Exploiting Method-Level Parallelism in Single-Threaded Java Programs

Michael K. Chen and Kunle Olukotun
Computer Systems Laboratory, Stanford University
Stanford, CA 94305-4070
http://www-hydra.stanford.edu/

Abstract

Method speculation of object-oriented programs attempts to exploit method-level parallelism (MLP) by executing sequential method invocations in parallel, while still maintaining correct sequential ordering of data dependencies and memory accesses. In this paper, we show why the Java virtual machine is an effective environment for exploiting method-level parallelism, and demonstrate how method speculation can potentially speed up single-threaded, general purpose Java programs. Results from our study show that significant speedups can be achieved on data-parallel applications with minimal programmer and compiler effort. On control-flow dependent programs, moderate speedups have been achieved, suggesting more significant performance improvements on these types of programs may come from more careful analysis or re-coding of the application. For both classes of applications, we discover performance debugging drastically improves speedups by eliminating or minimizing dependencies that limit the effectiveness of method speculation.

1 Introduction

In this paper, we investigate the effectiveness of using method speculation running on a chip multiprocessor [12] to exploit method-level parallelism (MLP) in single-threaded, general purpose Java programs. Method speculation might be thought of as the next logical step beyond current superscalar processors that exploit instruction-level parallelism (ILP). As depicted in Figure 1, methods correspond to blocks of many instructions. Coarse grain parallelism found between method blocks can potentially lead to speedups not available to superscalar processors. Studies [18] [9] have shown that instruction level parallelism (ILP) in superscalar processors is ultimately bounded by the limited size of the instruction window and control-flow dependencies. The lack of hardware for effective memory disambiguation also limits the parallelism available to these processors. Even with large instruction windows, current superscalar processors are architecturally designed to resolve dependencies between registers, not memory locations. Likewise, bus-based multiprocessors may be good at exploiting thread-level parallelism, but they are ineffective on loop-level and method-level parallel tasks because of the relatively high cost of communication. With a low-latency communication network and method speculation support, a speculative chip multiprocessor configuration can exploit levels of parallelism not available to superscalar processors or traditional bus-based multiprocessors [12].

Current microprocessors [5] exploit instruction level parallelism using out-of-order execution. The processor selects instructions from a large instruction window, speculatively executes these instructions out-of-order, and buffers the results in a reorder buffer. The instructions are then committed to the permanent architectural state in the original program order. In the case of instructions that are incorrectly executed due to branch mispredictions or the use of stale values, the processor must back up and restart execution.

Whereas the granularity of an entry in the reorder buffer of a superscalar machine corresponds to a single instruction, such an entry for a speculative multiprocessor analogously corresponds to a single speculative task. In method speculation, sequential method invocations are mapped to speculative tasks that are executed in parallel with the in-order thread. When execution reaches a method marked as speculative, the in-order thread continues to execute that method, but forks off a new speculative task that executes in parallel starting from the method return (continuation). Speculative memory stores and register file writes encountered during execution are buffered with each speculative task. These changes are committed to the head, in-order thread when sequential execution reaches the point at which the speculative task would have executed normally without speculation

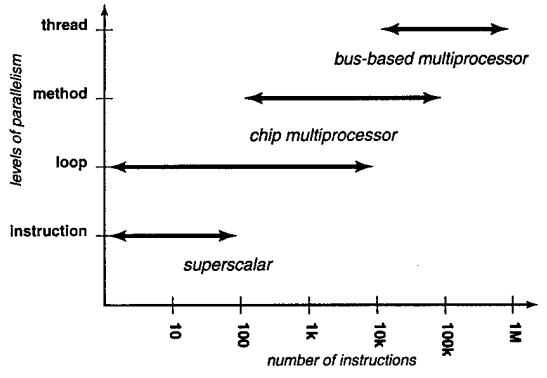


Figure 1 - A single chip multiprocessor can exploit levels of parallelism not available to traditional superscalar processors or bus-based multiprocessors.

support.

To guarantee correct parallel execution of these speculative tasks, our hardware ensures stores from earlier tasks are forwarded to reads by later tasks. Because this special hardware guarantees correct sequential ordering of memory references, we can forgo any explicit synchronization that is usually required for correct parallel execution on traditional parallel architectures. If a memory read-after-write (RAW) violation occurs (caused by a preemptive load by a later speculative task to a shared memory location written late by an earlier speculative task), the speculative task is aborted and restarted so that it can load the correct value from this memory location. To restart a speculative task, buffered stores from the aborted speculative task are discarded, and the register file and program counter (PC) are restored to their state prior to the start of speculative execution.

Franklin and Sohi first proposed the basis of hardware speculation in the context of the Wisconsin Multiscalar project [16] [3]. Their architecture is tailored more to speculating on relatively fine-grained tasks. Our work is based on a speculation model proposed for a chip multiprocessor, a design targeted to speculate on coarser-grained tasks [4]. Details of this design and more careful analysis of assumptions are discussed in [4] and [14].

We believe the Java language and virtual machine environment can serve as a vehicle to explore capabilities of speculation in a real system. Java will enable us to examine speculation performance for object-oriented programs (OOP), create a clean execution model for method speculation, and develop a realistic runtime system to dynamically manage method speculation.

Results from our study show that significant speedups can be achieved through method speculation on data-parallel applications with minimal programmer and compiler effort. On control-flow dependent programs, moderate speedups have been achieved, suggesting more significant performance improvements on these types of programs may come from more careful analysis or re-coding of the application. For both classes of applications, we discovered performance debugging drastically improves speedups by eliminating or minimizing dependencies that limit the effectiveness of method speculation.

In Section 2, we describe our motivation for studying method speculation under Java. The assumptions about the target architecture and simulation methodology used to evaluate speculation are discussed in Section 3. In Section 4, we describe our benchmark suite, with the results of our study given in Section 5. Closing remarks are made in Section 6, and future plans are presented in Section 7.

2 Method Speculation of Java Programs

While this paper concentrates on using Java for method speculation, it is important to briefly discuss why we believe a chip multiprocessor architecture is an ideal platform for general, high-performance Java computing. A chip multiprocessor architecture supports low latency communication between processors [12]. Such a parallel architecture is ideal for supporting many features of the Java language and Java virtual machine:

- Explicit thread model and synchronization primitives in the Java language allow the programmer to easily write programs to exploit the underlying multiprocessor hardware.
- Low latency inter-processor communication can reduce the overhead for accessing locks in the virtual machine. Our evaluations confirm Hölzle et al.'s findings that show such overheads can represent a significant fraction of overall execution time [7].
- Virtual machine data structures shared between application threads can be cached in the shared L2 cache, reducing access latency to these system resources.
- The overheads of many coarse-grained virtual machine operations like class loading, bytecode verification, garbage collection and just-in-time (JIT) compilation could be hidden by executing them concurrently with actual application execution.

Many obvious levels of parallelism exist in the virtual machine. Unfortunately, most of the coarse-grained parallelism present in the virtual machine only represents single event parallelization opportunities. As Amdahl's Law tells us, the effects of speeding up these phases of execution will have a smaller performance impact on long running applications because most of the execution time will be spent executing application code. While multithreaded Java applications will clearly benefit from a multiprocessor, method speculation can provide the following additional advantages:

- Facilitates easy and straightforward parallelization. Modern parallelizing compilers have been most successful with scientific applications. For many classes of general applications, though, these compilers fail because they cannot analyze non-uniform memory accesses patterns and cannot resolve memory pointer ambiguities. Without a compiler, significant programmer effort is required to explicitly hand-parallelize applications using Java thread and synchronization primitives. Worst of all, parallelization results in programs that are difficult to understand. Method speculation is a simpler programming model that can expose loop-level and method-level parallelism in the application so that it can be exploited by the underlying hardware. As we shall see, most programs can use method speculation with little or no modification to the original application.
- Reduces parallelization overheads. Thread and synchronization primitives generally introduce significant overheads into the execution time not present in the sequential version of a program. Method speculation can guarantee that dynamic execution dependencies will always be correct, with smaller overheads then those introduced by locks and barriers. Without these costly synchronization overheads, we can also expect to see speedups on finer grains of parallelism that would not be possible using traditional parallelization methods.
- Can speed up control-flow limited programs that have very little obvious parallelism. Programmers should be able to benefit from the multiprocessor architecture even when running single-threaded applications with very little fine- or coarse-grained parallelism.

Java will enable us to examine method speculation performance for object-oriented programs (OOP), create a clean execution model for method speculation, and develop a realistic runtime system to dynamically manage method speculation.

Using procedure or function calls as a framework to parallelize programs was first mentioned by Knight in the context of Lisp [8]. Oplinger et al. has also examined loop and procedural speculation through a limit analysis based on C programs using an ideal environment with many simplifying assumptions [13]. The focus of their study was to show that general applications exhibit significant amounts of loop- and procedural-level parallelism. Since this paper is more concerned with implementing a real system, our analysis will rely on more realistic assumptions given in Section 3.

2.1 Method Speculation on Object-Oriented Programs

Object-oriented programs represent a class of applications that may behave differently from general C programs under method (procedure) speculation. A study by Calder et al. shows the characteristics of C and C++ programs to be significantly different [2]. They show the dynamic function size of C programs to be four times that of C++ programs, and the frequency of procedure calls and returns in C++ to be three times that of C programs. These findings support our understanding of how object-oriented programs are written. The encapsulation model increases the frequency of small method calls, reducing the dynamic function size and increasing the number of method calls and returns.

These characteristics suggest that method invocations can efficiently expose loop- and method-level parallelism in object-oriented programs. Method speculation uses the notion of speculative tasks. When a method marked as speculative is encountered, the current processor executes the method and a forked speculative task executes in parallel starting from the method return (continuation). This mapping of methods to speculative tasks is depicted in Figure 2. If the method call returns a value, the speculative task executes assuming a predicted return value based on previous executions of that method. If this predicted return value turns out to be incorrect or there is a RAW violation in the heap due to the ordering of field loads and stores between tasks executing in parallel, then the speculative task must be terminated and restarted.

Method speculation is most effective for frequently executed methods that return `void` or a predictable return value. These types of return values are frequently found in methods returning `boolean` values that check for infrequent cases, but are not crucial to the flow of program execution. A method call like `isValid()` might do checks to make sure certain conditions are met, returning true 95% of the time. Another example is a method `isEOF()` that signals the presence of another value in the data stream. This method will almost always return true when iterating through a large data structure.

Method speculation can permit sequences of read only or write only methods to execute in parallel without worrying about

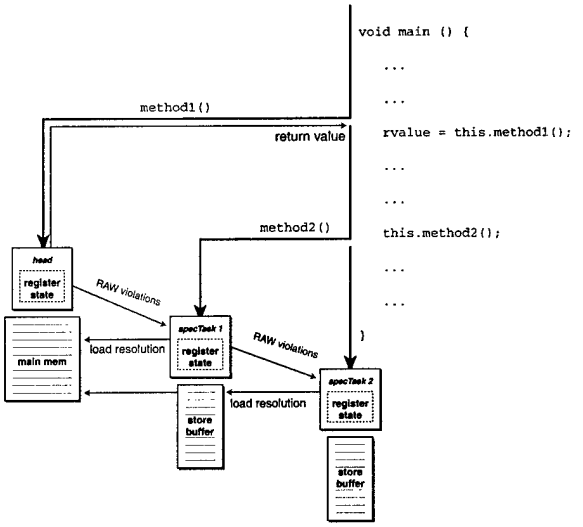


Figure 2 - Mapping of methods to speculative methods.

RAW violations. Traditional loop-level parallelism can even be exploited within our method speculation framework with minimal programmer effort or with modern parallelizing compilers, as we shall show later.

2.2 Mapping Speculation to Method Invocations

Using method calls as the granularity of speculative tasks conveniently allows us to exploit characteristics of the Java virtual machine specification so that no transformations to the source classfiles are required for speculative execution. In Oplinger et al.'s study, static analysis of the C source programs was required to convert a normal program into a single-program-multiple-data program suitable for running on a speculative machine [13] [14]. This analysis can be simplified considerably in Java. The Java Virtual Machine Specification [10] states:

- A method can only access its private Java stack and locals, or heap allocated objects. A callee method shares values from the caller only through explicitly passed arguments using a copying, pass-by-value convention and a single return value. Arguments and return values may include references to heap allocated objects shared between the caller and callee.
- Specific Java bytecodes (`getfield`, `putfield`, `getstatic`, `putstatic`, `arrayload`, and `arraystore`) make object (heap) accesses explicit and distinct from Java stack and local operations (`load`, `store`, `push`, `pop`, etc.).

These restrictions allow us to simplify our simulator by eliminating Java stack and local accesses from our dependency analysis. Since caller and callee methods work with their own private Java stack and locals, RAW violations can only occur between speculative tasks through objects dynamically allocated in the heap. This simplified analysis is not possible with C programs since the ability to manipulate pointers makes it impossible to guarantee that heap and execution stack accesses are well-behaved.

2.3 Dynamic Management of Method Speculation

Previous performance results for speculation [13] assumed optimally chosen method granularities taken from the dynamic execution profile. Our simulations suggest that choosing the appropriate methods for method speculation can dramatically affect performance gains. The Java virtual machine serves as an ideal platform for investigation of a runtime profiling system that can dynamically identify methods for method speculation from Java bytecodes using some set of heuristics.

Our system would be similar to the runtime system of the HotSpot Java virtual machine that uses dynamic method call frequencies to identify methods to compile just-in-time [7] [17]. Relevant statistics collected during runtime would be fed back to mark or unmark methods to speculate on so that our system can dynamically adapt to new data sets, or converge on an optimum runtime configuration. Profile information collected during execution could also be returned to the programmer and/or user to help increase the effectiveness of speculation.

We have developed a basic profiling system used to help identify methods to speculate on. An effective runtime profiling system preserves a delicate balance between collecting the most accurate and detailed data, and limiting the overhead data collection incurs on overall execution time. Since our goal is to identify large regions in the application program suitable for method speculation, we do not insert our profile annotations on small methods, methods returning unpredictable values (e.g. `float`, `long` and `double`), and methods included in the `java.*` package. This reduced our profile annotations to only methods that we expect could possibly benefit from speculation. Data collected from annotations placed on these methods are shown in Table 1. Simulations comparing execution times indicate only minimal impact (< 1%) on the overall execution time from these profiling annotations. While we expect that in the future, profile data would be used in algorithms to dynamically identify speculative methods, data collected from profiling is only used as a heuristic in this study to statically identify methods for speculation prior to execution.

3 Simulation Methodology

We use trace-driven simulations to evaluate the performance of method speculation for Java programs. Annotated traces from sequential execution of Java benchmarks running on a virtual machine are fed into a simulator that models a speculative multiprocessor. Although our trace-driven simulator does not model low-level details of the speculation hardware and memory system, it provides accurate estimates of the performance of method speculation.

3.1 Choosing a Virtual Machine

For a performance critical application like a virtual machine, an intrusive mechanism to generate traces would have skewed our results. SimOS allows us to generate traces with no overhead because it fully models the operating system and hardware units in software [15]. SimOS has complete support for the MIPS ISA and IRIX operating system, so our choice of virtual machines was narrowed to the two virtual machines available for this platform: the freely available *kaffe* (ver. 0.9.2) [19], and the Sun JDK1.1.3 port to IRIX. Both of these virtual machines support just-in-time (JIT) compilation and Sun's JDK1.1 APIs. We chose *kaffe* because we could not obtain the full source code for Sun's Java virtual machine (JVM), and this study and future work require modifications to the virtual machine.

Only results for method speculation with the JIT compiler enabled are presented for two reasons. We believe that speculation is only useful to further enhance speedups achieved by proven techniques like JIT compilation. Furthermore, our analysis has shown that results from simulations based on interpretive execution do not accurately reflect the performance of method speculation with JIT compilation.

3.2 Speculative Hardware Model

Hardware support for speculation makes it possible for the chip multiprocessor to correctly resolve memory dependencies, and to back-out of memory violations and restore the memory system to a previous, known state. Our evaluation of method speculation uses the underlying assumptions for a speculative

Annotation	Action	Purpose	Assembly Instructions per Method	Memory References per Method
Fixed overhead	Load pointer to profile data buffer		2	none
Dynamic access counter	Count call frequency of given method	Identify frequently executed methods	3	1 load, 1 store
Nested method counter	Count number of nested methods called from given method	Identify methods with elapsed execution times suitable for speculation	6 + 1 * method calls	2 loads, 2 stores
Loop counter	Counter numerber of backward branches taken in given method	Identify methods with elapsed execution times suitable for speculation or methods that can be modified to run under speculation	4 + 1 * method calls	1 load, 1 store
Prediction accurarcy counter	Count number of mispredicted return values	Identify methods that will not frequently cause return value misprediction violations under speculation	2 (correct) 6 (mispredict)	2 loads (correct) 2 loads, 2 stores (mispredict)

Table 1 - Profile annotations used to identify suitable methods for speculation.

	Operations	Overhead in Cycles
Start speculative method	Create and save checkpoint of register file to memory, fork a free processor that executes speculatively, load checkpoint into new processor.	~50
End speculative method	Commit speculative data, check actual return value against guess and restart if necessary.	~50
RAW violation	Discard speculative data, reload checkpoint, restart speculative task	~50

Table 2 - Speculation overheads per speculative method.

chip multiprocessor [4] to define the high-level behavior of speculative methods:

- The system has four single-issue, in-order MIPS R4000 processors, permitting up to three speculative tasks to execute in parallel with the main thread. Future designs of the chip multiprocessor could potentially use out-of-order processors with higher issue width, so that the system could exploit both ILP and MLP.
- New speculative tasks are created only from the most speculative task, or from the in-order thread, if no speculative task exists. Although not discussed here, our hardware can actually support other models of speculation [4].
- Memory store buffers can hold up to 1kByte (256 words) of writes to memory for each speculative task.
- A RAW memory violation forces a restart of the speculative task on which the violation occurred and termination of speculative tasks that occur sequentially after this task. This involves flushing all the corresponding memory store buffers and restoring the initial register state and PC of the speculative task that caused the violation.
- The unit of coherency that RAW violations in memory are detected is one word (4 bytes). Consequentially, byte accesses in the same word could potentially cause false violations.
- A simple return value prediction scheme is implemented that predicts using the last value returned for a given method [11].
- Speculation overheads are incurred for starting new speculative tasks and for restarting tasks due to RAW violations or return value mispredictions [4]. These overheads are described in detail in Table 2.

3.3 Trace-Driven Simulation

To simulate method speculation, we use *kaffe* running under SimOS to generate an execution trace containing data relevant to method speculation. Embra, the fastest but least detailed SimOS CPU model, was used to minimize simulation time. The execution trace is fed into a simulator that reconstructs execution under method speculation, with appropriate detection of RAW violations and return value mispredictions. To generate this trace, we utilize the annotation capabilities of SimOS and modified the native code generated by the JIT compiler.

As described in Section 2.2, our execution model simplifies data dependency analysis to heap object and array accesses between speculative methods. The execution trace only records these heap accesses, which can be easily distinguished from Java stack and local access. This eliminates the need to examine loads and stores to non-heap allocated memory (e.g. execution stack accesses, constant loads, and method lookups) in the dynamically generated code that cannot represent true dependencies between speculative tasks.

This sparse trace is generated by modifying the JIT compiled code generated by *kaffe*. Illegal instructions, used as markers, are added to the generated code immediately after loads and stores associated with field accesses during JIT compilation. During execution under SimOS, these marker instructions are trapped within SimOS. The trapped instructions call a routine to log to our execution trace file the addresses of the memory locations accessed by the real loads and stores. Because SimOS is a full software simulation environment, we can force these marker instructions to disappear from simulated execution. Thus, the only side effect from our annotation methodology is mild code expansion of the JIT compiled code during simulated execution.

The start and end of speculative methods are marked in a similar fashion. Speculative methods are chosen statically prior to execution using statistics collected from the basic profiling system described in Section 2.3 and another program that takes dynamic method calls from the execution trace to generate a readable method call graph with call frequencies and associated execution times. Speculative methods are identified to the virtual machine by denoting one of the unused method attribute flags as the method speculation flag. These attribute flags are normally used to identify certain characteristics of the method (e.g. `private`, `static`, `synchronized`). Classfile binaries are modified prior to simulation with the method speculation flag raised on methods that have been chosen to be speculative. When our modified JVM encounters the speculation flag raised during JIT compilation of a given method, the JIT compiler inserts assembly code to mark the beginning and end of the speculative region. This code is used to generate appropriate entries in the execution trace and to determine if the return value is correctly predicted. In a real system, the JIT compiler would insert assembly code at these points to invoke method speculation on the actual hardware.

Our initial simulation results were difficult to interpret because we could not associate violation addresses that our simulator generated with actual variables in the program. This led us to develop an extensive non-intrusive symbol facility under SimOS to aid in performance debugging. This is more challenging than looking for symbols in a standard program binary. Java symbols for method calls and static fields from a classfile have to be resolved dynamically to the corresponding addresses generated for these structures at runtime. Addresses for new objects and arrays created from program execution also have to be resolved to the corresponding text symbol. SimOS annotations [6] set on stub functions in *kaffe* are used to collect symbol and address information during execution into a symbold file so that these dynamically

allocated objects can be identified. To simplify symbol resolution, we also disable garbage collection so that objects are not relocated during runtime.

Virtual machine support functions are eliminated from the execution traces that we analyzed so that we can focus on the behavior of the JIT compiled application code under speculation. In addition to disabling the garbage collector, methods are loaded and compiled into the virtual machine in advance to avoid pauses during program execution. Overheads from small functions to implement new object allocations, lock operations and symbol lookups are also removed from the traces.

The execution trace files are analyzed offline using our method speculation simulator. The simulator reconstructs execution under method speculation from the sequential execution trace with marked speculative regions and object accesses. This trace-driven simulator correctly handles RAW violations and return value mispredictions. It also incorporates assumptions about the behavior of the underlying hardware described in Section 3.2. A symbol resolver works in conjunction with the simulator, taking information from the symbol file to associate dynamically allocated objects with the appropriate text symbol. Using this system, textual information can be produced to associate a violation address with a specific object, and with the method and call nesting from which the reference was made.

4 Benchmarks

Benchmark selection was largely limited by the availability of representative Java applications. In general, Java applications that represent traditional benchmark-style programs with compute intensive kernels and critical sections of code were difficult to locate. To date, Java has been most successful as a high-level development language used to implement user-interfaces. Unfortunately, user-interactive programs are not amenable to compute intensive benchmarking. We also included some small kernels in our benchmarks, but we avoided toy benchmarks like `CaffeineMarks` that do not reflect the structure or behavior of real Java programs.

The results in this paper are based on the benchmarks listed in Table 3. `StringBuffer` and `Hashtable` are frequently encountered core Java libraries that can be sped up using method speculation. `IDEA` and `NeuralNet` are two benchmarks taken from the `jBYTEmark` suite [1], and `LinpackApp` is a popular floating-point kernel. The remaining programs represent popular, full scale applications.

5 Performance Results

The results of our simulations of method speculation on a chip multiprocessor with four single-issue processors are shown in Table 4. Speedups are measured relative to one single-issue MIPS R4000 processor executing the benchmark. We show results with and without the speculation overheads (see Table 2) included in the analyzed traces. Average utilization is also computed, providing a measure of the occupancy on the available processors. As we would expect, applications that frequently abort large speculative regions due to memory RAW violations or return value mispredictsion will have significantly higher processor utilization relative to the actual speedup achieved.

These results represent the best performance that was achieved by varying which methods to speculate on. The pool of suitable candidates for speculation were identified by our call graph tool discussed in Section 3.3 and basic profiling system described in Section 2.3.

5.1 Performance Analysis

Speedups appear to be split between the two types of applications represented in our benchmarks. Method speculation is effective for speeding up data-parallel applications like `IDEA`, `NeuralNet`, `RayTrace` and `LinpackApp` that have significant coarse-grained parallelism. With very few data dependencies between speculative tasks, almost no memory violations are seen during execution of these programs.

`RayTrace` is the only benchmark of these four that has little ILP. Since `NeuralNet`, `IDEA`, and `LinpackApp` have significant ILP, it could be argued that similar speedups on these programs could be achieved on a superscalar processor. What is important to note is that method speculation can achieve these speedups without true compiler support. Most JIT compilers found in Java virtual machines only translate bytecodes into native code, sacrificing high-level optimizations that improve code quality in order to keep compilation times short. Superscalar processors, unfortunately, must rely on time consuming instruction scheduling and optimizations from a good compiler to fully exploit ILP. We believe this distinction can allow the speculative chip multiprocessor to outperform a superscalar machine on these types of data-parallel applications when only a simple JIT compiler is used.

Control-flow based programs, unfortunately, only benefit modestly from method speculation. Speedups on these programs do not exceed 1.4 on our chip multiprocessor with four CPUs. Compared to data-parallel applications, they have significantly higher violation and restart rates (see Table 4), reflecting the control and data dependent nature of these programs. Low processor utilization numbers also indicate that speedup is limited due in part

name	binary (bytes)	# methods (static)	# classes (static)	# methods (dynamic)	description
Hashtable	17507	19	4	3779	java.util.Hashtable
IDEA	18637	26	4	14051	encryption/decryption [1]
Jasmin	64635	309	63	585648	Java assembler
JavaCUP	117471	355	40	540034	yacc-like compiler
javac	573120	1492	174	236357	Java compiler
javap	172505	259	38	295726	classfile disassembler
LinpackApp	5109	15	2	13190	Linpack FP kernel
NeuralNet	25867	33	4	97332	neural net simulation [1]
OROMatcher	61277	202	20	96735	Perl 5 regexp
RayTrace	13319	53	6	1541478	raytrace of scene
StringBuffer	5906	36	2	4800	java.lang.StringBuffer

*NOTE: These statistics do not include calls to core JDK methods or classes.

Table 3 – Benchmark programs.

benchmark		speedup (with speculation overhead)	speedup (no speculation overhead)	average utilization (on 4 CPUS)	violations (% of speculative tasks)	restarts (% of speculative tasks)	violation (% of speculative cycles)	restart (% of speculative cycles)	average cycles / speculative task	average cycles / violation	average cycles / restart	notes
Hashtable	(original)	1.59	1.69	2.54	50.1	0.0	55.0	0.0	2271	2490	0	frequent violations on Hashtable.count
Hashtable	(modified)	2.30	2.47	2.72	3.2	0.0	14.2	0.0	2348	10239	0	moved loop-carried dependency
IDEA	(original)	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	speculative tasks were too large, resulting in store buffer overflow
IDEA	(modified)	3.60	3.76	3.76	0.0	0.0	0.1	0.0	8164	52773	0	transformed loop bodies into methods
Jasmin		1.23	1.24	1.35	41.0	7.8	26.8	4.0	1663	1086	850	
javac		1.15	1.15	1.68	29.9	4.4	59.6	18.1	23147	46097	96240	
JavaCUP		1.23	1.23	1.57	16.1	12.4	33.1	25.8	2909	5958	6037	
javap		1.29	1.29	1.84	38.9	3.5	62.8	2.5	5854	9456	4190	
LinpackApp		2.04	2.09	2.21	4.7	1.9	2.8	6.7	5163	3071	18606	large sequential section reduces speedup from > 3 in the kernel
NeuralNet	(original)	1.61	1.55	2.18	54.5	0.0	53.2	0.0	27726	27059	0	violations concentrated on certain variables
NeuralNet	(modified)	2.82	3.10	3.11	0.2	0.0	0.7	0.0	2679	8361	0	transformed loop bodies into methods
OROMatcher		1.22	1.18	1.33	10.3	12.0	45.2	0.3	2427	10606	70	violations concentrated on certain variables
RayTrace	(original)	1.01	1.01	1.95	99.9	0.0	98.8	0.0	20408	20169	0	frequent violations on Point.x
RayTrace	(modified)	2.77	2.80	2.80	0.0	0.0	0.0	0.0	15745	0	0	eliminated false loop-carried dependency
StringBuffer	(original)	1.47	1.63	2.43	65.2	0.0	56.2	0.0	1709	1471	0	frequent violations on StringBuffer.count
StringBuffer	(modified)	2.14	2.46	2.68	7.8	0.0	13.6	0.0	1375	2420	0	moved loop-carried dependency

Table 4 - Performance results.

to the few available opportunities to use method speculation. A cursory examination confirms that these applications have proportionally fewer methods with `void` and predictable return values than data-parallel programs. This suggests that with more time to understand and modify the source code so that processor utilization is increased, it may be possible to see better performance on control-flow based programs.

We discovered that the size of methods chosen for speculation plays an important role in performance and processor utilization. Small speculative tasks are less likely to cause memory violations, but only produce small speedups. Two factors seem to contribute to this phenomenon. Since small speculative regions are short-lived, there is less opportunity to overlap speculative tasks, resulting in low utilization of the processors. Secondly, small speculative regions suffer a larger relative penalty from the fixed speculation overheads.

In general, we found better results come from speculating on larger speculative tasks. Large speculative regions execute longer, so that there is a greater chance of having numerous speculative methods executing concurrently. Unfortunately, large regions have more memory references, increasing the amount of parallel, overlapping loads and stores to the execution heap. Large speculative regions also tend to move memory references further apart in time from their original positions in sequential execution. Consequently, speculating on larger methods increases processor utilization, but places a greater strain on the memory system and speculation hardware. As expected, we found that choosing excessively large speculative regions results in speculation buffer overflows and unacceptable rates of memory RAW violations.

Speedups never approach four, the number of processors in our system, even for our data-parallel benchmarks. Our simple scheduling algorithm bypasses speculative methods and executes them sequentially when no more free processors are available, resulting in idle gaps that limit speedup. This limitation is due to an inefficiency in our current model for speculative methods in loops that we believe will be remedied as we continue to develop our system.

5.2 Modifying Source Code to Improve Performance

We were disappointed with our initial results on unmodified source code. We found low processor utilization numbers for our data-parallel benchmarks and frequent memory violations on control-flow based benchmarks that had very regular loops. This led us to look more closely at the program source code. With some experimentation, we found that relatively simple modifications to the code could significantly boost performance under method speculation.

For data-parallel applications, we discovered that the original methods often represented regions that were too large to speculate on. Inspection of the code revealed smaller data-parallel loops that are closer to granularities of parallelism suitable for our system. To expose this loop-level parallelism under method speculation, we encapsulate the loop body that we want to represent a single task as a nested method call, as shown in Example 1. `IDEA`, `NeuralNet` and `LinpackApp` show significant improvements by speculating on these finer-grained parallel tasks.

Our simulator also identified variables that frequently caused memory violations under speculation for certain control-flow-based applications. For `Hashtable` and `StringBuffer`, these violations corresponded to variables that are written to late in a method, precluding significant overlap between successive iterations. As illustrated in Example 2, moving writes to the dependent variable to an earlier point and reads to this variable to a later point in the method can increase overlap. For `Raytrace`, we eliminated a false inter-method dependency by moving a write from the loop body into the body of the method, as shown in Example 3. By moving such dependencies between speculative methods in our benchmarks, violations are either eliminated, or occur earlier in execution,

```
private void do_mid_forward(int patt)
{
    for (int neurode = 0; neurode < MID_SIZE; neurode++)
    {
      do_mid_forward_iteration(patt,neurode);
    }
}

private void do_mid_forward_iteration(int patt, int neurode)
{
  double sum = 0.0;
  for (int i = 0; i < IN_SIZE; i++)
    {
      sum += mid_wts[neurode][i] * in_pats[patt][i];
    }

  sum = 1.0 / (1.0 + Math.exp(-sum));
  mid_out[neurode] = sum;
}
```

Example 1 - Loop body of `do_mid_forward()` transformed into a nested method call.

resulting in improved speedups.

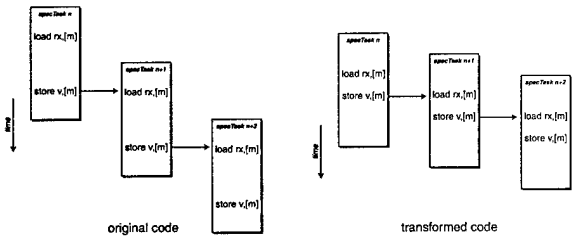
In future studies, we hope to show that program sites that may benefit from these types of optimizations can be identified automatically, with minimal effort from the programmer.

5.3 Limitations of Our Study

Our choice to use the *kaffe* virtual machine dictated the absolute performance of our system. Our experiments have shown that Sun's JDK with JIT compiler support is about 20-30% faster than *kaffe*. For this study, though, we believe a reasonably implemented JIT is sufficient for comparing the relative performance of a chip multiprocessor with method speculation to a conventional single-issue processor.

The effects of virtual machine support functions were eliminated from the traces used to generate results. Including these functions would make it more difficult to interpret the results since they tend to reflect overheads specific to the *kaffe* Java virtual machine implementation. In a real system, it is also hard to predict how these support functions will affect performance, since they could either be hidden by speculation or extend sections of sequential execution.

We will examine the effects of optimized JIT compiler code and inclusion of virtual machine support functions on method speculation as our system becomes more fully developed.



Example 2 - Moving a loop carried dependency to improve method speculation performance.

6 Conclusions

This study describes how the Java virtual machine can be an effective environment for exploiting method-level speculation. In our model, method invocations are used as a convenient abstraction of the tasks that we would like to speculate on. Not only does this framework simplify data dependency analysis, but it also results in a clean execution model. In conjunction with a JIT compiler, this system can invoke method speculation with virtually no modification to the source bytecode program.

We show that speculation based on method invocations can achieve significant speedups on data-parallel applications with minimal programmer and compiler effort. For control-flow limited applications, we show that method speculation can produce modest speedups. For both application classes, we find that analysis of runtime behavior helps to eliminate and minimize dependencies that limit speedup gains. Our preliminary experiences suggest that more study is required to understand how non-data-parallel applications behave under method speculation so that performance on these type of applications can be improved.

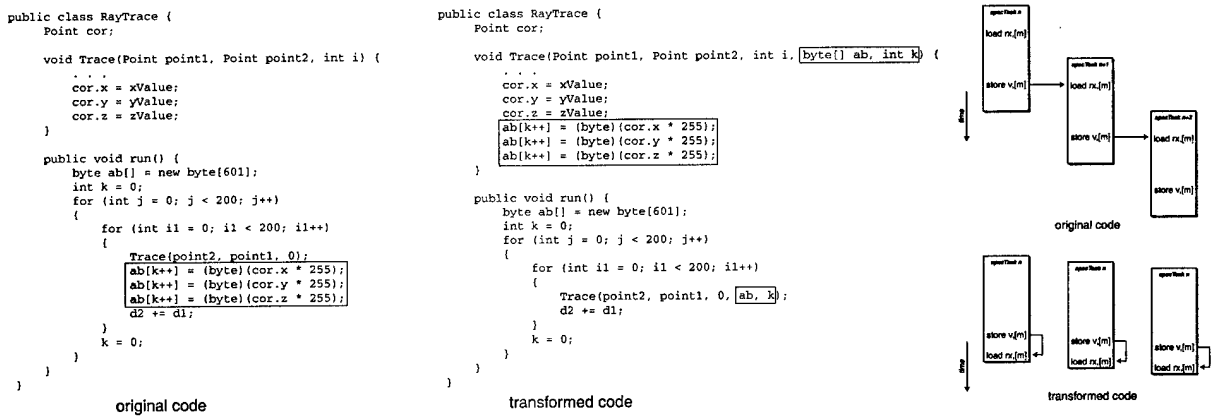
Although superscalar processors may perform as well as our system on data-parallel applications that have significant instruction level parallelism (ILP), instruction scheduling from smart compilers are usually required to fully exploit ILP on superscalar processors. Since JIT compilers generally sacrifice these high-level optimizations for faster compilation, we believe that a speculative chip multiprocessor can outperform a superscalar processor in this configuration. It should also be noted that speedups from method-level parallelism (MLP) are largely orthogonal to those resulting from ILP, so that an implementation of a speculative chip multiprocessor using superscalar processors could take advantage of both types of parallelism.

7 Future Work

Continued research on method speculation of Java programs will progress along two related fronts: improving the performance of method speculation on Java programs, and implementation of a virtual machine targeted for a chip multiprocessor architecture that can dynamically manage method speculation.

Improving the performance of method speculation on Java applications without obvious data parallelism appears to be the most interesting and challenging area of further study. We expect that additional performance improvements on these programs will result from combining incremental speedups from several techniques that we are currently exploring. These techniques include using algorithms to reliably identify methods that benefit from speculation, applying general code transformations, such as those described in Section 5.2, that improve speculation performance, and expanding the applicability of speculation to a larger set of method call sites.

We have also started development on a Java virtual machine designed specifically for a chip multiprocessor architecture. An



Example 3 - Eliminating a false loop carried dependency (to `ab[]`, an array of bytes).

integrated low overhead profiling and feedback system with a new JIT compiler will dynamically manage method speculation. The new compiler will addresses performance limitations of our current JIT [7] [17], and will allow us to study the effectiveness of speculation on optimized JIT code. This virtual machine will also look beyond explicit Java threads to parallelism within the virtual machine. By enabling concurrent execution of explicitly coarse-grain virtual machine tasks like the JIT compiler, class loader, garbage collector and bytecode verifier, this virutal machine, together with the chip multiprocessor, will be able achieve speedups even on single threaded applications.

Acknowledgments

The authors wish to thank Basem Nayfeh for discussions that led to many of the ideas that appear in this paper. This work was supported by DARPA contract DABT63-95-C-0089.

References

[1] Byte Magazine. jBYTEmarks v0.9. Available at http://www.byte.com/, 1995.

[2] Brad Calder, Dirk Grunwald, and Benjamin Zorn. "Quantifying Behavioral Differences Between C and C++ Programs." Technical Report, University of Colorado, Boulder, Jan 1994.

[3] M. Franklin and G.S. Sohi. "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism." In *Proceedings of 19th Annual ISCA*, pages 56-67, Gold Coast, Australia, May 1992.

[4] Lance Hammond and Kunle Olukotun. "Data Speculation Support for a Chip Multiprocessor." To appear in *Proceedings of ASPLOS-VIII.*

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach 2nd Edition*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[6] Stephen A. Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Jinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben Verghese. The SimOS Simulation Environment. Available at http://simos.stanford.edu/, Feb 10 1996.

[7] Urs Hölzle, Lars Bak, Steffen Grarup, Robert Griesemer, and Srdjan Mitrovic. "Java On Steroids: Sun's High-Performance Java Implementation." In Hot Chips IX, Stanford California, August 25-26 1997.

[8] Tom Knight, "An Architecture for Mostly Functional Languages," Proceedings of the ACM Lisp and Functional Programming Conference, pages 500-519, August 1996.

[9] M. S. Lam and R. P.Wilson. "Limits of Control Flow on Parallelism." In *Proceedings of 19th Annual ISCA*, pages 46-57, Gold Coast, Australia, May 1992.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value locality and load value prediction." In *Proceedings of ASPLOS-VII*, pages 138-147, Cambridge MA, October, 1996.

[12] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kun-Yung Chang. "The Case for a Single-Chip Multiprocessor." In *Proceedings of ASPLOS-VII*, October 1996.

[13] Jeffrey Oplinger, David Heine, Monica S. Lam and Kunle Olukotun. "In Search of Speculative Thread-Level Parallelism." Computer Systems Laboratory Technical Report CSL-TR-98-765, Stanford University, July 1998.

[14] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam and Kunle Olukotun. "Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors." Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University, February 1997.

[15] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[16] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors." In *Proceedings of 22nd Annual ISCA*, pages 414-425, Ligure, Italy, June 1995.

[17] David Ungar and Randall B. Smith. "Self: The Power of Simplicity." In *Proceedings of OOPSLA '87, ACM SIGPLAN Notices*, pages 227-242, December 1987.

[18] D.W. Wall. "Limits of Instruction-Level Parallelism." Digital Western Research Laboratory, WRL Research Report 93/6, November 1993.

[19] Tim Wilkinson. The kaffe Virtual Machine v0.9.2. Available at http://www.kaffe.org/ or http://www.transvirtual.com/, 1997.