \text{time}_{\text{this\_tsd}} + \sum_{s=\text{nested\_tsd}} \text{time}_s \bullet \left(\frac{1}{\text{speedup}_s} - 1 \right)$$

Equation 2 – Comparison function for choosing an optimal STLs.

Table 3 – Application of Equation 2 to the loops from Figure 3 chooses the outer loop as the better STL.

	Outer loop		Inner loop	Serial
Sequential time (cycles)	18941K		13774K	5167K
Speedup	1.85		1.30	1.00
TLS time (cycles)	10238K		10595K	5167K
Total time (cycles)	***10238K***	<	*15762K*	

5. TEST Hardware Implementation

Simulations indicate program execution slows over 100x when profiling using a software-only implementation of the trace analyses described in Section 4.2. Overheads result from callback annotations on every memory and local variable access, and comparisons required to resolve inter-thread dependencies and compute speculative state requirements. This magnitude of slowdown is unacceptable in a real dynamic compilation system, even if the analysis is only performed on a limited basis. Furthermore, the significant software overheads introduce imprecision into the analysis, making difficult to derive accurate thread size and *dependency arc* lengths.

The overheads and imprecision of software-only analysis led us to consider how hardware support could speedup profiling and improve accuracy. The hardware we designed analyzes a sequentially executing program and works when speculation is disabled. Annotation instructions inserted by the dynamic compiler (Section 5.1) into native code mark important events. The annotation instructions communicate events to the hardware comparator banks that perform the actual trace analyses (Section 5.2). The speculative store buffers, which are idle during sequential non-speculative execution, hold *timestamps* of previous events (Section 5.3).

Table 4. Summary of annotating instructions and associated operations.

Instruction	Description	Normal operation	Trace operation (when enabled)
`lw|lb|lbu|lh lhu|lwc1` *addr*	Load	Load	Get memory access *store timestamp* and *cache line timestamp* Record memory access *cache line timestamp*
`sw|sb|sh| swc1` *addr*	Store	Store	Get previous *cache line timestamp* Record memory access *store timestamp* and *cache line timestamp*
`lwl` *vn*	Local variable load	none	Get *store timestamp* for local variable *vn*
`swl` *vn*	Local variable store	"	Record *store timestamp* for local variable *vn*
`sloop` *n*	Start loop	"	Allocate comparator bank Increment current bank pointer Set current *thread start timestamp* Reserve *n* local variable *store timestamps*
`eoi`	Loop end-of-iteration	"	Shift *thread start timestamps* for current bank Set current *thread start timestamp* for current bank
`eloop` *n*	End loop	"	Free comparator bank Decrement current bank pointer Free *n* local variable *store timestamps*

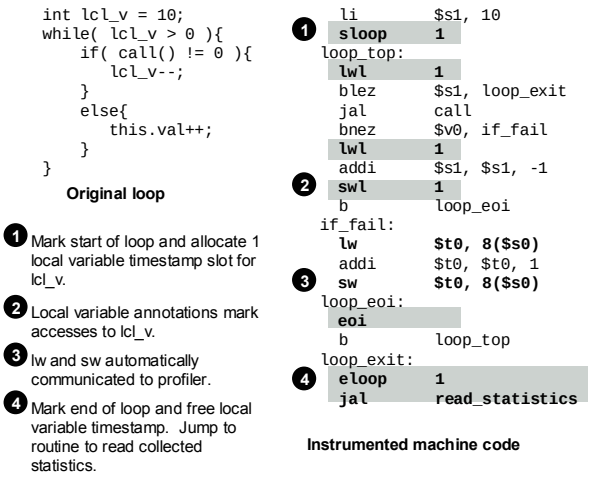


Figure 5 – A sample loop compiled with annotating instructions.

5.1. Annotating Instructions

The annotating instructions, shown in Table 4 and Figure 5, mark events relevant to the trace analyses described in Section 4.2. Memory load and store events are automatically communicated to the tracing hardware when tracing is enabled. `sloop`, `eloop`, and `eoi` instructions mark the entry, exit and end-of-iteration of a potential STL. Local variables in the same calling context as a potential STL are tracked with explicit annotations to simplify tracking of named variables that move between registers and the runtime stack in optimized compiled code. Block-local and temporary variables are not annotated because they never cause a dependency. At the end of a STL (e.g. exit from a loop), special routines read the collected statistics from TEST.

Program execution slowdown results from local variable annotations and overheads to read collected statistics at the end of a STL, as shown in Figure 6. Several optimizations were performed by the JIT compiler to reduce slowdowns of the annotated native code. Only the first local variable load in a block or a loop is annotated since it would result in the shortest *dependency arc*, if one existed. Within loop nests, calls to read collected statistics at the end of a STL are hoisted to the outer-most loop when there is only one loop at each level. After performing these optimizations, most benchmarks experience no more than 10% slowdown, and only 2 applications have slowdowns approaching 25%.

5.2. Comparator Array

The comparator banks carry out the bulk of the dependency and overflow trace analyses. One comparator bank, shown in Figure 7, tracks the progress for a given STL. Each bank, primarily composed of comparators and counters, analyzes incoming loads and stores. An array of comparator banks allows us to trace multiple potential STLs executing concurrently, as would be the case when analyzing nested loops. Comparators are used to compare *thread start timestamps* against incoming *cache line timestamps* to check for speculative state overflows and against *store timestamps* to identify *critical arcs*. At the end of each thread of a STL, arc lengths, critical arc counts, and buffer overflows are accumulated into counters.

In a real implementation of TEST with multiple banks, logic in the *critical arc calculation* block can be shared between banks. A given load access can be a dependency for only one STL. For example, in a nested loop, a load dependency may exist to a previous iteration in the current loop or iteration in an enclosing loop, but not to both. Consequentially, only one *critical arc calculation* block in an array of comparator banks will be active for a given load access. To share this block, the *critical arc calculation* block is pipelined after *dependency arc identification*, as shown in Figure 8a.

Multiple comparator banks are assigned in loop nests when multiple potential STLs execute concurrently. The finite number of comparator banks restricts the number of loops in a loop nest that can be analyzed concurrently. Several mechanisms help ensure the banks are still applied

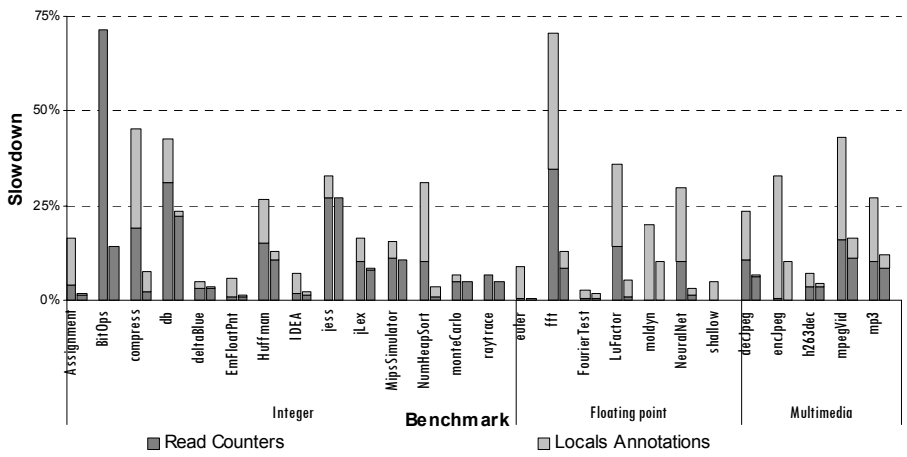


Figure 6 – Execution slowdown during profiling of benchmarks. For each application, the 1st column is base annotations and the 2nd column is optimized annotations.

effectively. Precedence is initially given to the outer-most loop, and analysis of deeply nested loops is disabled when there are no comparator banks left or no room left for local variable *timestamps* (see `sloop` and `eloop` in Table 4). When a comparator bank consistently predicts speculative buffer overflows for an outer STL, it can be freed to be used deeper in a loop nest. When sufficient data has been collected to predict behavior for a STL, the annotations marking it can be disabled dynamically (e.g. overwriting JIT compiled code with `nop` instructions). This ensures the system can eventually collect information on decompositions deep within a loop nest.

Pipelining access to the critical arc calculation block also enables TEST to provide detailed dependency information that can be used for optimization. In an extended implementation of TEST, the registers and counters in the critical arc calculation block are replaced with accesses to content addressable SRAM, as shown in Figure 8b. This configuration allows critical arc lengths, accumulated *critical arc* lengths, and *critical arc* counts to be binned by the load instruction PC to be later analyzed by a programmer or compiler.

We estimated how much logic would be consumed by an implementation of TEST. Transistor counts were

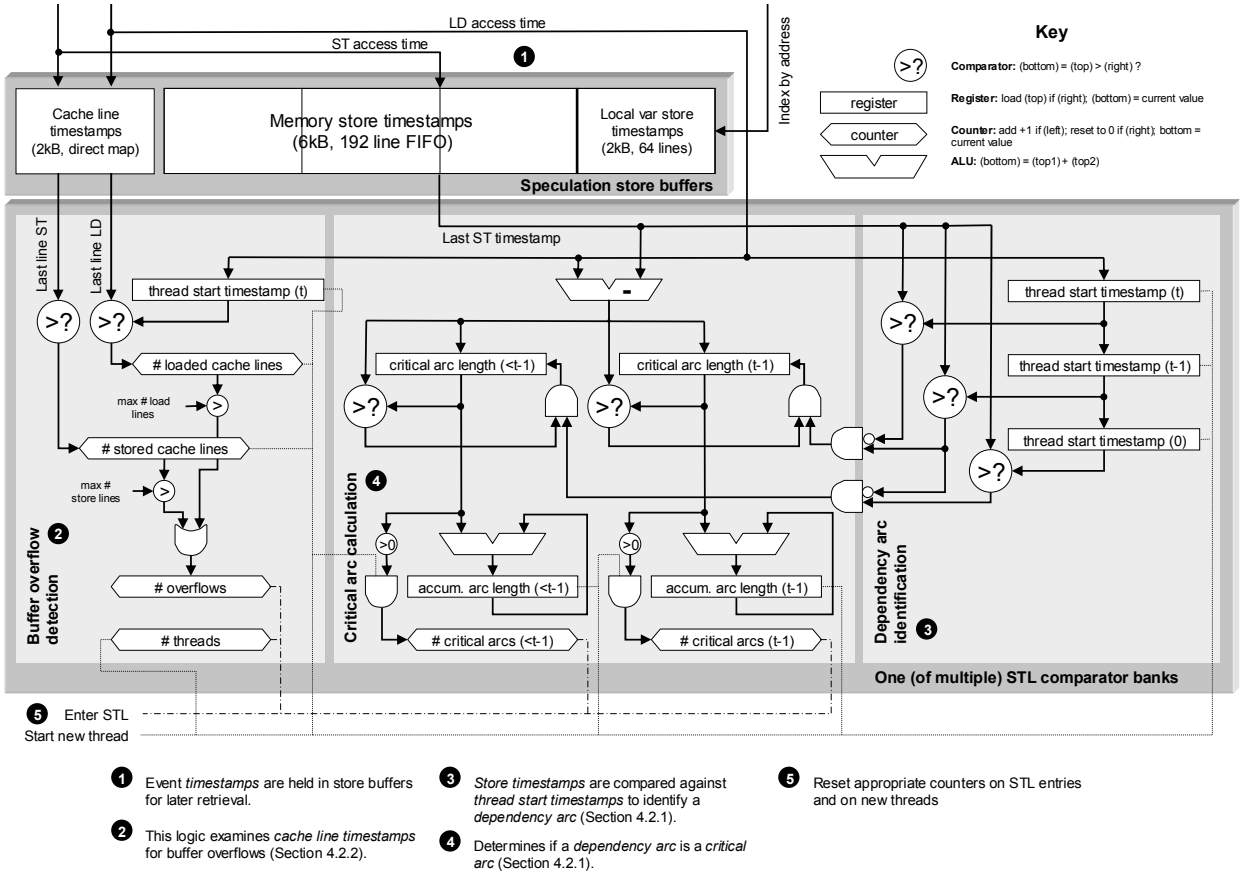


Figure 7 – Block diagram of one comparator bank.

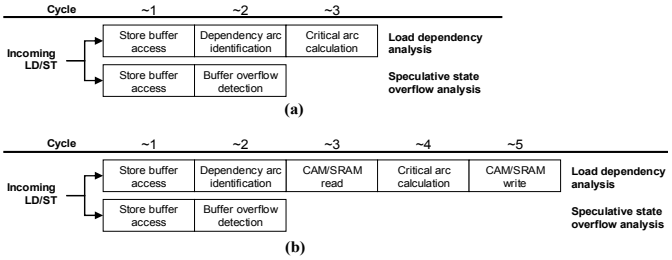


Figure 8 – Comparator bank (a) base pipeline design (b) extended pipeline design.

Table 5 – Transistor count estimates Hydra with TLS and TEST support.

Structure	Count	Transistors Each	Transistors Total	% of total
CPU + FP core	4	2500K	10000K	8.64%
16kB I / 16kB D Cache	4	1573K	6291K	5.43%
2MB L2 cache	1	98304K	98304K	84.91%
Write buffer	5	172K	861K	0.74%
Comparator bank	8	39K	322K	0.28%
Total			115778K	100.00%

derived from the logic required for a implementation with eight comparators eight. As Table 5 suggests, TEST hardware would add less than 1% to the transistor count of the Hydra CMP with TLS support.

5.3. Store Buffers

The five store buffers (see [15] for an explanation of why there are five) that normally hold writes during speculative execution hold *timestamps* during profiling. Each store buffer is 2KB (64 32B cache lines). The store buffers are statically partitioned, with three buffers holding heap access *store timestamps*, one holding *cache line timestamps*, and one holding *store timestamps* to local variables. An address' *timestamp* is returned when requested by an annotating memory or local variable instruction.

Store timestamps in the store buffers, organized as FIFO (first-in, first-out) during tracing, effectively hold a limited history of memory and local variable accesses. For heap access *store timestamps*, 192 cache lines, or 6KB, of write history can be held.

To keep logic additions for the *speculative state overflow analysis* simple, the lower bits of an address index the store buffer holding *cache line timestamps* and *cache line tags* like a direct mapped cache. The actual speculation store buffers are fully associative and the L1 caches are 4-way set associative. Not accounting for associativity introduces some error into the overflow analysis, but should not affect its usefulness in estimating speculative state requirements.

6. Benchmark Analysis

6.1. Benchmark Results

Table 6 lists benchmarks we have evaluated using TEST, including applications from the jBYTEmark (http://www.byte.com/), SPECjvm98 (http://www.specbench.org/), and Java Grande (http://www.epcc.ed.ac.uk/javagrande/javag.html) benchmarks suites. The columns on the right of this graph summarize the characteristics of the STLs chosen by TEST. Figure 10 shows graphically the coverage and expected speedups of selected STLs.

Overall, the selected STLs exhibit significant thread size (column h) and coverage diversity. MipsSimulator, raytrace, IDEA, EmFloatPnt and FourierTest have very coarse threads while moldyn and NeuralNet have very fine-grained threads. Ignoring inter-thread dependencies, analysis of the selected STLs suggest that the thread size is primarily constrained by the limited store buffer size rather than the speculative load state in the L1 cache lines.

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 shown in the table, but the omitted decompositions do not have any significant coverage (< 0.5%). mp3, db, jess, and jLex have significant sections of serial execution not covered by any potential STLs, limiting total speedup for these applications.

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 a third of the benchmarks can be analyzed by a traditional parallelizing compiler (column a) (e.g. programs that resemble Fortran floating point code w/ affine array accesses, no dynamic objects/pointers, bounded loops, and little control flow). The average height of selected loops from the inner loop (column f) suggests that desired STLs have granularities larger than the inner-most loop in a loop nest. The maximum depth of loop nests executed (column d) indicates that eight comparator banks are sufficient to analyze most of the benchmark programs without intervention from the runtime system.

Apart from simplifying parallelization of floating point programs for a CMP and automatically selecting STLs in integer applications that are difficult to analyze statically, our experiments suggests that dynamic parallelization has other potential benefits. One advantage is that STL selections can be made that account for input data set sizes. We noticed several applications where selected 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

Table 6 – Benchmarks evaluated with STLs selected by TEST.

Category	Benchmark	Description	Data set	Characteristics (a) Analyzable for parallelism	(b) Data set sensitive	(c) Loop count	(d) Loop depth	TEST Analysis (e) Selected loops (>0.5% coverage)	(f) Avg. selected loop height	(g) Threads / STL entry	(h) Thread size (cycles)
Integer											
	Assignment	Resource allocation	51x51	N	Y	32	5	11	2.0	29	199
	BitOps	Bit array operations		N	N	4	2	2	1.0	7646	29
	compress	Compression		N	N	28	4	4	1.8	93755	546
	db	Database	5000.	N	Y	37	5	6	1.7	23142	510
	deltaBlue	Constraint solver		N	N	22	4	5	2.6	82	501
	EmFloatPnt	FP emulation		N	N	7	3	1	2.0	255	20127
	Huffman	Compression		N	N	14	3	6	1.3	502	108
	IDEA	Encryption		Y	N	2	2	1	2.0	242	6307
	jess	Expert system		N	N	134	11	4	5.3	166	339
	jLex	Lexical analyzer gen		N	N	128	6	7	3.0	71	2699
	MipsSimulator	CPU simulator		N	N	19	4	2	3.5	51931	1313
	monteCarlo	Monte carlo sim		N	N	15	3	5	1.4	942	119
	NumHeapSort	Heap sort		N	N	5	3	2	2.0	6081	555
	raytrace	Raytracer		N	N	14	4	1	1.0	65	158
Floating point											
	euler	Fluid dynamics	33x9	Y	Y	32	2	13	1.1	66	304
	fft	Fast fourier transform	1024.	Y	Y	5	3	2	2.0	187	231
	FourierTest	Fourier coefficients		Y	N	2	2	1	2.0	100	167802
	LuFactor	LU factorization	101x101	Y	Y	13	3	7	1.6	64	455
	moldyn	Molecular dynamics		Y	N	8	2	1	1.0	1026	96
	NeuralNet	Neural net	35x8x8	Y	Y	19	4	8	1.9	9	617
	shallow	Shallow water sim	256x256	Y	Y	11	3	3	1.0	257	1420
Multimedia											
	decJpeg	Image decoder		N	N	61	5	21	2.2	34	124
	encJpeg	Image compression		N	N	62	8	9	1.6	54	121
	h263dec	Video decoder		N	N	54	5	3	3.0	165	212
	mpegVideo	Video decoder		N	N	69	8	9	1.4	23	701
	mp3	mp3 decoder		N	N	98	6	17	2.3	55	181

2-dimensional data arrays. 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. Choosing STLs dynamically also allows selected STLs to change as CMP designs evolve. For example, larger STLs that would cause speculative buffer overflows in our current system could be chosen during runtime by a future Hydra design with larger speculative store buffers and L1 caches.

6.2. Imprecision Effects

Each benchmark was run speculatively on Hydra using the STLs selected by TEST. Figure 11 shows our analysis does a good job of predicting speculative performance. A comparison of speculative performance and results from TEST suggests disparity results mostly from selected STLs with highly varying thread sizes and large violation rates during actual speculative execution. One should keep in mind, though, that relative speedup estimates to other potential STLs are more important. The primarily role of TEST is to identify the best STLs, so absolute values are not critical.

Precision is lost during the accumulation and binning of thread statistics. With our algorithm, temporal dependency information is lost that could detect multi-iteration parallelism, as illustrated in Figure 9. In non-constant sized loops or in loops where work increases or decreases monotonically by iteration, accumulating statistics for a STL averages thread sizes and dependency arc lengths, hiding per-thread variance or linear changes in these values. Finally, the limited history of heap access *store timestamps* and collecting dependency statistics in only two bins (*t-1* and < *t-1*) limit the analysis' accuracy on distant thread dependencies.

```
for( i=0; i < limit; i++ ){
    if( i % n != 0){
        A[i] = A[i-1];
    }
}
```

Figure 9 –TEST analysis may incorrectly conclude this loop to be non-parallel. Parallelism exists at every n^{th} iteration, but the count of dependencies to the previous iteration is high.

In practice, the lost precision does not appear to decrease TEST's ability to identify good STLs. Parallel loops with complex dependencies that might fool TEST analysis were not found in any critical regions of the benchmark programs. While many of the benchmarks, particularly the integer programs (e.g. MipsSimulator, Huffman, db and NumHeapSort), exhibit STLs with highly varying thread sizes and dependency arc lengths, this did not affect our ability to identify the best decomposition. The accuracy of distant thread dependencies was also not critical. For floating point applications, parallelism existed at many levels and granularities, and selected decompositions were mostly limited by speculative buffer limits. In integer applications, we found that available

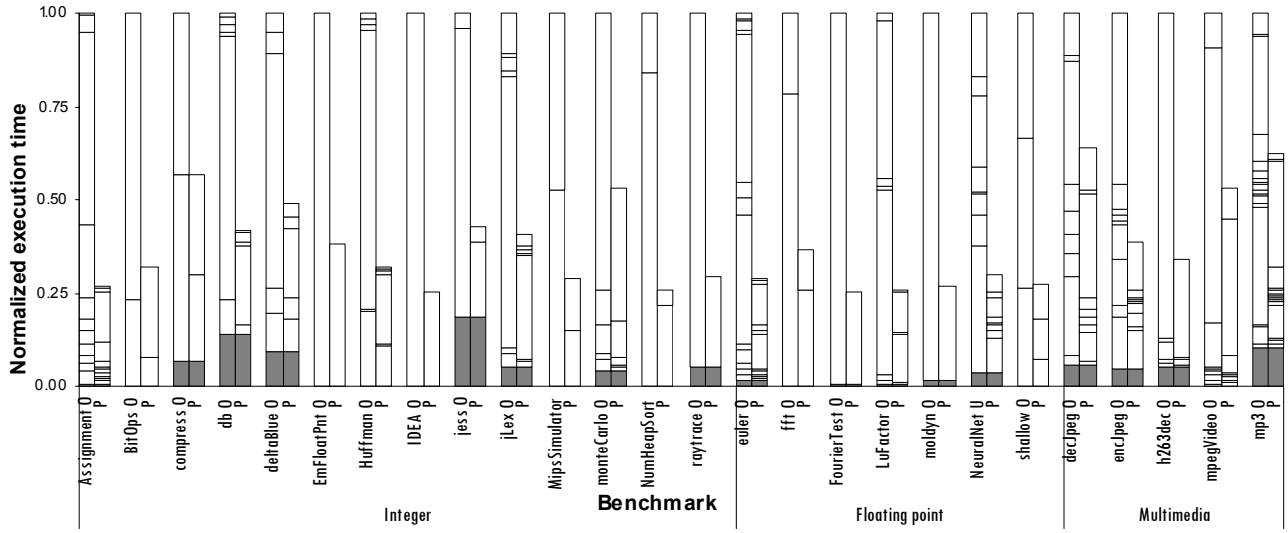


Figure 10 – Graphical representation of selected STLs. The 1st column (O) is sequential execution and the 2nd column (P) is predicted speculative execution. Each block in a column represents one STL and its contribution to execution time. The bottom dark block represents non-parallelizable serial execution (if any).

parallelism was mostly determined by dependency behavior to recent, not distant, past threads.

6.3. Guiding Optimization

TEST generates statistics that aid compiler optimization of selected STLs. The extended implementation of TEST can bin STL dependencies according to the load PC (and consequently, to specific program regions and variables). Dependency *critical arc lengths* relative to the thread size can also be collected. For frequently occurring *critical arcs* significantly less than the size of the loop, these statistics direct the compiler to variables where optimized placement of loads and stores can extend *critical arcs* [30][10] or where synchronization can be inserted to minimize violations [22].

These statistics are also invaluable for speculative programmer optimizations beyond that which can be uncovered by compiler analysis or can be performed safely by automatic transformations. Feedback from our simulations of TEST has helped identify false dependencies and aided the optimization of several benchmarks, specifically NumericSort, Huffman, db, and MipsSimulator. In these applications, the statistics quickly identified one or two critical dependencies that could be restructured or removed to expose parallelism to the speculation hardware.

7. Conclusions

This paper describes Tracer for Extracting Speculative Threads (TEST), a hardware tracer that analyzes potential speculative-thread loops (STLs) in running Java applications. The proposed hardware requires minor additions to our CMP with speculation support and incurs only moderate slowdown of a program during analysis. The results show that the proposed hardware can select decompositions that maximize speedup and parallel coverage without exceeding speculative buffer limits. The Java runtime system can then dynamically recompile selected decompositions into speculative threads.

Using this approach, we can identify speculative parallelism dynamically in integer and floating-point applications. A traditional parallelizing compiler would be

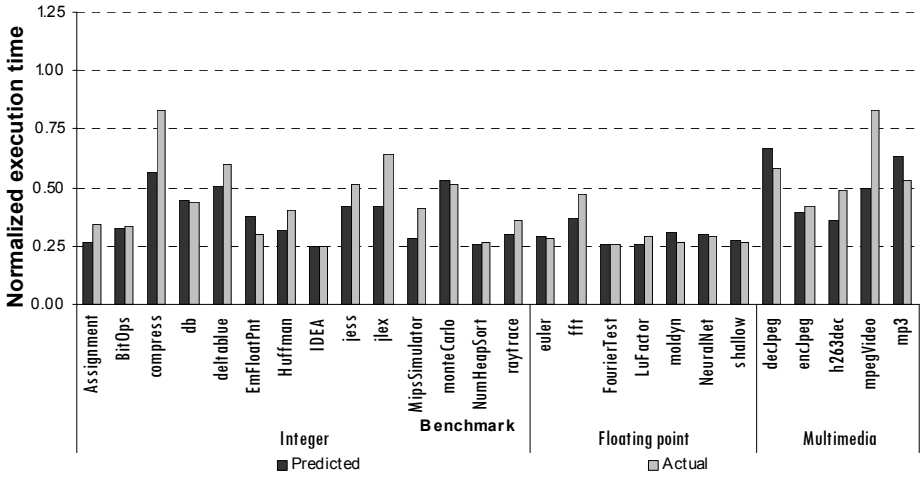


Figure 11 – Estimated speedup versus actual speedup.

challenged because of the difficulty of analyzing integer programs statically, and profiling techniques without hardware support would execute too slowly for run-time analysis. Our experiences with the analyzer also suggest it can enhance compilation by providing statistics to guide speculative optimizations and by producing feedback to aid programmers with optimizations that cannot be performed automatically.

8. Acknowledgements

This work was supported by DARPA Air Force Contract F29601-01-2-0085.

9. 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 Supercomputing '98. November 1998.

[3] Allen, R and Kennedy, K. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann, 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 Workshop on Languages and Compilers for Parallel Computing. Ithaca, NY, August 1994.

[6] Chaudhry et al. Space-Time Dimensional Computing for Java Programs on the MAJC Architecture. In Java Microarchitectures, Kluwer Academic Publishers, Boston, MA, April 2002.

[7] Chen, M. and Olukotun, K. Exploiting Method-level Parallelism in Single-threaded Java Programs. In PACT'98, Paris, France, October 1998.

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

[9] 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.

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

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

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

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

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

[15] Hammond, L., Willey, M., and Olukotun, K. Data Speculation Support for a Chip Multiprocessor. In ASPLOS'98, San Jose, CA, October 1998.

[16] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In ISCA-17, pp. 364-373, Seattle, WA, June 1990.

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

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

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

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

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

[22] 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.

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

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

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

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

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

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

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

[30] Steffan, J. G. et al. Improving Value Communication for Thread-Level Speculation. In HPCA'02, Cambridge, MA, February 2-6, 2002.

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

[32] Vijaykumar, T. N. and Sohi, G. S. Task Selection for a Multiscalar Processor. In MICRO'98, Chicago, IL, August 10-13, 1998.

[33] Wolfe, M. J. High Performance Compilers for Parallel Computing. Addison-Wesley, Redwood City, CA, 1996.

[34] Zhai, A. et al. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In ASPLOS'02, San Jose, CA, October, 2002.