A Concurrent Trace-based Just-In-Time Compiler for JavaScript

Jungwoo Ha
Department of Computer Sciences
The University of Texas at Austin
habals@cs.utexas.edu

Mohammad R. Haghighat
Software Solution Group
Intel Corporation
mhaghigh@intel.com

Shengnan Cong
Software Solution Group
Intel Corporation
shengnan.cong@intel.com

Kathryn S. McKinley
Department of Computer Sciences
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

JavaScript is emerging as the ubiquitous language of choice for web browser applications. These applications increasingly execute on embedded mobile devices, and thus demand responsiveness (i.e., short pause times for system activities, such as compilation and garbage collection). To deliver responsiveness, web browsers, such as Firefox, have adopted trace-based Just-In-Time (JIT) compilation. A trace-based JIT restricts the scope of compilation to a short hot path of instructions, limiting compilation time and space. Although the JavaScript limits applications to a single-thread, multicore embedded and general-purpose architectures are now widely available. This limitation presents an opportunity to reduce compiler pause times further by exploiting cores that the application is guaranteed not to use. While method-based concurrent JITs have proven useful for multithreaded languages such as Java, trace-based JIT compilation for JavaScript offers new opportunities for concurrency.

This paper presents the design and implementation of a concurrent trace-based JIT that uses novel lock-free synchronization to trace, compile, install, and stitch traces on a separate core such that the interpreter essentially never needs to pause. Our evaluation shows that this design reduces the total, average, and maximum pause time by 88%, 97%, and 93%, respectively compared to the base single-threaded JIT system. Our design also improves throughput by 5% on average and up to 36%, because it delivers optimized application code faster. This design provides a better end-user experience by exploiting multicore hardware to improve responsiveness and throughput.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental compilers, code generation.

General Terms Design, Experimentation, Performance, Measurement

Keywords Just-In-Time Compilation, Multicore, Concurrency

1. Introduction

JavaScript is emerging as the scripting language of choice for client-side web browsers [10]. Client-side JavaScript applications initially performed simple HTML web page manipulations to aid server-side web applications, but they have since evolved to use asynchronous and XML features to perform sophisticated, interactive dynamic content manipulation on the client-side. This style of JavaScript programming is called AJAX (for Asynchronous JavaScript and XML). Companies, such as Google and Yahoo, are using it to implement interactive desktop applications such as mail, messaging, and collaborative spreadsheets, word processors, and calendars. Because Internet usage on mobile platforms is growing rapidly, the performance of JavaScript is critical for both desktops and embedded mobile devices. To speed up the processing of JavaScript applications, many web browsers are adopting Just-In-Time (JIT) compilation, including Firefox TraceMonkey [5], Google V8 [11], and WebKit SFE [20].

Generating efficient machine code for dynamic languages, such as JavaScript, is more difficult than for statically typed languages. For dynamic languages, the compiler must generate code that correctly executes all possible runtime types. Gal et al. recently introduced a trace-based JIT compilation for dynamic languages to address this problem and to provide responsiveness (i.e., low compiler pause times and memory requirements) [7]. Responsiveness is critical, because JavaScript runs on client-side web browsers. Pause times induced by the JIT must be short enough not to disturb the end-user experience. Therefore, Gal et al.'s system interprets until it detects a hot path in a loop. The interpreter then *traces*, recording instructions and variable types along a hot path. The JIT then specializes the trace by type and translates it into native code in linear time. The JIT sacrifices code quality for linear compile times, rather than applying heavy weight optimizations. This trace-based JIT provides fast, light-weight compilation with a small memory footprint, which make it suitable for resource-constrained devices.

On the hardware side, multicore processors are prevailing in embedded and general purpose systems. The JavaScript language however lacks a thread model, and thus all JavaScript applications are single-threaded. This limitation provides the opportunity to perform the JIT and other VM services concurrently on another core, transparently to the application, since the application is guaranteed not to be using it.Unfortunately, state-of-the-art trace-based JIT compilers are sequential [7, 5, 19], and have not exploited concurrency to improve responsiveness.

In this paper, we present the design and implementation of a concurrent trace-based JIT compiler for JavaScript that combines responsiveness and throughput for JavaScript applications. We address the synchronization problem specific to the trace-based JIT compiler, and present novel lock-free synchronization mechanisms for wait-free communication between the interpreter and the compiler. Hence, the compiler runs concurrently with the interpreter reducing pause times to nearly zero.

Our mechanism piggybacks a single word, called the *compiled state variable (CSV)*, on each trace, using it as a synchronization variable. Comparing with CSV synchronizes all of the compilation actions, including checking for the native code, preventing duplicate traces, and allowing the interpretation to proceed, without using any lock.

We introduce lock-free *dynamic trace stitching* in which the compiler patches new native code to the existing code. Dynamic trace stitching prevents the compiler from waiting for trace stitching while the interpreter is executing the native code, and reduces the potential overhead of returning from native code to the interpreter.

We implement our design in the open source Tamarin-Tracing VM, and evaluate our implementation using the SunSpider JavaScript benchmark suite [21] on three different hardware platforms. The experiments show that our concurrent trace-based JIT implementation reduces the total pause time by 88%, the maximum pause time by 93%, and the average pause time by 97% on Linux. Moreover, the design improves the throughput by an average of 2–7%, with improvements up to 36%. Our concurrent trace-based JIT virtually eliminates compiler pause times and increases application throughput. Because tracing overlaps with compilation, the interpreter prepares the trace earlier for subsequent compilation, thus the JIT delivers the native code more quickly. Consequently, the system traces and compiles more hot paths during execution. This approach also opens up the possibility of increasing the code quality with compiler optimizations without sacrificing the application pause time.

2. Related Work

Gal et al. proposed splitting trace tree compilation steps into multiple pipeline stages to exploit parallelism [6]. This is the only work we can find seeking parallelism in the trace-based compilation. There are a total of 19 compilation pipeline stages, and each pipeline stage runs on a separate thread. Because of data dependency between each stage and the synchronization overhead, the authors failed to achieve any speedup in compilation time. We show having a parallel compiler thread operating on an independent trace provides more benefit than pipelining compilation stages. With proper synchronization mechanisms, our work successfully exploited parallelism in the trace-based JIT by allowing tracing to happen concurrently with the compilation, even when only one compiler thread was used.

Kulkarni et al. explored maximizing throughput of background compilation by adjusting the CPU utilization level of the compiler thread [16]. This technique is useful when the number of application threads exceeds the number of physical processors and the compiler thread cannot fully utilize a processor resource. They conducted their evaluation on method-based compilation, though the same technique can be applied to trace-based compilation. However, because JavaScript is single-threaded, it is less likely that all the cores are fully utilized in today's multicore hardware. Hence, the effect of adjusting CPU usage levels will not be as significant as it is in multi-threaded Java programs.

c-AOTC performs an ahead-of-time compilation on the client side for the embedded systems [14]. When the method is first called, the compiler runs concurrently with the interpreter, and the compiled native code is cached in persistent memory. The following VM instances later reuse the code stored in the cache, saving compilation time. Code caching technique can be combined with our work, but the effect will not be as high as in method-based compilation. This is because the code reusable ratio is not as high because the changes in input may cause specializing types and paths to change.

A number of previous efforts have sought to reduce compilation pause time in method-based JIT. SELF-93 VM introduced adaptive compilation strategies for minimizing application pause time [13]. When a method is invoked for the first time, the VM compiles it without optimizations using a light weight compiler. If method invocations exceed a threshold, the VM recompiles the method with more aggressive optimizations. While the SELF-93 VM provided reasonable responsiveness, it must pause the application thread for compilation when initially invoked.

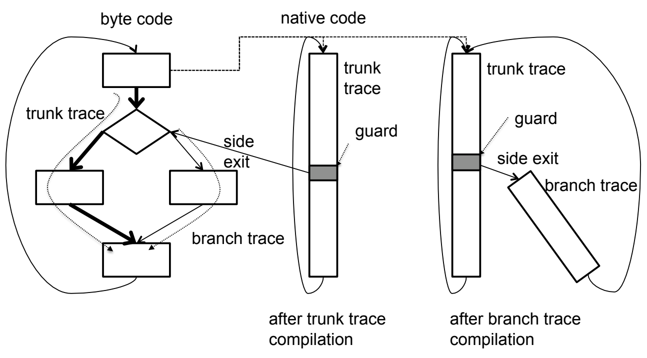

Figure 1. Byte code and native code transition in the trace-based JIT. Initially, the interpreter interprets on the byte code. First detected hot path (thick path) is traced forming a trunk trace. Following hot paths guarded and installed in a side-exit. The compiler attach the branch trace, which begins from the hot side-exit to the loop header, to the trunk trace.

Krintz et al. implemented profile-driven background compilation in the Jalapeño Virtual Machine (now called Jikes RVM) [15, 2]. In multiprocessor systems, a background compiler thread overlaps with application execution, which reduces compilation pause times. Jikes RVM also applied lazy compilation, where the JIT only compiles the method on demand within a class instead of compiling every method in a class at class loading time. When the method is invoked for the first time before the optimized code is ready, the VM pauses the application and run the baseline compiler.

These novel techniques have made adaptive compilation in method-based compilation practical in Java Virtual Machines, such as Sun HotSpot [17], IBM J9 [18], and Jikes RVM [1]. However, issues specific for trace-based JIT has not been successfully evaluated by any previous work.

3. Background

3.1 Dynamic Typing in JavaScript

JavaScript is a *dynamically* typed language. The type of every variable is inferred from its content dynamically. Furthermore, the type of JavaScript variables can change over time as the script executes. For example, a variable may hold an integer object at one time and later hold a string object. A consequence of dynamic typing is that operations need to be dispatched dynamically. While the degree of type stability in JavaScript is the subject of current studies, our experiences and empirical results indicate that JavaScript variables are type stable in most cases. This observation suggests that type-based specialization techniques pioneered in Smalltalk [4] and later used in Self [12] and Sun HotSpot [17] have the potential for tremendously improving JavaScript performance.

3.2 Trace-based JIT Compilation

Hotpath VM is the first trace-based JIT compilation introduced for Java applications in a resource-constrained environment [8]. The authors later explored trace-based JIT for dynamic languages, such as JavaScript [7].

The trace-based JIT compiles only frequently executed path in a loop. Figure 1 shows an example of how the interpreter identifies a hot path, and expands it. Initially, the interpreter executes the byte code instructions, and identifies the hot loop with backward branch profiling which operates as follows. When the execution reaches the backward branch, the interpreter assumes it a loop backedge and increments the counter associated with the branch target address. When the counter reaches a threshold, the interpreter enables tracing, and records each byte code instruction to a trace buffer upon execution. When the control reaches back to the address where the tracing started, the interpreter stops tracing and the compiler compiles the trace to native code. As the interpreter is not doing an exact path profiling, the traced path may or may not be the real hot path. The first trace in a loop is called a *trunk trace*.

Instructions are *guard*ed if they potentially diverge from the recorded path. If a guard is triggered, the native code *side-exit* back to the interpreter, and begin interpreting from the branch that caused the side-exit. The interpreter counts each side-exit to identify the frequent side-exit. When a side-exit is taken beyond a threshold, it means the loop contains another hot path, and the interpreter enables tracing from the side-exit point until it reaches the address of the trunk trace. This trace is called a *branch trace*. A branch trace is compiled and the code is stitched to the trunk trace at the side-exit instruction. As the interpreter finds more hot paths, the number of branch traces grows forming a *trace tree*.

Since the compilation granularity is a trace, which is smaller than a method, the total memory footprint of the JIT is smaller than that of method-based JITs. And because no control flow analysis is required, start-up compilation time is less than that of the method-based compilers. However, as optimization opportunities are limited, the final code quality may not be as good as code generated by method-based compilation. Therefore, trace compilation is suitable for embedded environments where resources are limited, or the initial JIT cost is far more important than the steady state performance.

4. Design

4.1 Parallelism to Exploit

To design a proper synchronization mechanism to maximize the concurrency, we must understand what parallelisms can be exploited. Figure 2 explains an execution flow example of sequential and concurrent JIT. As the compilation phase is offloaded to a separate thread, the interpreter is responsive and making progress while compilation happens, as is common for generic concurrent JIT compilers. For trace-based

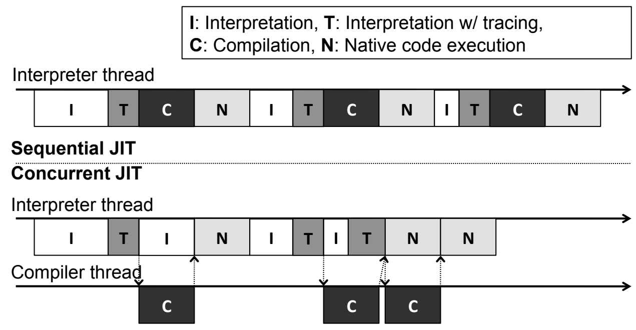

Figure 2. Example of sequential vs concurrent JIT execution flow.

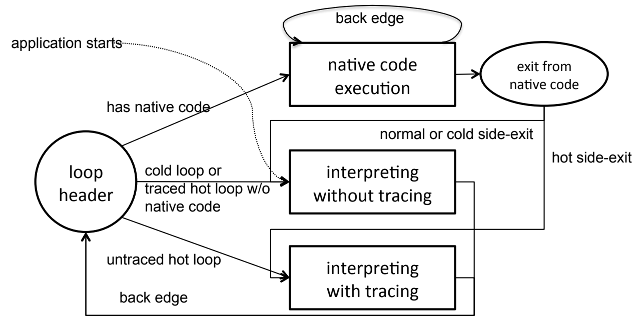

Figure 3. The interpreter state transition at a loop header.

JIT, tracing must precede the compilation phase. If tracing can happen concurrently with compilation, subsequent compilation may start earlier, and deliver the native code faster. Furthermore, more hot paths can be compiled during the execution. We can expect to achieve throughput improvements as well as a reduction in the pause time. The concurrent JIT also opens the possibility to do more aggressive optimizations without hurting pause time. The following sections explain how we designed the synchronization to achieve the parallelism shown in Figure 2.

4.2 Compiled State Variable

In the trace-based JIT compiler, the interpreter changes state at loop entry points. As shown in Figure 3, when the control flow reaches a loop entry point, the interpreter must identify four different states. First, if compiled native code exists for the loop, the interpreter calls it. The native code executes until the end of the loop or a side-exit is taken. Second, if the loop has never been traced and the loop is a hot loop, the interpreter executes byte code with tracing enabled. Identifying hot loop path is explained in Section 3 in detail. Third, if tracing is currently enabled at the loop header, the interpreter disables it and requests compilation. While compiling the trace, the interpreter continues to execute the program. Fourth, if the loop is cold, the interpreter increments the associated counter and keeps on interpreting the byte codes.

Checking all these cases at a loop header requires a synchronization with the compiler thread. Otherwise, race conditions may cause overhead or incorrect execution. For example, the interpreter may make duplicate compilation requests, or trace the same loop multiple times. The simplest synchronization method is using a coarse-grained lock around the checking routine. However, the lock can easily be contended after the compilation request is made, especially with a short loop body, because the control reaches the loop header frequently. We could use a fine-grained lock for accessing each loop data structure. However, this is also infeasible because the native code for the loop can change as the trace tree grows, and holding a lock while executing the native code would stall the compiler too often.

To overcome these challenges, we design a lock-free synchronization technique using a *compiled state variable (CSV)*. A word size CSV piggybacks on each loop data structure, and it is aligned not to cross the cache line. Thus, stores to it are atomic. The value of the CSV is defined as shown in Table 4.2. By following simple but efficient ways of incrementing the CSV value, the state check at the loop header can be done without any explicit synchronization. The initial value of CSV is zero, and only the interpreter increments 0 to 1 when it requests a compilation. As it is a local change, the interpreter sees the value 1 on the subsequent operations before the compiler sees the value 1. The compiler changes the value 1 to 2 after it registers the native code to the loop data structure. Thus, when the interpreter reads the value 2, it is guaranteed that the native code is ready to call. Therefore, the pause time for waiting is almost zero for both the interpreter and the compiler, maximizing the concurrency.

When the compiler makes a JIT request, the trace buffer is pushed to a queue before the CSV is incremented to 1. We use a simple synchronized FIFO queue for the JIT request, because it is normally not contended. However, a generic, concurrent, lock-free queue for one producer and consumer [9] could always replace this queue, but we think it would not affect performance.

4.3 Dynamic Trace Stitching

The trace-based JIT specializes types and paths, and injects guard instructions to verify the assumptions for the type and path of the trace. Guards trigger side-exit code if the assumption is not met, and returns the control back to the interpreter.

If two or more hot paths exist in a loop, the first hot path will be compiled normally, but the subsequent hot paths will frequently trigger guards. As explained in Section 3, the interpreter traces from the branch that caused the side-exit (branch trace), and compiles it. As more hot paths are revealed, trunk and branch traces form a trace tree. Recompiling the whole trace tree is good for the code quality, but the compilation time will grow quadratic if the whole trace tree is recompiled every time a new trace is attached to the tree. Also, this strategy would keep the trace buffer in memory for future recompilation, which is infeasible in memory constrained environments. Instead of recompiling the

Description	Action	CSV
has native code	Call native code	2
compilation already requested	normal interpretation	1
Hot loop	Enable tracing	0
Cold loop	normal interpretation	0
Trace enabled	Disable tracing and request compilation	0 to 1

Table 1. Value of Compiled State Variable(CSV) at a loop header.

whole tree, we use trace stitching technique. Trace stitching is a technique that compiles the new branch trace only, and patches the side-exit to jump to the branch trace native code.

Branch patching modifies code that is produced by more than one trace. Hence, it is probable that interpreter is executing the native code at the same time that the compiler wants to patch it. Naive use of a lock around the native code will incur a significant pause time on both the interpreter and the compiler. Waiting becomes a problem if time spent in the native code grows large, reducing the overall concurrency. The compiler may also make a duplicate copy of the code instead of patching, or delay the patching until the native code exits to the interpreter. Either method has inefficiencies, and we propose lock-free *dynamic trace stitching* for the branch patching. The key factor of dynamic trace stitching is that a side-exit jump is a safe point where all variables are synchronized to the memory. We use each side-exit jump instruction as a placeholder for the patching. When the compiler generates the native code for the branch trace, both jumping to the previous side-exit target or jumping to the branch trace code does not change the program semantic. Therefore, if the patching is atomic, the compiler can patch the jump instruction directly without waiting for the interpreter. If the branch target operand is properly aligned, patching is done by a single store instruction. There is no harmful data race even without any lock. With these benign data races, the interpreter and the compiler run concurrently without pausing.

5. Implementation

We implemented our design in the open-source Tamarin-Tracing Virtual Machine [19]. TamarinTracing VM conforms to the ECMAScript language standard, where JavaScript, JScript, and ActionScript are all dialect of the ECMAScript language. Our design implements Gal et al.'s sequential trace-based JIT [7] and targets the 32bit x86 architecture.

In this section, we discuss implementation details not explained in previous section.

Incrementing the Compiled State Variable The compiled state variable is piggybacked on both trunk and branch traces. We aligned the compiled state variable at a word granularity, and since the cache line is multiple of word size, CSV does not cross the cache line boundary. Therefore, it is safe to increment the variable without any lock prefix or using a compare-and-swap instruction. We simply declared the CSV `volatile` to force a memory load.

Dynamic Trace Stitching A direct branch instruction(`jmp`) is 5 bytes in 32bit x86, where the last 4 bytes are the branch target operand. Since the TamarinTracing VM's JIT compiler generates the machine code in reverse order, padding the branch instruction to align it with the cache line is difficult. Thus, we used compare-and-swap instruction to replace the branch target operand of the side-exit jump instruction.

GC Thread-Safety TamarinTracing uses a mark-sweep garbage collector, called *MMGC*, to manage application and VM objects. The current MMGC implementation is not thread-safe. To make the compiler concurrent, we have to make MMGC thread-safe or eliminate compiler objects from the MMGC heap. We choose the later, and use explicit allocation and deallocation in the compiler, i.e., malloc and free. This change to explicit memory management improves the throughput by $\sim$10%. To isolate the improvement caused by modifying the JIT, we use this sequential JIT with explicit memory management as the baseline of our evaluation in Section 6. Our concurrent JIT is implemented on top of this baseline.

Implementation Correctness To validate the correctness of our implementation, we used Intel Thread Checker to detect harmful race conditions. We also ran the acceptance test suite used to verify sequential Tamarin VM. Our implementation passes the same test entries that Tamarin VM passes. Furthermore, our concurrent JIT runs the same set of SunSpider benchmark suites correctly as compared to sequential JIT. Therefore, we believe our implementation produces the same results as the original Tamarin VM implementation.

6. Evaluation

6.1 Experiments Setup

We evaluate our implementation of the concurrent trace-based JIT on four different configurations:

- **Moblin**: Moblin (Linux 2.6 kernel) and NPTL pthread library running on Intel Atom 1.6GHz, which has one core, and two SMT threads, and a 512KB L2 cache.
- **Linux**: Ubuntu Linux 2.6 and NPTL pthread library running on Intel Core 2 Quad 2.4GHz, which has four cores and two 4MB shared L2 caches.

Benchmarks	Bytecode (bytes)	Compiled Traces	Compilation (%)	Native (%)	Interpreter (%)	Runtime (ms)
access-binary-trees	697	37	5.4	89.1	5.5	74
access-fannkuch	823	49	2.4	94.2	3.3	117
access-nbody	2,202	27	3.5	91.6	4.9	144
access-nsieve	543	14	1.4	96.8	1.7	56
bitops-3bit-bits-in-byte	414	6	4.0	89.7	6.3	12
bitops-bits-in-byte	385	15	1.5	96.5	2.1	40
bitops-bitwise-and	264	3	0.2	99.4	0.4	179
bitops-nsieve-bits	586	11	1.4	96.6	2.0	50
controlflow-recursive	504	35	8.3	84.5	7.3	28
crypto-aes	7,004	158	11.4	63.2	25.4	150
crypto-md5	5,470	6	24.6	17.4	58.0	120
math-cordic	832	9	1.8	95.0	3.1	32
math-partial-sums	758	11	1.3	93.2	5.5	41
math-spectral-norm	841	35	7.8	78.3	13.9	36
s3d-cube	4,918	188	8.4	41.6	50.0	155
s3d-morph	573	14	1.5	95.9	2.6	81
s3d-raytrace	7,289	147	9.3	68.1	22.6	170
string-fasta	1,426	22	1.9	95.6	2.5	141
string-validate-input	1,511	28	1.4	96.0	2.6	261

Table 2. Workload characterization of SunSpider benchmarks with sequential Tamarin JIT.

- **Windows**: Windows XP and Win32 thread libraries running on Intel Core 2 Duo 2.4GHz, which has two cores and a 2MB shared L2 cache.
- **Mac**: Mac OS X Leopard and pthread library on Intel Core 2 Duo 2.8GHz, which has two cores and a 4MB shared L2 cache.

Because Linux can be easily configured to reduce noise to a minimum, we ran 50 runs and averaged the results. On other configurations, because of the inevitable perturbation of the OS services, we picked the 10 best runs out of 50 and calculated the average. We also set the confidence interval at 95%, assuming Student's t-distribution on all results to validate the statistical significance. For easy comparison, all graphs are presented so that the lower bar represents the better result.

6.2 SunSpider Benchmarks Characterization

The SunSpider benchmark suite is a set of JavaScript programs intended to test performance [21]. It is widely used to test and compare the JavaScript engine on web browsers, such as Firefox SpiderMonkey, Adobe ActionScript, and Google V8. Table 2 characterizes the benchmarks in the Linux configuration.

Figure 4 shows the execution of the components for the sequential and concurrent JIT. The first bar represents the sequential JIT, and the second bar shows the interpreter thread activity in the concurrent JIT. This thread activity includes the interpreter, native code, and pause time caused by compilation requests. The third bar shows the compile time of the compiler thread. The y-axis value for concurrent JIT is normalized to the sequential JIT total execution time. Hence, bar 2 less than 100% is the speedup. The compilation time in bar 2 represents the total pause time in the concurrent JIT.

In most benchmarks, the concurrent JIT implementation improves both responsiveness and throughput. The pause time with the concurrent JIT is negligible, and speedups are noticeable in many benchmarks. In many cases, speedup results from the faster delivery of native code. For example, in `crypto-aes`, the compilation time in both sequential and concurrent is about the same, and the amount of speedup results from the compiler being offloaded. The `s3d-cube` benchmark shows the most speedup because the compilation time in the concurrent JIT is almost twice as much as in the sequential JIT. The compiler was able to compile more hot paths since the tracing overlapped with the compilation. As a result, the time spent in the interpretation is reduced significantly, which accounts for the speedup.

Notice that the larger, more complex benchmarks (`crypto-aes`, `crypto-md5`, `s3d-cube`, `s3d-raytrace`) are influenced most by the concurrent JIT. This trend indicates that as JavaScript programs grow in size and complexity, the concurrent JIT is likely to provide more benefits due to an increased fraction of native code execution.

6.3 Responsiveness

We evaluate application responsiveness using total, average, and maximum pause time. Total pause time for running a benchmark is a good indicator of the application's responsiveness, and the average reflects the end-user experience. Many small pauses are better than one big pause in terms of responsiveness [3]. We compare maximum pause time, which is the most noticeable pause to the end-user, therefore we want it to be as low as possible.

Figure 5 demonstrates that our concurrent JIT implementation reduces both maximum and total pause time significantly. The y-axis is the pause time normalized to the pause time in the sequential JIT. A value of 1.0 means that the pause time is the same, and 0.1 means the pause time is reduced by 90%. Tics at the top of each bar shows 95% confidence interval.

Geometric means in the Linux configuration show that we reduced the total pause time by 89% and 93% for the

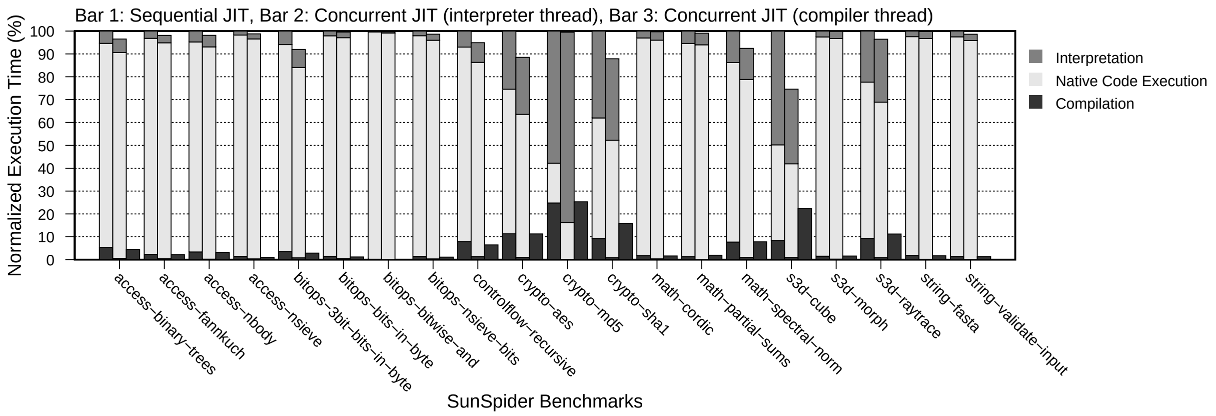

Figure 4. Average time break down in compilation, native code, and interpretation.

maximum pause time, showing a huge improvement in responsiveness. Furthermore, the average pause has reduced by 97% of the sequential JIT. Even in the worst case on Moblin with the `access-fannkuch` benchmark, the total pause time is reduced by 36%. Yet, the average is reduced by 87%, which shows the implementation successfully avoided long pauses.

As the compilation time per trace grows, it is likely that the concurrent JIT will reduce the pause time more. Table 2 shows that `crypto-md5` has the highest per trace compilation time, compiling six traces for 25% of the execution time. It also achieves the best reduction in pause time, with 99% for all three metrics.

6.4 Throughput

Improving responsiveness does not necessarily mean improving throughput. On the other hand, better responsiveness often requires sacrificing throughput, e.g., concurrent garbage collectors versus stop-the-world garbage collectors. However, we improved throughput as well as the responsiveness because our implementation executes the interpreter and tracing concurrently with the compiler.

Figure 6 shows the speedup for each configuration. The x-axis is the SunSpider benchmarks and the y-axis is the speedup normalized to the execution time of the sequential JIT. The concurrent JIT achieves 2–7% speedup on all platforms on average, and achieves up to 36% on `s3d-cube` on Windows. As explained in Section 6.2, the speedup in `s3d-cube` is due to increasing the number of compiled traces.

Another noticeable result is that the concurrent JIT improves performance uniformly. Only three benchmarks on Moblin, and two benchmarks on Mac have degradations, The performance variation of `crypto-md5` among the platforms is due to the fact that it only spends 17% of the time in compiled native code, and only compiles six traces. The remaining 18 programs improve or stay the same on all platforms except for Moblin. This is because SMT threads share hardware resources and is not fully parallel.

6.5 Effect of Using Multicores on Performance

This section presents the effect of thread scheduling on the compiler and interpreter threads in multicore systems. A major difference between multicore and traditional off-chip multiprocessors is memory latency. Especially in shared-cache multicore systems, load and store latency is significantly less than off-chip multiprocessors. Moreover, shared-cache multicores bring nonuniformity in load/store latency, which makes the performance less predictable.

The Intel Core 2 Quad processor that we used for the Linux configuration is a shared-cache multicore, where each pair of cores shares a L2 cache. Thus, core 0 and 1 communicate via the L2 cache, but core 0 and 2 go through the interconnect network. In the concurrent JIT, the compiler thread is a producer whose output is transferred to the instruction cache in the interpreter thread. Hypothetically, we can expect that lower communication latency will improve the performance.

Figure 7 shows how thread assignment influences the performance. Lack of operating support for hard pinning a thread to a given core, we could only hint the OS scheduler using `pthread_setaffinity_np` provided by NPTL pthread library. The first bar corresponds to the case when the compiler and interpreter threads are configured to share the L2 cache. The second bar represents the case where the compiler and interpreter do not share caches. The third bar is measured without any hint to the scheduler. In all benchmarks, the shared L2 configuration performed the best, though the benefit is small for most of the benchmarks. However, a couple of benchmarks, such as `bitops-3bit-bits-in-byte`, show a large difference. Performance is slightly stable in the shared L2 configuration. You can observe that the confidence interval is the narrowest in this configuration. Therefore, we recommend putting both compiler and interpreter threads on cores that share caches if the cache is big enough not to cause thrashing.

We also compare the total and maximum pause time in Figure 8. As in the performance comparison, the overall

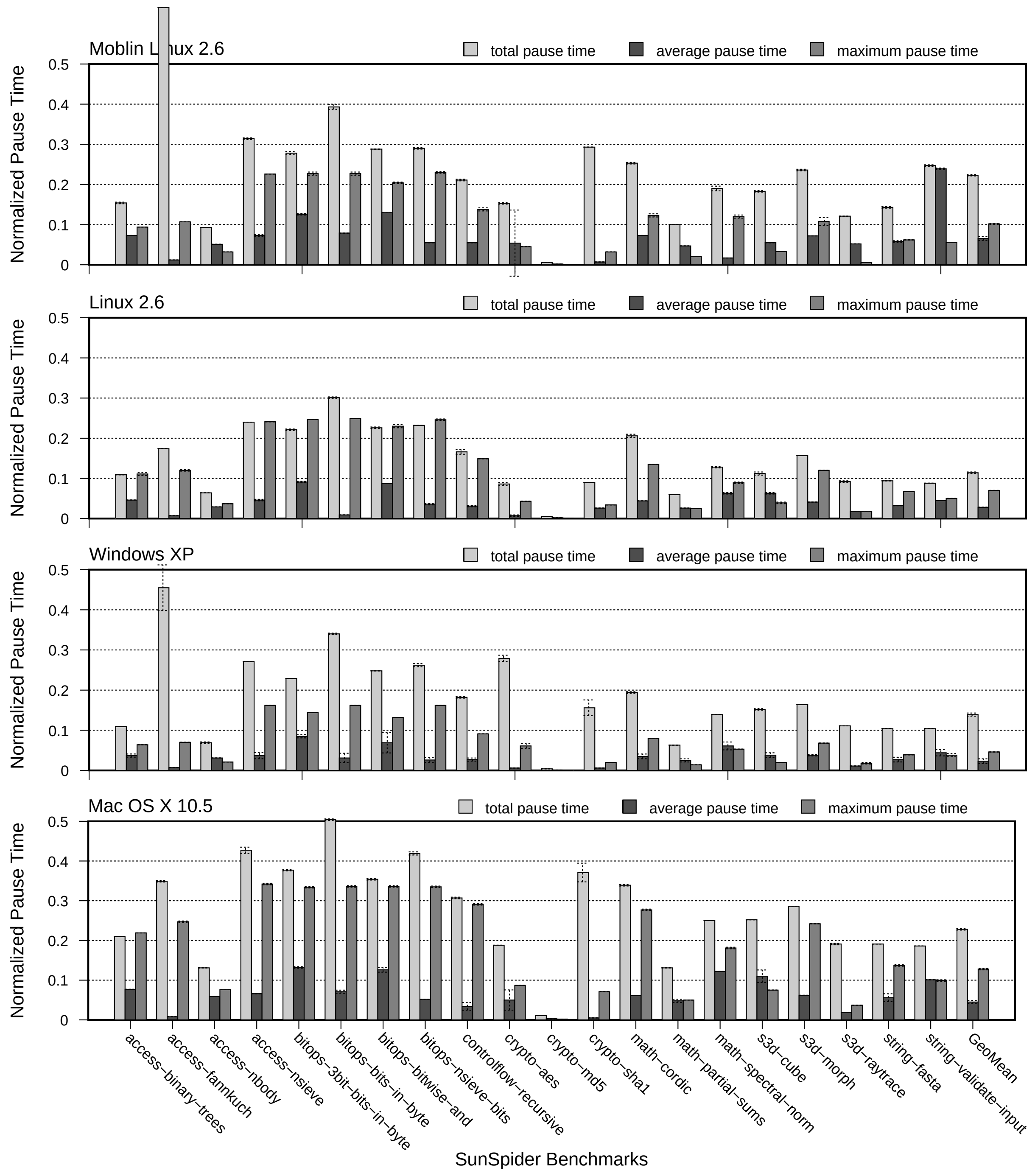

Figure 5. Pause time ratios of concurrent vs. sequential JITs.

reduction in total and maximum pause time for the shared L2 configuration is modest, and the measurements are slightly more stable. Even though improvement in overall pause time is small, it is as significant as 5% for some benchmarks.

Our results indicate that the choice of the right pair of cores for the compiler and interpreter threads can influence performance. The impact may increase in multicore systems with more complex memory hierarchies.

7. Conclusion

In this paper, we showed that even though JavaScript language itself is currently single-threaded, both its throughput and responsiveness can benefit from multiple cores with our concurrent JIT compiler. This improvement is achieved by running the JIT compiler concurrently with the interpreter. Our results show that most of the compile-time pauses can be eliminated, resulting in a total, average, and maximum re-

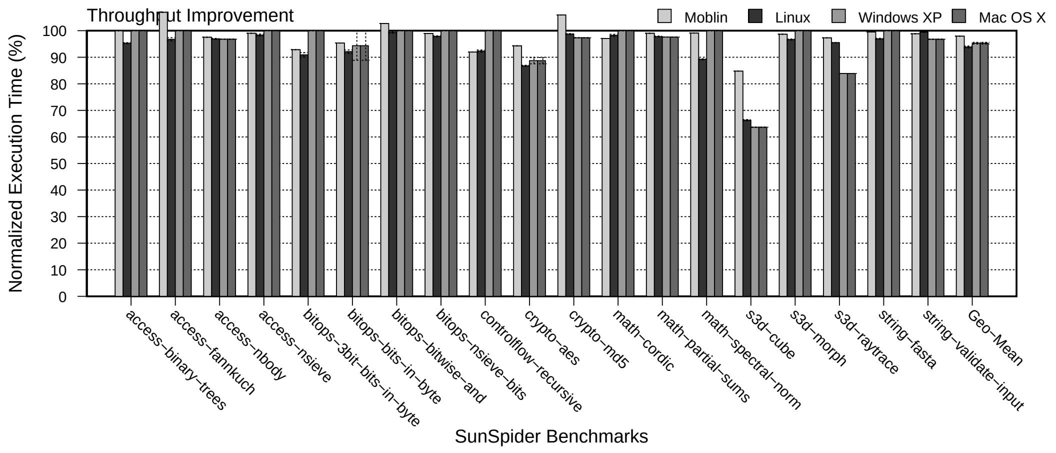

Figure 6. Execution time improvement with concurrent JIT.

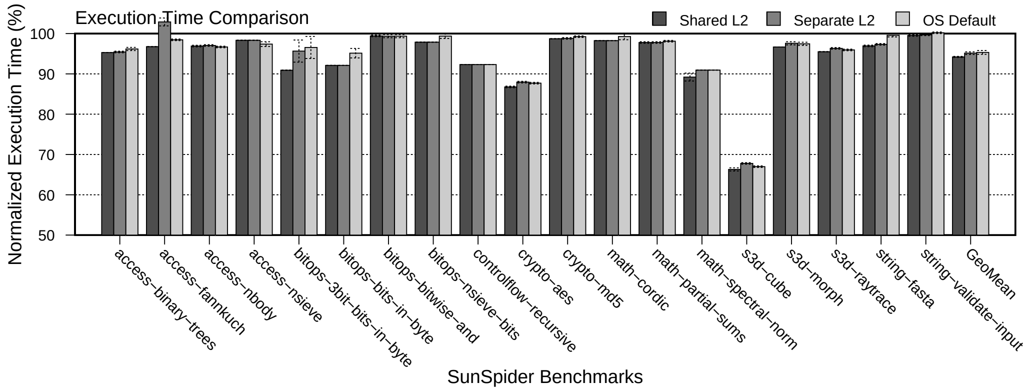

Figure 7. Effect of performance on various core configurations.

duction in pause time by 88%, 97%, and 93%, respectively. Moreover, the throughput is also increased by an average of 5%, with a maximum of 36%. This paper demonstrates a way to exploit multicore hardware to improve application performance and responsiveness by offloading system tasks.

References

[1] ALPERN, B., ATTANASIO, D., BARTON, J. J., BURKE, M. G., P.CHENG, CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J., SMITH, S., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeño virtual machine. *IBM System Journal 39*, 1 (Feb. 2000).

[2] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the Jalapeño JVM. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, US, 2000), ACM, pp. 47–65.

[3] CHENG, P., HARPER, R., AND LEE, P. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Canada, 1998), ACM, pp. 162–173.

[4] DEUTSCH, L. P., AND SCHIFFMAN, A. M. Efficient implementation of the smalltalk-80 system. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, UT, 1984), ACM, pp. 297–302.

[5] FOUNDATION, M. Tracemonkey, 2008. `https://wiki.mozilla.org/JavaScript:TraceMonkey`.

[6] GAL, A., BEBENITA, M., CHANG, M., AND FRANZ, M. Making the Compilation "Pipeline" Explicit: Dynamic Compilation Using Trace Tree Serialization. Tech. Rep. 07-12, University of California, Irvine, 2007.

[7] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., KAPLAN, B., HOARE, G., MANDELIN, D., ZBARSKY, B., ORENDORFF, J., JESSE RUDERMAN, SMITH, E., REITMAIER, R., HAGHIGHAT, M. R., BEBENITA, M., CHANG, M., AND

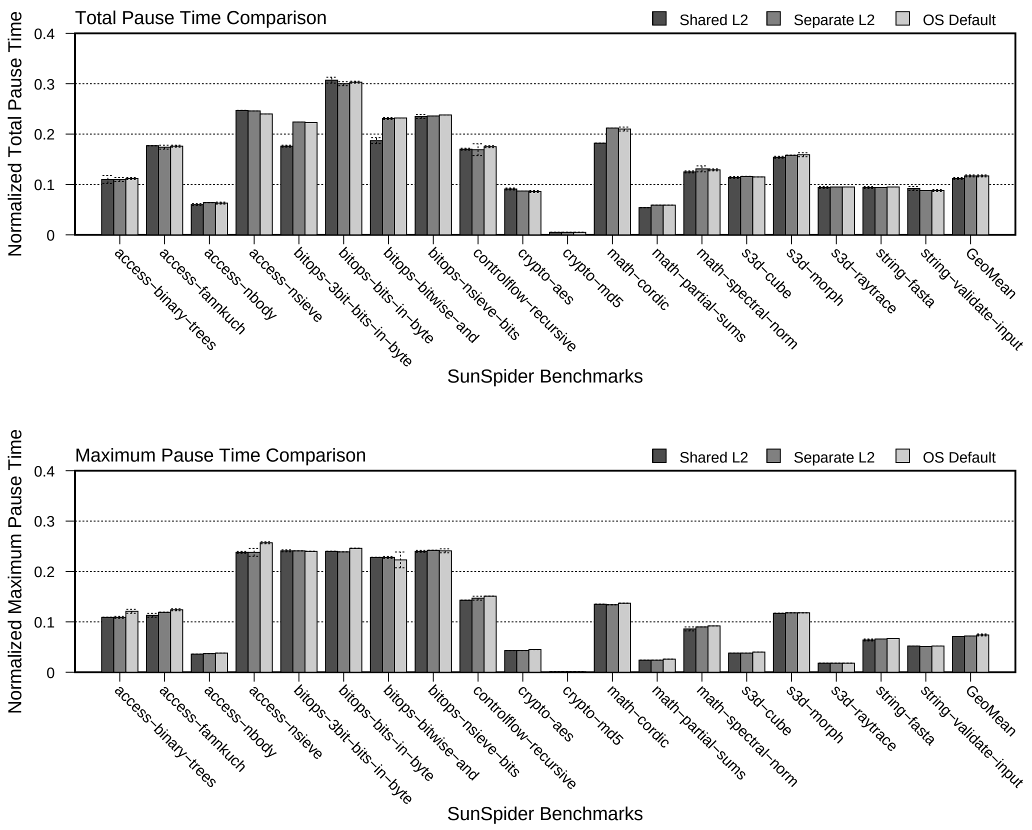

Figure 8. Effect of pause time on various core configurations.

FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* (Dublin, Ireland, 2009), ACM.

[8] GAL, A., PROBST, C. W., AND FRANZ, M. HotpathVM: an effective JIT compiler for resource-constrained devices. In *International Conference on Virtual Execution Environments* (Ottawa, Canada, 2006), ACM, pp. 144–153.

[9] GIACOMONI, J., MOSELEY, T., AND VACHHARAJANI, M. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA, 2008), ACM, pp. 43–52.

[10] GOODMAN, D. *JavaScript Bible*, 3rd, ed. IDG Books Worldwide, Inc., Foster City, CA, 1998.

[11] GOOGLE INC. V8, 2008. `http://code.google.com/p/v8`.

[12] HÖLZLE, U., AND UNGAR, D. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceesings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, FL, USA, 1994), ACM, pp. 326–336.

[13] HÖLZLE, U., AND UNGAR, D. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems 18*, 4 (1996), 355–400.

[14] HONG, S., KIM, J.-C., SHIN, J. W., MOON, S.-M., OH, H.-S., LEE, J., AND CHOI, H.-K. Java client ahead-of-time compiler for embedded systems. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, USA, 2007), ACM, pp. 63–72.

[15] KRINTZ, C., GROVE, D., LIEBER, D., SARKAR, V., AND CALDER, B. Reducing the overhead of dynamic compilation. *Software: Practice and Experience 31* (2001), 200–1.

[16] KULKARNI, P., ARNOLD, M., AND HIND, M. Dynamic compilation: the benefits of early investing. In *International Conference on Virtual Execution Environments* (San Diego, CA, 2007), ACM, pp. 94–104.

[17] PALECZNY, M., VICK, C., AND CLICK, C. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium* (Monterey, CA, USA, April 2001), Sun Microsystems, USENIX.

[18] SUNDARESAN, V., MAIER, D., RAMARAO, P., AND STOODLEY, M. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2006), IEEE Computer Society,

pp. 87–97.

[19] TAMARIN. Tamarin Project, 2008. `http://www.mozilla.org/projects/tamarin/`.

[20] WEBKIT. SquirrelFish Extreme, 2008. `http://webkit.org/blog/`.

[21] WEBKIT. SunSpider JavaScript Benchmark, 2008. `http://webkit.org/perf/sunspider-0.9/sunspider.html`.