2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory

Thomas Shull, Jiho Choi, María J. Garzarán†, Josep Torrellas
University of Illinois at Urbana-Champaign †*Intel Corp.*
http://iacoma.cs.uiuc.edu

***Abstract*—Scripting languages' inferior performance stems from compilers lacking enough static information. To address this limitation, they use JIT compilers organized into multiple tiers, with higher tiers using profiling information to generate high-performance code. Checks are inserted to detect incorrect assumptions and, when a check fails, execution transfers to a lower tier. The points of potential transfer between tiers are called *Stack Map Points (SMPs)*. They require a consistent state in both tiers and, hence, limit code optimization across SMPs in the higher tier.**

This paper examines the code generated by a state-of-the-art JavaScript compiler and finds that the code has a high frequency of SMPs. These SMPs rarely cause execution to transfer to lower tiers. However, both the optimization-limiting effect of the SMPs, and the overhead of the SMP-guarding checks contribute to scripting languages' low performance. To tackle this problem, we extend the compiler to generate hardware transactions around SMPs, and perform simple within-transaction optimizations enabled by transactions. We target emerging lightweight HTM systems and call our changes *NoMap*. We evaluate NoMap on the SunSpider and Kraken suites. We find that NoMap lowers the instruction count by an average of 14.2% and 11.5%, and the execution time by an average of 16.7% and 8.9%, for SunSpider and Kraken, respectively.

***Keywords*-JavaScript; Transactional Memory; Compiler Optimizations; JIT Compilation.**

I. Introduction

Scripting languages such as JavaScript, PHP, Ruby, Perl, and Python have become very popular in recent years. Programmers enjoy their flexibility, Read-Eval-Print Loop (REPL) support, and ease of use within an agile workflow. These languages are currently used in applications ranging from the low-power embedded domain [1], [2], [3], [4] to the performance-critical server-side [5], [6], [7] and scientific [8], [9], [10] domains.

One reason why scripting languages have become popular is that their performance has become acceptable, thanks mostly to software improvements. Indeed, techniques such as inline caching, type specialization, and Just-In-Time (JIT) support have been added to many scripting language implementations.

Despite these improvements, however, the steady-state performance of scripting languages still lags compared to that of statically-compiled languages such as C and C++. This is seen in Figure 1, which compares the steady-state execution time of the Shootout benchmarks [11] written in several languages. The figure shows the execution time

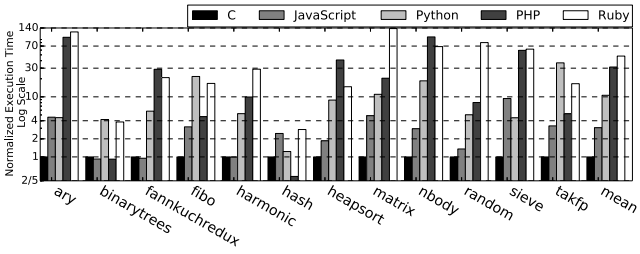


Figure 1: Execution time of the Shootout benchmarks.

of C, JavaScript, Python, PHP, and Ruby implementations normalized to C on a log scale. For each scripting language, we use a high-performance JIT, namely, JavaScriptCore [12] for JavaScript, PyPy [13] for Python, HHVM [14] for PHP, and JRuby [15] for Ruby. On average, JavaScript, Python, PHP, and Ruby take 3.1, 10.6, 31.4, and 47.7 times longer to execute than C, respectively. Note that the Python, PHP, and Ruby JIT implementations shown are 5.7, 2.3, and 1.9 times faster than their reference interpreter implementations. This remaining performance gap to C prevents scripting languages from becoming truly ubiquitous.

The goal of this paper is to improve the steady-state performance of scripting languages. We want to narrow the performance gap between statically-compiled and scripting languages, so that programmers can leverage the latter's ease-of-use and employ them in even more domains. Among the different scripting language implementations, we choose to analyze and modify a JavaScript compiler, namely JavaScriptCore [12] — one of the currently highest-performing JIT compilers, as shown in Figure 1. JavaScriptCore is the compiler in Apple's Safari [16].

JavaScript performance has been the focus of vast resource investments from companies, resulting in many highly-tuned JavaScript virtual machines. Conceptually, however, JavaScript is similar to most scripting languages, and the compilation strategies and optimizations used by JavaScript compilers can be traced back to Smalltalk [17] and Self [18]. Hence, the changes proposed in this paper can be applied to and benefit nearly all scripting languages.

A major reason for the lagging speed of scripting languages is their dynamic nature [19], [20], [21], [22], [23]. Since programmers do not declare types, the compiler has to discover the type and structure of objects at runtime. This is why advanced scripting language compilers are JITs.

Advanced JIT compilers have several traits. First, they

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00054

perform extensive profiling to detect the common case or "hot path" behavior of the program. This is necessary, so that they can perform code optimizations that improve the performance of the common case. Another trait is that they are organized in multiple *tiers*, where each tier offers a different tradeoff between the time to generate code and the quality of code generated. A final trait is that the code generated by the more advanced tiers includes checks against corner cases. This is because the specialized code generated by these tiers can only handle the common cases. When the code detects a corner case or "cold path", it jumps to less optimized tiers.

Figure 2 shows the tiers in JavaScriptCore. When the code starts to run, it is processed by the Interpreter. In the background, the *Baseline* compiler generates code containing profile information. This Baseline code is later invoked. As it executes and popular functions are discovered, they are re-compiled by the more advanced *DFG* compiler. Finally, for the really hot functions, the most advanced compiler (*FTL*) creates very specialized, high-performance code.

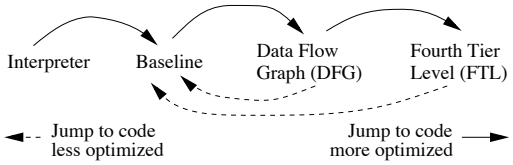


Figure 2: Tiers in JavaScriptCore, the compiler in WebKit. WebKit is the open-source version of Apple's Safari.

When a corner case causes a check in a specialized version of the code to fail, execution transfers to a lower-tier version (left-pointing arrows in Figure 2). The code there can handle the corner case. This means that, at the point of the transfer, both the higher and lower tier versions of the code must have *consistent* states. Otherwise, the transfer cannot take place. Unfortunately, this means that some code motion compiler optimizations are restricted in the higher tier's compilation; otherwise, on a jump to the lower tier, the code would find an inconsistent state. These points of potential cross-tier transfer are called *Stack Map Points (SMPs)*.

In this paper, we examine the code generated by the JavaScriptCore compiler for popular applications. We find that, surprisingly, the code generated by its most advanced tier (FTL) has a high frequency of SMPs. This is because the compiler introduces many checks in the code and, for each check, there needs to be an SMP that will be invoked if the check fails. These checks probe a variety of conditions, such as array bounds, arithmetic overflow, or correct type and property. Our experiments show that, on average, there is one check for about every 12 dynamic assembly instructions executed.

This effect contributes to the lackluster performance of JavaScript and scripting languages in general. Clearly, the checks add additional overhead. More importantly, the compiler struggles to perform code optimizations across SMP-guarding checks. The reason is that, if a check fails, execution jumps to code that expects a certain state. Ironically, in practice, for the routines compiled by the most advanced tier, the checks fail very rarely. Still, the checks have to remain.

To minimize this performance bottleneck, we extend the FTL tier of JavaScriptCore. Our compiler emits code that places *hardware transactions* around performance-critical code sections with SMPs, and converts the SMPs within these transactions into transactional aborts. With the SMPs removed, the compiler is then able to more effectively perform code optimizations in these performance-critical sections. In addition, we also propose a *Sticky Overflow Flag (SOF)* to eliminate overflow checks. Overall, if a remaining check fails, the transaction rolls back and execution resumes at a lower-tier version. The code target is a lightweight hardware transactional memory (HTM) like IBM's Rollback-Only Transaction (ROT) mode [24]. While the use of HTM to enable unsafe code optimizations is not new, this is the first time that an industrial-strength JavaScript compiler is extended to generate transactions to eliminate the deleterious effects of SMPs.

We call our changes *NoMap*. We use NoMap on the SunSpider and Kraken application suites, and run them natively on a machine that emulates emerging lightweight HTM architectures. NoMap reduces the instruction count of the applications by an average of 14.2% and 11.5%, and reduces the execution time by an average of 16.7% and 8.9%, for SunSpider and Kraken, respectively. These are substantial speed-ups for a very highly-tuned compiler. Moreover, they are obtained completely automatically. We also evaluate NoMap on Intel's heavyweight HTM hardware.

Overall, this paper's contribution is three-fold. First, we uncover and characterize a heretofore largely unnoticed performance bottleneck in JavaScript (and other scripting language) implementations: frequent inter-tier jump points that limit optimization by requiring a consistent state across tiers. Second, we propose changes to mitigate the effects of SMPs and extend a JavaScript JIT compiler to support our changes. Third, we evaluate the improvement of NoMap using both lightweight and heavyweight TM hardware.

II. Background

A. Creating Scripting Language Machine Code

JavaScript and most scripting languages are dynamically typed, which means one can add and remove properties (i.e., fields) from an object at runtime. Also, a given property can hold different types of data, such as numbers, strings, etc. In addition, JavaScript tries to eliminate undefined behaviors by handling all corner cases, rather than causing an exception and crashing the program, forcing operations to have complex semantics. As a result, it is tricky to generate high-performance code for these languages.

To maximize performance, advanced compilers for scripting languages are multi-tiered JITs. As an example, Figure 2

shows the structure of JavaScriptCore. Each compiler tier offers a different tradeoff between the time to generate machine code and the quality of code generated. The low-tier compilers create code quickly and handle all corner cases. However, the code is not efficient. The higher-tier compilers produce high-quality code. However, the code is specialized for one case and takes longer to generate.

Table I shows the speedup of each JavaScriptCore compiler tier over its Interpreter tier once steady-state execution is reached. For each configuration (*Baseline*, *DFG*, and *FTL*), we limit the maximum compiler tier it can use to the configuration name. For instance, in the *DFG* configuration, only a combination of the Interpreter, Baseline compiler, and DFG compilers are allowed to be used. We run the SunSpider [25] and Kraken [26] benchmark suites and show two averages, *AvgS* and *AvgT*. *AvgT* is the average of all the benchmarks in the suite, while *AvgS* is the average of a subset described in Section VI-C.

Highest Tier Enabled	SunSpider		Kraken	
	AvgS	AvgT	AvgS	AvgT
Baseline	2.13x	1.88x	1.22x	0.87x
DFG	7.71x	6.64x	8.45x	6.67x
FTL	11.48x	9.37x	15.03x	10.94x

Table I: Speedup of JavaScriptCore tiers over interpreter.

Overall, each tier typically improves on the performance of the tier below by spending more time performing compilation. For instance, the FTL tier is 9.37x and 10.94x faster than the Interpreter, and 41% and 64% faster than DFG for the SunSpider and Kraken suites, respectively. FTL is able to attain such performance improvements over DFG by using the LLVM [27] infrastructure. As the LLVM compiler is used heavily in industry, many advanced optimizations and analysis passes exist in its infrastructure – many more than in the DFG compiler. FTL runs the optimizations included normally in the `-O2` configuration of LLVM, which contains more optimizations passes than DFG and more advanced implementations of the passes in DFG. FTL also uses LLVM's backend instruction selector, which provides significant speedups.

In JIT compilers, the algorithm used to transfer from one version of code to another within the execution of a function is called On-Stack-Replacement (OSR) [28]. The points in a function where execution can jump in from another tier are Entry Points. The points where execution can jump out to another tier are Exit Points; if the exit is to a lower tier due to a failed check, the point is called a Deoptimization Point.

B. Transfers Between Compiler Tiers

At Entry and Exit points, the compiler has to guarantee that the values of the program variables are the same in the source and destination code versions. This limits the types of code optimizations that the compiler can perform in one tier without adjusting the other. In addition, since some of these variables may be in different register or stack locations in the two tiers, the compiler has to keep a map of the location of the variables and intermediates in both tiers.

At Entry and Exit points, the compiler places machine code to execute the OSR algorithm. When a cross-tier transfer is necessary, the OSR algorithm is invoked. It uses the maps of the two tiers to move variables around until they match the locations expected in the destination code version.

In FTL, these Entry and Exit points are called *Stack Map Points (SMPs)*. With each SMP, FTL associates a *Stack Map Entry*, which is a structure that describes what variables are in what registers and in the stack.

III. The Problem with Tier Transfers

An analysis of SMPs in FTL code shows that they hurt performance significantly. We now describe the problem.

A. Deoptimization Stack Map Points

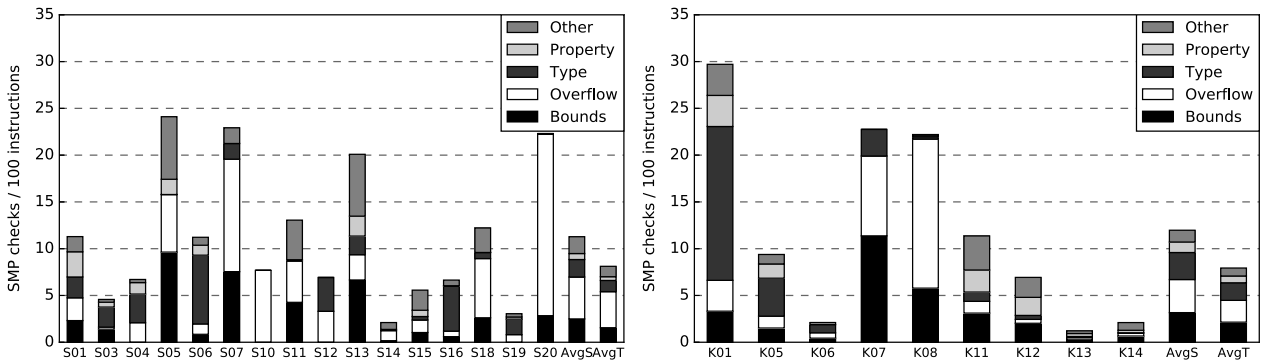
1) Frequency of Deoptimization SMPs: Through profiling, we discover that the code generated by the high-performance LLVM-based FTL compiler is full of checks which, on failure, cause execution to reach deoptimization SMPs. Specifically, we profile the number of SMP-guarding checks in code generated by the FTL compiler. In the SunSpider and Kraken JavaScript suites, we observe, on average, one such check per 12 dynamic assembly instructions executed. These checks guard SMPs. The SMPs are at the same points in the FTL code as in the versions generated by the Baseline compiler tier, since these are points of cross-tier execution transfer.

Figures 3(a) and (b) show the number of SMP-guarding checks in the FTL tier per 100 dynamic x86-64 assembly instructions executed by the SunSpider and Kraken suites, respectively. For each suite, we show bars for each of the benchmarks discussed in Section VI-C, a bar for the average of such benchmarks (*AvgS*), and then a bar for the average of all the benchmarks (*AvgT*). More details are provided in Section VI-C.

The bars are broken into bounds checks, overflow checks, type checks, property checks, and other checks. These checks test assumptions made by the FTL compiler. At array accesses, they check that the accesses are within bounds (*Bounds* checks); at integer arithmetic operations, they check for overflow (*Overflow* checks); and at data uses, they check that an object's type and the offset of an object's property are as expected (*Type* and *Property* checks, respectively).

Other checks include those guarding against accessing an undefined location in an array (i.e., a "hole"), trying to index an array with the wrong kind of value, and executing an unexpected code path. In the *Other* category, the most common check is to guard against accessing an undefined location within an array.

From the figures, we see that checks are very frequent in the optimized FTL code. From the *AvgT* bars, we see that there are 8.1 and 8.5 checks per 100 instructions in SunSpider and Kraken, respectively. This means that there is one check every 12 instructions executed. If we focus on



(a) (b)

Figure 3: Number of SMP-guarding checks in FTL per 100 dynamic assembly instructions in Sunspider (a) and Kraken (b).

the *AvgS* bars, the averages go slightly higher, to 11.3 and 12.0 checks in SunSpider and Kraken, respectively.

Of the checks, overflow checks are the most frequent, accounting for 47% and 29% in SunSpider and Kraken, respectively, for *AvgT*. Bounds checks are also numerous, and account for 19% and 27% in the two suites, respectively, for *AvgT*. We find that over 95% of bounds checks occur within loops.

These numbers are significant, as JavaScriptCore already includes passes for the explicit purpose of removing and simplifying bounds, overflow, and type checks. Specifically, the *IntegerCheckCombiningPhase* pass aims to hoist/remove unnecessary bounds and overflow checks. The *TypeCheckHoistingPhase* pass tries to hoist/remove redundant type checks. Unfortunately, their effectiveness is limited by the presence of SMPs, and SMPs' deleterious effect on conservative optimizations. The checks shown in Figures 3(a) and (b) are the checks which remain after all original JavaScript optimizations are performed.

2) Frequency of Invoking Deoptimization SMPs: While there are many check-guarded deoptimization SMPs in FTL code, failures very rarely occur. To determine the frequency of invocations to deoptimization SMPs, we run the SunSpider and Kraken benchmarks 1,000 times each in a loop, and count the number of failures in FTL-compiled functions. Of a total of about 85 million FTL functions called, there were less than 50 deoptimizations. In addition, after around 50 iterations, checks practically never fail. Still, the checks and deoptimization SMPs have to remain to guarantee correctness.

3) Performance Impact of Deoptimization SMPs: To ensure correctness, at these deoptimization SMPs, the compiler guarantees that the values of program variables are the same as in the corresponding entry points in the Baseline code version. As a result, FTL is limited in the code optimizations it can perform across these deoptimization SMPs, since they might change the values of the program variables at SMPs and thus break the agreement between tiers.

For example, FTL cannot move memory accesses across an SMP. Otherwise, if execution transferred to the Baseline tier, the program would perform the same access twice, or fail to perform the access at all. Even moving non-memory operations across an SMP is not without drawbacks. If post-SMP operations are moved before the SMP, the code must also keep the values existing before the operations, in case execution transfers to the Baseline tier. This causes increased register pressure. If pre-SMP operations are moved after the SMP, FTL needs to generate bailout code when execution transfers to the Baseline tier.

B. Example of Performance Impact

To see an example of SMPs hurting performance, consider a loop where each iteration accesses a different element of an array, and accumulates it into a field. Figure 4(a) shows the initial JavaScript code. In this code, object `obj` has two properties: an array of numbers (`values`) and a number value (`sum`). In this loop, the sum of `obj.values` is stored to `obj.sum`. Figure 4(b) shows the code generated by the Baseline compiler for this JavaScript loop. Note that in this and subsequent parts of this figure, we do not include `obj`'s initialization. In addition, for simplicity, we do not expand the for-loop control flow, nor how the induction variable is optimized. Also, while lines 1,2,4,5,6,7,8 in Figure 4(b) require SMP Entry Points, we do not show them. In the code, to ensure that all of JavaScript's corner cases are covered, most of the operations are executed by runtime calls, as shown in the figure. These calls ensure that the operations can be correctly executed for all object types.

Figure 4(c) shows the FTL tier code before optimization. To be able to eliminate the runtime calls for the various operations, many checks have been inserted. The figure shows these checks, with their SMPs, using the format $SMP_{linenum}$, where linenum is the line in the Baseline tier of Figure 4(b) where the execution should be transferred to, if the check fails. Even with all of these checks, this code is significantly faster than the Baseline version in Figure 4(b).

The FTL compiler tries to eliminate as many checks as possible, and make all operations as efficient as possible. Figure 4(d) shows the ideal code that we want the FTL tier

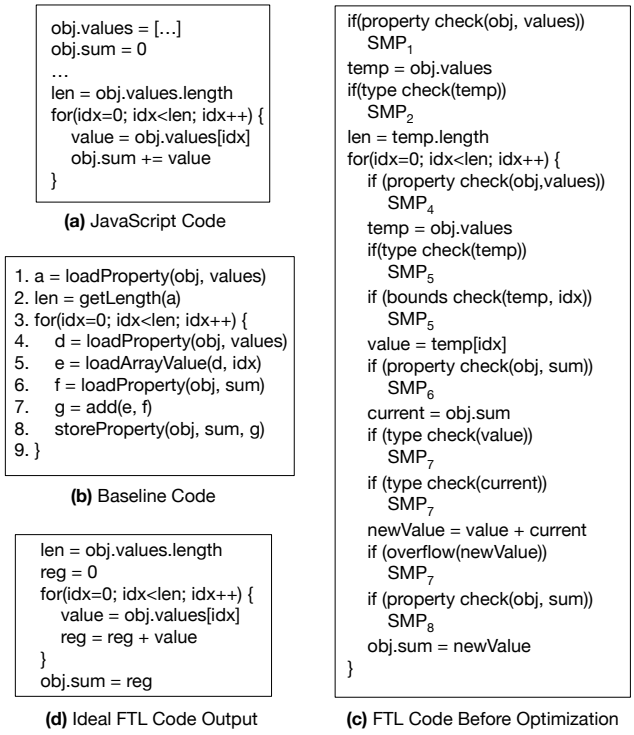


Figure 4: JavaScript code compilation example.

to generate. In this figure, all checks have been eliminated and, also, instead of storing to `obj.sum` at every loop iteration, a register is used for accumulating the sum, and only the final result is stored in `obj.sum`.

Unfortunately, generating this code is not possible. First, it is not possible to remove all checks. For instance, one cannot remove the overflow check, as the sum into `obj.sum` in Figure 4(d) may overflow. Also, while possible in this example, hoisting array bounds checks sometimes requires complex analysis to determine the maximum index retrieved. Second, once checks and their corresponding SMPs remain, it is hard for the compiler to generate efficient code. The presence of SMPs inside the loop invalidates the optimization of sinking `obj.sum` to outside of the loop in Figure 4(d). This is because it is now possible for the execution to transfer to the Baseline tier from within the loop. As shown in lines 6-8 of Figure 4(b), the Baseline's code uses `obj.sum` as the accumulator, so it must hold the correct accumulated value if a SMP is reached. Hence, `obj.sum` in Figure 4(d) must be stored in every loop iteration. While it is possible to speculatively sink the store if bailout code is also inserted before each SMP point in the code of Figure 4(c), speculative optimizations are notoriously complicated to implement correctly in practice.

Overall, our analysis shows that the FTL tier of the state-of-the-art JavaScriptCore compiler is crippled by two effects: the optimization-limiting presence of the SMPs, and the overhead of the SMP-guarding checks. The result is disappointing performance. A similar situation also occurs in both other JavaScript and scripting language compilers (Section VIII).

IV. Speeding-up JavaScript

A. Main Idea

We propose placing transaction annotations around performance critical sections with SMPs. SMPs, which are only invoked if their corresponding checks fail, are then replaced by transaction abort instructions. If an abort is triggered, the transaction is rolled back and execution transfers to non-transactional code generated by a lower compiler tier.

With the SMPs removed, conventional optimizations are now effective in these regions, on both memory and non-memory operations. In some transactions, it is also now possible to perform special optimizations on the formerly SMP-guarding checks. We introduce two very simple passes which are effective at combining and/or removing some checks.

The use of HTM to enable unsafe code optimizations similar to the ones we apply here is not new (e.g., [29], [30], [31]). However, this is the first time that HTM is used to solve the bottleneck of cross-tier transfers in a multi-tier scripting language compiler.

The IBM POWER8 features two modes of HTM execution: a lightweight mode to leverage HTM's rollback capabilities called Rollback-Only Transaction (ROT) mode [24], and a heavyweight mode to prevent data races. We target the first one. In ROT mode, only the write footprint of transactions is buffered, not the read one. Moreover, transactions can commit without waiting for the write buffer to drain the stores inside the transaction. Since JavaScript is single-threaded, this lightweight HTM mode can be safely used.

There are two reasons why using transactions to enhance conventional optimizations suits codes generated by advanced tiers such as FTL. First, by the time a function reaches the FTL tier, checks rarely fail. The function has been executed enough times for highly-accurate profiling data to be collected, and thus the checks match the characteristics of the code. Second, the codes that reach FTL are codes where attaining high-performance is most crucial. They have been deemed "hot" enough that it is worth spending the extra time to recompile them with more aggressive optimizations.

We call our proposal *NoMap*. NoMap has two parts: replacing SMPs by aborts to enhance conventional optimizations, and adding passes to combine and remove the formerly SMP-guarding checks.

B. Replacing SMPs by Abort Instructions

Our algorithm places transactions around "hot" loops with SMPs in FTL-generated code. Inside a transaction, each SMP is replaced by an abort instruction. It is safe to remove these SMPs because they are not entry points; the only entry points allowed in FTL-generated code are the beginning of functions. Then, our algorithm creates an SMP to coincide with the begin-transaction instruction. It also creates an SMP before the corresponding code section in the Baseline version.

These two SMPs are needed when the transaction fails and execution transfers to the Baseline version.

Figure 5 shows an example of the changes our algorithm makes on a simple loop and how execution proceeds in the rare case of a check failure. Figure 5(a) shows the original code, while 5(b) shows the transformed code. In each case, we show two code versions: the FTL one (left) and the Baseline one (right).

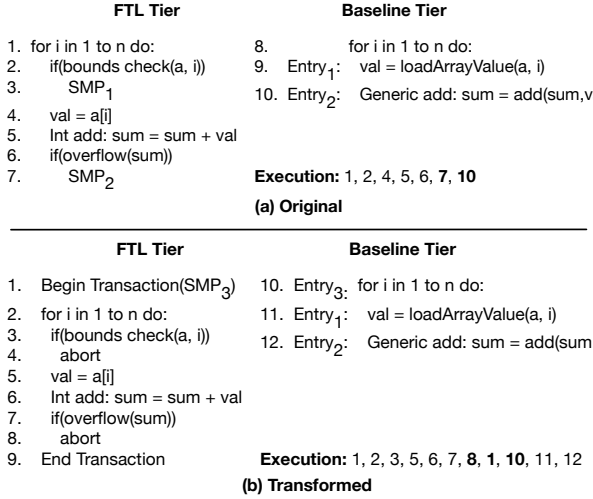


Figure 5: Replacing SMPs by transaction aborts.

In the FTL code of Figure 5(a), there is an array access (a[i]) preceded by a bounds check and SMP, and then an integer addition followed by an overflow check and SMP. In the Baseline code of that figure (right side), the array entry is accessed with a *loadArrayValue* call. This is a runtime call that checks if the access is within bounds and, if not, returns the undefined value; it never crashes. Also, the Baseline code uses a runtime call for the add. This is because JavaScript semantics define addition for arbitrary objects. In addition, JavaScript numbers use, by default, the double-precision floating-point format. The FTL code uses integer addition to reduce overhead, but must check for overflow.

Suppose that the overflow check fails, and the second SMP (SMP_2) is invoked. In this case, execution transfers to the Baseline version. It enters through $Entry_2$, and the addition is re-executed using floating-point arithmetic. The figure shows the sequence of executed code lines: 1,2,4,5,6,7,10.

Our algorithm generates the FTL code on the left side of Figure 5(b). The code is inside a transaction and the SMPs are now aborts. The Baseline version has a new entry point ($Entry_3$) in case the FTL transaction fails. If the overflow check in the FTL code fails, the transaction is rolled back, and execution enters the Baseline version through $Entry_3$. The Baseline version re-executes the whole loop. The figure shows the sequence of execution: 1,2,3,5,6,7,8,1,10,11,12.

$Entry_1$ and $Entry_2$ in the Baseline version need to be kept since they are entry points from the DFG tier (Figure 2) as well. Note that if an abort is triggered, our transformation results in more instructions being executed than the Baseline version. However, in heavily-profiled "hot" regions, checks virtually never fail.

We perform this transformation before LLVM runs its optimization passes. This allows all of the LLVM optimization passes to interact with the transformed code. Since there is no SMP within the transaction, optimization passes are able to much more effectively optimize this transformed code.

C. Combining and Removing Checks

When an SMP-guarding check inside a transaction fails, it does not matter when the failure is detected or when the rollback occurs — only that the transaction is eventually rolled back. Hence, in NoMap, we move the checks around within the transaction, if it reduces overhead. In particular, we combine some checks and remove others.

1) Combining Array-Bounds Checks: In JavaScript, a compiler cannot always tell the size of an array statically, since array size declarations are optional. Instead, arrays are automatically elongated at runtime as new elements are updated. Hence, if the index requested is not in the array, JavaScript returns the "undefined" value.

The result of these two JavaScript traits is that there is a bounds check before many array accesses. As we show in Section III-A1, on average about 23% of the SMP-guarding checks are bounds checks and over 95% of the checks are in loops.

Many array accesses occur in loops, often with progressively increasing or decreasing indices. In this case, it should be easy to identify the largest index used by the iterations and check it against the array bounds, making all the other checks unnecessary. This "combined" check could be hoisted before the loop or, if there is an early exit out of the loop, sunk after the loop.

Unfortunately, while JavaScriptCore does implement an advanced loop bounds check removal pass similar to many previous works [32], [33], [34], SMPs limit the effectiveness of their conservative analysis and many bounds checks still remain in the code. In our algorithm, however, after removing within-transaction SMPs, combining bounds checks is straightforward. Figure 6 extends the transactional FTL code of Figure 5(b)-left with bounds-check combining. Figure 6(a) shows the original code, where each iteration performs a bounds check. Figure 6(b) shows the transformed code. Since the loop has a monotonically increasing array index, all the checks are combined into a single check for the last index used, and the check is sunk after the loop. The single bounds check handles early exits of the loop as well.

To implement our algorithm, we build upon LLVM's `Scalar Evolution` analysis to identify monotonic loop variables. Once candidates are identified, we insert a new bounds check outside of the loop and delete the original checks from within the loop. Variables which are monotonically increasing have their check sunk to below the loop whereas monotonically decreasing variables have their check hoisted. Note that, without transactions, it would not be

```
1.  Begin Transaction(SMP3)
2.  for i in 1 to n do:
3.    if(bounds check(a, i))
4.      abort
5.    val = a[i]
6.    Int add: sum = sum + val
7.    if(overflow(sum))
8.      abort
9.  End Transaction
```

(a) Before Combining

```
1.  Begin Transaction(SMP3)
2.  for i in 1 to n do:
3.    val = a[i]
4.    Int add: sum = sum + val
5.    if(overflow(sum))
6.      abort
7.  if(bounds check(a, last_i))
8.    abort
9.  End Transaction
```

(b) After Combining

Figure 6: Combining bounds checks.

possible to safely sink checks to outside of loops while maintaining JavaScript's semantics.

2) Removing Overflow Checks: JavaScript semantics state that numbers must be represented in double-precision floating-point format by default. Since, in practice, many numbers are integers, JavaScript compilers try to convert as many floating-point operations as possible into integer operations. However, these integer operations must be checked for overflow, and reexecuted using double-precision values if overflow occur.

As a result, there is an overflow check after many arithmetic operations. As shown in Section III-A1, on average about 38% of the SMP-guarding checks are overflow checks.

In x86-64, arithmetic operations set the overflow (OF) flag. The flag must be checked soon after the operation because it can be overwritten by later instructions. To improve performance, we propose a *Sticky OF flag (SOF)* that remains set from the time an overflow occurs until the end of the transaction. With that, we can *remove all the overflow checks inside the transaction*. We simply check the SOF flag at the end of the transaction, to see if any overflows occurred.

The IBM Power ISA has an SOF (called summary overflow flag [35]) that can be used. Specifically, we want arithmetic instructions to set both OF and SOF, as in the IBM Power. The SOF flag should only be checked at the end of the transaction.

Figure 7 extends the example in Figure 6 with overflow-check removal. Figure 7(a) repeats the code in Figure 6(b), where each iteration checks for overflow. In Figure 7 (b), we remove the check. The end-transaction instruction automatically checks the SOF flag and aborts if it is set.

V. Implementation Aspects

A. TM Support

Since JavaScript is single threaded, NoMap does not need to deal with cross-thread conflicts; it can target a lightweight HTM like IBM POWER8 ROT mode [24]. Specifically, the hardware only needs to record and buffer in caches a transaction's write footprint. Furthermore, it does not require a fence at a transaction commit; execution proceeds past a transactional commit (XEnd) instruction while the write buffer drains of transaction writes in the background.

The rest of the expected hardware support is similar to other HTM implementations [36], [37], [24], [38], [39]. In particular, each cache line has an Speculative Write (SW) bit to record if the line was written during the transaction. XEnd

```
1.  Begin Transaction(SMP3)
2.  for i in 1 to n do:
3.    val = a[i]
4.    Int add: sum = sum + val
5.    if(overflow(sum))
6.      abort
7.  if(bounds check(a, last_i))
8.    abort
9.  End Transaction
```

(a) Before Removing

```
1.  Begin Transaction(SMP3)
2.  for i in 1 to n do:
3.    val = a[i]
4.    Int add: sum = sum + val
5.  if(bounds check(a, last_i))
6.    abort
7.  End Transaction
```

(b) After Removing

Figure 7: Removing overflow checks.

clears the SW bits with a flash-clear hardware operation. NoMap expects flattened nested transactions: in a nest, all the transactions commit or get aborted. Irrevocable events such as I/O, exceptions, and signals also cause the transaction to be aborted.

Note that supporting ROT HTM requires a subset of the hardware needed to support conventional multithreaded HTM. Hence, it will not negatively affect the performance of the conventional HTM.

B. Sticky Overflow Flag

As described in Section IV-C2, NoMap leverages a Sticky Overflow Flag (SOF) like the one in the IBM Power ISA [35], and uses it to enhance TM. NoMap expects all arithmetic instructions to set both the OF and SOF flags. The integer arithmetic operations within transactions now have no overflow checks.

The SOF is automatically cleared at the outermost transactional begin (XBegin) instruction of a transaction nest — i.e., when execution enters transactional mode. The outermost XEnd instruction in a transaction nest automatically checks if the SOF is set. If it is, the transaction is aborted.

C. Creation of Transactions

The algorithm used by NoMap to create transactions around code with SMPs is as follows. By default, transactions are placed around loop nests. However, if NoMap estimates that the data footprint is too large, the transaction is placed around successively smaller code sections. Specifically, first, it is placed around the innermost loop in the nest. Then, the innermost loop is tiled so the state fits in cache, and the transaction is wrapped around the tile.

At run time, if a transaction aborts due to transactional state overflow, execution transfers to the Baseline version. NoMap then tries to change the code, also following the above algorithm, so that it does not overflow, and compiles it again. If the cache-overflowing transaction has a function call, then the transaction is removed; NoMap assumes that the overflow was caused by the callee's execution.

VI. Evaluation Setup

To evaluate NoMap, we use a real machine rather than a simulator, so that we can perform long application runs, measure real execution times, and use complex code generated by our compiler for a 64-bit machine. While the

lightweight HTM support envisioned is similar to IBM's ROT, JavaScriptCore does not support the Power ISA. In addition, while NoMap can use the Intel Transactional Synchronization Extensions (TSX) [36], TSX is a heavyweight HTM with many additional overheads, including speculative read buffering, slow transaction reads, and slow commit due to requiring write buffer draining at XEnd. Therefore, we evaluate NoMap natively using two approaches: first, *emulating* the HTM support expected by our design, and, second, using Intel's Restricted Transactional Memory (RTM) primitives.

The machine we use is an Intel Skylake i7-6700 CPU executing Ubuntu 14.04.4 LTS. We use a single core with a single context. It has 32KB 8-way L1 data and instruction caches, and a 256KB 8-way L2 cache.

A. Lightweight HTM Platform

Properly emulating HTM in a platform without the requisite hardware requires accounting for the TM overheads. These overheads are due to executing the XBegin and XEnd instructions, keeping transactional write state in the cache, and handling aborts. To model the execution of XBegin and XEnd, we add extra instructions to the execution (Section VI-A1). To understand the interaction of transactional data with caches, we run under Pin [40], modeling the caches of the native machine (Section VI-A2). In Section VI-A3 we validate that our emulated platform accurately models lightweight HTM.

1) Executing Transaction Begin and End: We model the lightweight HTM XBegin and XEnd instructions using LLVM intrinsics which prevent instruction reordering. As a result, during code transformations, the compiler cannot move code across transactions. At the end, when all the assembly instructions have been emitted, we replace these intrinsics with x86-64 instructions as follows.

The intrinsic representing XBegins is replaced by an *mfence* instruction. The fence models the main overhead of XBegin: the forced ordering of memory operations before and after the transaction begin. The extra work that the hardware would need to do in a real transaction, such as moving the PC destination for failures to a register, or clearing the SOF, would not be on the execution's critical path.

The XEnd instruction does not need to stall for the write buffer to drain. Before execution can continue, the XEnd instruction must flash-clear the SW bits in the L1 and L2 caches. As indicated by previous work [41], this can be done in a few cycles with a simple circuit in the tag array of the caches. To model this overhead, we add 5 cycles to the overall execution time for each completed transaction.

2) Caching Transactional State: Our native runs do not provide us visibility into the cache behavior. Hence, we perform a second set of runs using Pin, modeling the cache hierarchy of the real machine. Since the machine has a 256KB 8-way L2 cache, the goal of our Pin runs is to determine whether: (i) the write footprint of our applications' transactions fits in such a cache, and (ii) keeping the write footprint in the cache hurts performance by introducing more cache misses for reads.

An analysis of the Pin data shows that the write footprint fits amply, and does not hurt performance by causing read misses. We show data in Section VII-C.

We do not need to model slowdowns for transactional reads or writes. First, reads have no additional overhead because they are not part of the transactional state. Second, since writes are traditionally not on the critical path, any slight increase in latency does not affect performance. This is shown empirically by Ritson and Barnes [42]. Their experiments with Intel's RTM [36] hardware show no slowdowns due to transactional writes.

3) Validating the Platform Overhead: We perform a limited validation of our emulated lightweight HTM platform. We execute some small programs with transactions on an IBM POWER8 multiprocessor with support for ROT mode and with the additional instructions needed to properly leverage the SOF flag. We conclude that our emulated platform does not underestimate the overhead of lightweight HTM in a real machine such as the POWER8 multiprocessor. The experiments are described in the appendix.

B. Heavyweight HTM Platform

We also test NoMap on a machine with Intel's RTM [36] hardware. Our goal is to see the speedups that can be achieved on heavyweight HTM. However, using RTM has drawbacks. First, in order for a transaction to successfully complete, all transactional writes must fit within the 32KB L1D cache, and all transactional reads must fit within the 256KB L2 cache. Second, XEnd stalls until all writes within the transaction have drained from the write buffer; according to [42], an XEnd takes at least 13 cycles to commit. Third, experiments have shown that reads within an RTM transaction take about 20% longer to execute than other reads [42]. Finally, x86-64 does not have SOF support and, hence, we disable NoMap's pass to eliminate overflow checks.

Having NoMap target the RTM primitives is straightforward. Moreover, when a transaction fails, an abort code provided to the runtime indicates the reason for the failure. Based on this code, our runtime makes a decision to either retry the transaction, adjust the transaction size to avoid cache overflow, or jump to a lower-tier version of the code.

C. Architectures and Applications Studied

Table II shows the architectures evaluated. They include a conventional machine (*Base*) and five configurations using HTM: a simple design (*NoMap_S*); one that additionally provides bounds-check optimization (*NoMap_B*); one that additionally provides both bounds-check and overflow-check optimization (*NoMap*, which is our proposed design); an unrealistic best case design where all checks within transactions are removed (*NoMap_BC*); and a design that uses Intel's heavyweight HTM (*NoMap_RTM*).

In all configurations, FTL uses all LLVM optimization passes enabled by the `-O2` flag, as well as additional passes

Architecture	Explanation
Base	Unmodified JavaScriptCore. No Transactions.
NoMap_S	Simple design of NoMap: * Transactions are inserted. * SMPs are replaced with aborts. * Code optimizations are performed across aborts.
NoMap_B	Intermediate design of NoMap: * *NoMap_S* + hoisting/sinking bounds checks.
NoMap	Complete NoMap: * *NoMap_B* + using SOF for no overflow checks. * This is our proposed design.
NoMap_BC	Unrealistic base case for NoMap: * All checks within transactions are removed.
NoMap_RTM	NoMap on RTM: * *NoMap_B* running on Intel RTM hardware.

Table II: Architectures evaluated.

inserted by JavaScriptCore custom-tailored for JavaScript. Our *NoMap* configurations add additional LLVM passes, which are run before the LLVM optimization passes execute, but after all JavaScriptCore-specific optimizations are performed.

We evaluate the steady-state performance of the SunSpider [25] and Kraken [26] suites. These suites have 26 and 14 benchmarks, respectively, which model various computations. In the plots, we list the benchmarks as numbers corresponding to the alphabetical order. Hence, SunSpider benchmarks *3d-cube* ... *string-validate-input* correspond to S01 ... S26, and Kraken benchmarks *ai-star* ... *stanford-crypto-sha256-iterative* correspond to K01 ... K14. As shown in Table III, in our evaluation, we report two averages for each suite. One is the average of all the benchmarks in the suite (*AvgT*). The other is the average of the benchmarks in the suite (*AvgS*) that we consider more informative of our optimizations. Specifically, AvgS does not include 3 benchmarks from SunSpider that our compiler practically optimizes away as dead code when the SMPs are converted to aborts. Also, AvgS does not include 7 benchmarks from SunSpider and 5 from Kraken where $\geq$95% of the instructions executed are not generated by FTL, but are from lower tiers or are C runtime code (Table III). When we show per-application bars, we only show the benchmarks featured in AvgS. However, *we always report both averages*.

Suite	Bench. included in **AvgT**	Benchmark # included in **AvgS**	Benchmark # not included in **AvgS**. Why?
SunSpider	All 26	1,3,4,5,6,7,10 11,12,13,14,15 16,18,19,20	2,8,9: Dead code 17,21,22,23,24,25,26: $\geq$95% non-FTL code
Kraken	All 14	1,5,6,7,8,11 12,13,14	2,3,4,9,10: $\geq$95% non-FTL code

Table III: Suites and benchmarks evaluated.

VII. Evaluation

A. Impact of NoMap on Instruction Count

Figures 8 and 9 show the total number of dynamic instructions executed by each benchmark of the SunSpider and Kraken suite, respectively, for the environments of Table II. For each benchmark, *AvgS*, and *AvgT*, we show six bars. From left to right, they are: the unmodified JavaScriptCore (*Base*); simple NoMap (*NoMap_S*); NoMap_S augmented with bounds check optimization (*NoMap_B*); NoMap_B augmented with overflow check optimization (our final *NoMap*); an unrealistic best case for NoMap (*NoMap_BC)*; and NoMap_B running on heavyweight HTM (*NoMap_RTM*). In each benchmark, the bars are normalized to *Base*.

Each bar is broken down into four categories: instructions not generated by FTL (*NoFTL*); FTL instructions outside transactions (*NoTM*); and FTL instructions inside transactions that either cannot be optimized (*TMUnopt*) or that can be optimized (*TMOpt*). *TMUnopt* are instructions in a function that is called from within a transaction. They cannot take advantage of being inside a transaction because the compiler compiled them without being aware that this code would eventually be called from a transaction. Hence, SMPs in these regions are not converted into aborts and optimized. Overall, the only code that can potentially take advantage of our framework is *TMOpt*.

Consider SunSpider first (Figure 8) and focus on *AvgS*. The simple *NoMap_S* eliminates 6.3% of the original instructions. The more advanced *NoMap_B*, which additionally hoists/sinks bounds checks, is able to eliminate 8.6% of the original instructions. The final *NoMap* design, which additionally removes overflow checks, eliminates 14.2% of the original instructions. This is a sizable instruction count reduction, as it is done in a *fully automated manner*. The best case of removing all checks within transactions (*NoMap_BC*) removes 17.1% of the original instructions.

NoMap_RTM only removes 5.1% of the original instructions, which is slightly less than NoMap_S (6.3%). This is because fewer instructions execute within transactions under heavyweight HTM than under the lightweight HTM architecture (Figures 8 and 9). The reason is the smaller memory footprint size supported by Intel. Indeed, as mentioned in Section VI-B, for a transaction to fit, all transactional stores must fit within the 32KB L1D cache and all transactional loads must fit within the 256KB L2 cache. In the lightweight HTM, all transactional stores must fit within the 256KB L2 cache, and there is no limit on the transactional load footprint.

AvgT shows similar trends. *NoMap* and *NoMap_RTM* reduce the instruction count by 19.7% and 14.2%.

Looking at individual benchmarks, we see that some benefit more than others. Some perform poorly, either because they have mostly *NoFTL* instructions (e.g., S16), or most FTL instructions are *NoTM* or *TMUnopt* (e.g., S03, S04, and S12). When there is a sizable *TMOpt* fraction, *NoMap* often eliminates many instructions (e.g., S10, S13, S18, and S20).

S18's improvement comes from allowing code movement around SMPs. Within function *cordicsincos*, the compiler now identifies a redundant load, and is also able to sink an additional load. Aggressive unrolling also occurs. S13 is improved by sinking bounds checks. Within three functions,

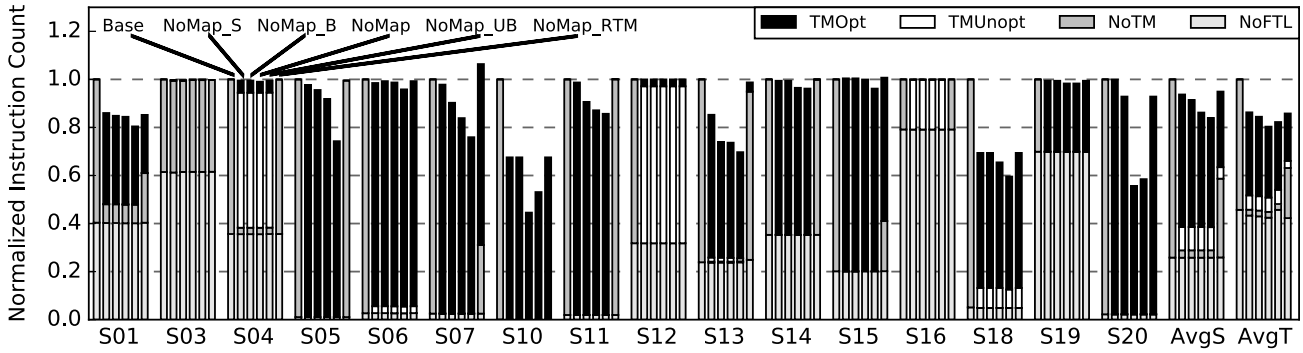


Figure 8: Number of instructions executed by SunSpider for different environments.

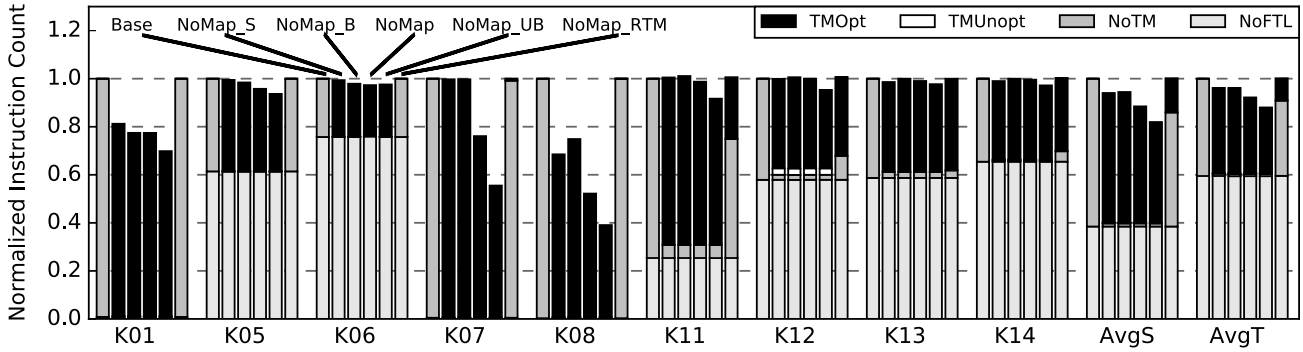


Figure 9: Number of instructions executed by Kraken for different environments.

72 bounds checks from 29 loops are sunk to outside of their original loops. S10 benefits from the SOF to remove overflow checks in a tight loop with simple arithmetic operations.

In Kraken (Figure 9) we see a similar picture, although the gains are slightly smaller than in SunSpider. Consider *AvgS* first. *NoMap* removes 11.5% of the original instructions, while the best case environment (*NoMap_BC*) eliminates 18.0%. *NoMap_RTM* does not reduce the instruction count. The reason is the lack of transactions with a footprint small enough to fit in the caches. The result is a very small *TMOpt* category. The *AvgT* bars show similar, if smaller, gains: *NoMap* removes 7.8% of the instructions.

B. Impact of NoMap on Execution Time

Figures 10 and 11 show the execution time of each benchmark in the SunSpider and Kraken suites for the usual environments. Like the previous figures, each benchmark is normalized to *Base* and has six configurations. Each bar is divided into two categories: time spent in transactions (TMTime) and remaining execution time (NonTMTime).

In SunSpider, we see that, for *AvgS*, *NoMap* reduces the execution time of *Base* by 16.7%. As this is accomplished without any programming changes to the applications, we conclude that NoMap is an effective optimization for JavaScript.

NoMap_RTM reduces the execution time by 6.5%. This is similar to its instruction count reduction. Based on the limitations of the heavyweight HTM discussed in Section VI-B, it is unsurprising that NoMap performs worse targeting heavyweight HTM than lightweight HTM.

The execution time reductions in *AvgT* are similar, if a bit higher: 21.7% for NoMap and 15.0% for NoMap_RTM. Generally, the execution time reduction is correlated with the instruction count reduction.

In Kraken, the impact of the optimizations is smaller. *NoMap* reduces the execution time by 8.9% in *AvgS* and 5.9% in *AvgT*. However, *NoMap_RTM* has no impact due to the shortcomings explained previously. Note that, for some benchmarks such as K05 and K06, *TMTime* is significantly higher than *TMOpt* in Figure 9. This indicates that much of the transaction time is spent executing unoptimized code (*NoFTL* in Figure 9), which is called from within a transaction. This limitation is described further in Section VIII. Overall, *NoMap* with a lightweight HTM is also effective in Kraken; even small gains are important in highly-optimized implementations such as JavaScriptCore.

C. Transaction Characterization

Finally, we characterize the transactions that *NoMap* inserts in the SunSpider and Kraken suites. Table IV shows this data for *AvgS* and the maximum value of any benchmark.

Columns 2 and 3 show the write footprint of a transaction and the maximum associativity needed by any cache set to store transactional writes, respectively. On average, the write footprint of a transaction is 44.9KB and 47.4KB, which is

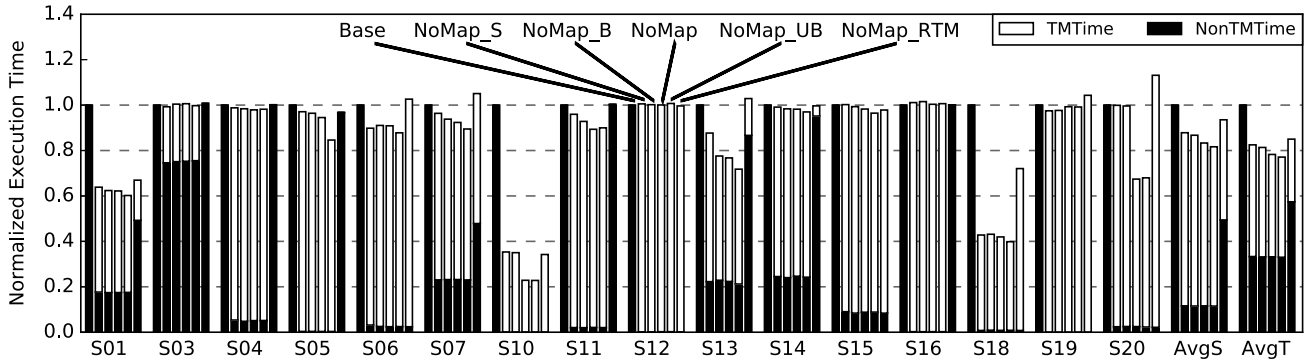


Figure 10: Execution time of SunSpider for different environments.

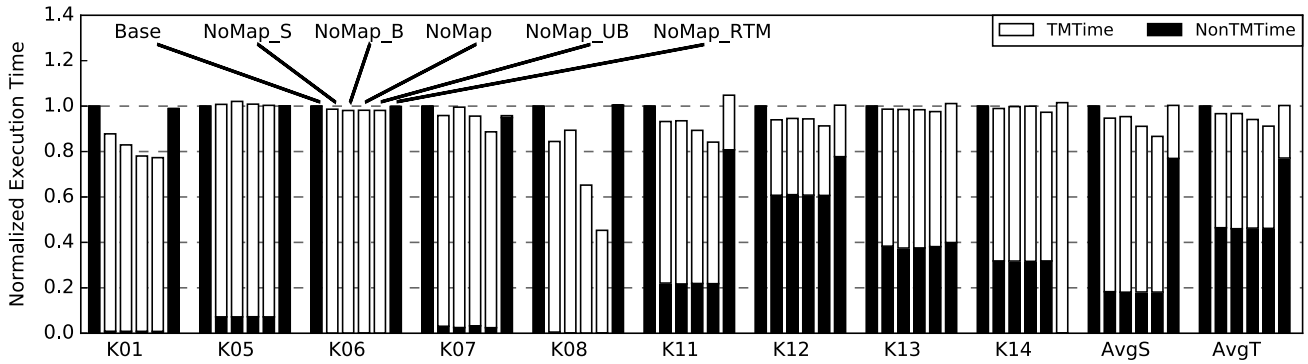


Figure 11: Execution time of Kraken for different environments.

Suite	Wr. Footp. (KB)	Wr. Asso.	# Static Trans. Full	Inn	Tile	# Static Trans. Elimin.
S (*AvgS*)	44.9	3.7	3.0	0.2	0.2	0.5
S (Max)	160.2	8	12	1	2	4
K (*AvgS*)	47.4	3.7	3.9	1.1	0.2	1.1
K (Max)	92.3	5	13	6	1	3

Table IV: Characterizing the transactions. *S* and *K* stand for SunSpider and Kraken.

about 18% of the size of the L2 cache. On average, 4-way associativity is needed. The worst case needs 160.2KB and 8-way associativity — which is still provided by the L2 cache.

The next 3 columns show the number of static transactions per benchmark that wrap a whole loop nest (*Full*), that wrap only the inner loop of a nest (*Inn*), and that wrap a tile of a loop (*Tile*). On average, a benchmark has about 3-5 static transactions. One benchmark reaches 20 transactions. The last column is the number of static transactions per benchmark that were eliminated because they were calling functions and overflowing the L2.

VIII. Discussion

FTL vs B3. Recently, JavaScriptCore has switched from using the LLVM-based FTL to B3, a new compiler designed by the JavaScriptCore Team, for its highest tier, due to LLVM's slow compilation times. B3 is focused on fast compilation times, and also adds some speculative optimizations. In NoMap, by using transactions, we maximize the performance potential of *conventional* compiler optimizations, in a way that is applicable to all scripting languages, with easy-to-maintain extensions. Note that LLVM supports many more conventional optimizations than B3, enabling it to better demonstrate NoMap's benefits.

Applicability of NoMap. The techniques described in this paper are directly applicable to other high performance compilers for JavaScript, and to compilers for other scripting languages such as PHP [43], [44], Python [13], [45], and Ruby [15]. All high-performance scripting language implementations have multiple tiers and use deoptimization SMPs to fall back to previous tiers. Deoptimization SMPs have different names in other implementations. For example, they are known as *Deoptimize Nodes* in V8 [46], *Snapshots* in SpiderMonkey [47], *Side Exits* in HHVM [44], and *Guards* in PyPy [13]. However, their functionality is similar in all implementations. While some implementations contain more speculative optimizations which can reduce the performance impact of SMPs, we believe that applying NoMap would result in performance improvements similar to JavaScriptCore in all implementations.

Implementing NoMap on POWER ISA. Since NoMap is applicable to many scripting languages, it is possible to

port our infrastructure to a language implementation that is supported on the POWER ISA. For this paper, we decided to focus on JavaScript because it is among the highest-performing scripting languages (see Figure 1). We wanted to show how NoMap can improve the performance of a state-of-the-art implementation developed by a company with vast resource investments. We decided to use JavaScriptCore for this project because it uses the LLVM compiler in the FTL tier. LLVM is stable, well-documented, and easy to modify. Our future work is to port NoMap to a language implementation that is supported on the POWER ISA.

Limitations of the Current Infrastructure. A limitation of our NoMap framework is that FTL does not perform interprocedural analysis. Compilation is performed at a method-level granularity. While functions are inlined, LLVM is unable to view functions beyond the compilation unit.

This function isolation hurts performance. As transaction information is not communicated between functions, even if a function is wrapped in a transaction via a caller function, the callee function is unaware of the transaction, and thus the compiler will not convert SMPs to aborts. Adding interprocedural analysis will likely result in additional performance improvements.

Effect on Website Performance. This paper has focused on improving the steady-state performance of scripting languages, focusing on a JavaScript compiler, due to its superior performance. Other authors have focused on JavaScript's speed when performing very short interactions on a web page [22], [48], [49], and have proposed the JSBench benchmark [50]. Such an environment is different than ours. JSBench benefits mostly from a low start-up time, as seen by the 7.0x speed-up of JavaScriptCore over vendors without interpreters [51]. NoMap, which targets steady-state execution, is not revelevant to JSBench. JSBench benchmarks are so short that the optimizing compiler tier is barely invoked.

JavaScript's steady-state performance is also important for websites that are single-page applications and have long running interactions. NoMap can help in these cases.

IX. Related Work

JavaScriptCore implements an advanced loop bounds check removal pass similar to many previous works [32], [33], [34]. However, we find that SMPs limit the effectiveness of these analyses. Many of the remaining SMP-guarding checks in JavaScriptCore are, in fact, array bounds checks. Unlike previous works, NoMap allows potentially illegal execution to proceed before rollback, enabling bounds checks with an undetermined upper bound to be moved outside of loops.

Prior work proposes using TM-like hardware to improve the execution of "hot" code paths and, more generally, to perform unsafe speculative compiler optimizations inside transactions. Some of these works create traces of hot code at the hardware level [52]. Other works use software-exposed HTM, and use rollback functionality if a "cold" path is entered [30], [53] or a speculative optimization fails [31].

HTM is becoming prevalent in real systems [54], [24], [36], [37], [29], [55], [38], [39]. The POWER8 ROT [24] supports a lightweight HTM. It is possible to implement NoMap on this system.

Other works use hardware to speed-up JavaScript. Mehara et al. [56], [57] make two proposals. In ParaGuard, they offload checks to another thread. In ParaScript, they parallelize loops and use TM to protect against data races. Both works expand single-threaded code to multiple threads. ParaScript requires full TM support, while we only need lightweight features. ParaGuard is based on a tracing compiler, while now all popular JavaScript compilers are method-based.

Anderson et al. [58] focus on the overhead of performing type checks in JavaScript. They propose a new instruction called Checked Load, which performs type checks in parallel with loads from memory. Checked Load is complementary to NoMap. NoMap improves performance by both enabling code-movement optimizations across all checks, and removing overflow and select bounds checks; Checked Load only accelerates type checks. Hence, they can be used together.

Su and Lipasti [59] recognize that Java's Exception Model hurts performance. They suggest allowing code motion around null and array bounds checks, while also removing many of these checks. Our work differs in three ways. First, our system allows move optimizations around all checks. Second, we describe the hardware needed to support NoMap, while they do not. Finally, we present a detailed evaluation of NoMap, while they show upper bounds on potential speedups.

X. Conclusion

To understand and improve scripting languages' lower performance, this paper made three contributions. First, it uncovered and characterized a heretofore largely unnoticed performance bottleneck in JavaScript (and other scripting language) implementations: optimization-limiting SMPs, and the overhead of SMP-guarding checks. Second, it proposed *NoMap*, a set of optimizations to mitigate the effects of SMPs, and extended a JavaScript JIT compiler to support these changes. Finally, it used NoMap to run the SunSpider and Kraken suites natively on a machine that emulated lightweight HTM. NoMap reduced the instruction count by an average of 14.2% and 11.5%, and the execution time by an average of 16.7% and 8.9%, for SunSpider and Kraken, respectively. These are substantial speed-ups on highly-tuned code. The paper also evaluated NoMap on heavyweight HTM but, due to its overheads, the gains were smaller.

Acknowledgments

This work was funded by NSF under grant NSF CCF 15-27223.

Appendix: Validation of the Emulation Platform

We perform a limited validation of our emulation of a lightweight HTM platform. Our goal is to ensure that our emulated platform does not underestimate the overhead of lightweight HTM in a real machine. We take an IBM

POWER8 multiprocessor with support for ROT mode and include the additional instructions needed to use the SOF flag. We prepare some small single-threaded programs with reads and writes surrounded by transactions. We measure the execution time of the programs with transactions on the IBM machine (*T_I*) and on our emulated platform (*T_E*). We then remove the XBegin, XEnd, and SOF-related instructions from the programs and time their execution on the IBM platform (*NT_I*) and on our emulated one (*NT_E*).

The two programs that we prepare perform either only reads (*All_Reads*) or only writes (*All_Writes*). We vary the number of such memory operations per transaction from 10 to 10,000. Note that the code with transactions will be slower because it has the overhead of the additional instructions and does not benefit from transactions.

We are interested in comparing the slowdown induced by the transactions in the IBM platform ($\frac{\text{T\_I - NT\_I}}{\text{NT\_I}}$) to the slowdown induced by the transactions in the emulated platform ($\frac{\text{T\_E - NT\_E}}{\text{NT\_E}}$). If the former is comparable to the latter, our emulated platform correctly estimates the lightweight HTM overhead. Table V shows the results.

Mem Ops / Transaction	All_Reads		All_Writes	
	$\frac{\text{T\_I - NT\_I}}{\text{NT\_I}}$	$\frac{\text{T\_E - NT\_E}}{\text{NT\_E}}$	$\frac{\text{T\_I - NT\_I}}{\text{NT\_I}}$	$\frac{\text{T\_E - NT\_E}}{\text{NT\_E}}$
10	24.1%	129.0%	30.0%	99.3%
100	3.5%	12.9%	4.0%	8.9%
1000	0.3%	1.0%	0.4%	0.7%
10000	0.1%	0.0%	0.0%	0.1%

Table V: Slowdown induced by lightweight HTM in the IBM (I) and emulated (E) platforms.

The table shows that, for small to modest transactions, the overheads of the lightweight HTM are higher in the emulated platform. For large transactions, the overhead is negligible in both cases. Overall, our emulated platform does not underestimate the overhead of a lightweight HTM system. If anything, for small to modest transactions, our emulated platform conservatively assumes a higher overhead for the lightweight HTM. This makes our NoMap evaluation report smaller gains than would be realized with a real platform.

REFERENCES

[1] "Tizen," https://www.tizen.org/.

[2] "Espruino," http://www.espruino.com/.

[3] "Tessel2," https://tessel.io/.

[4] "MicroPython," https://micropython.org/.

[5] "Node.js," https://nodejs.org/en/.

[6] "Django," https://www.djangoproject.com/.

[7] "Ruby on Rails," http://rubyonrails.org/.

[8] "Scipy," https://www.scipy.org.

[9] "NumPy," http://www.numpy.org/.

[10] "Julia," https://julialang.org/.

[11] "Computer Language Benchmarks Game," http://shootout.alioth.debian.org/.

[12] "JavaScriptCore," http://trac.webkit.org/wiki/JavaScriptCore.

[13] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," in *Proc. of ICOOOLPS*, 2009, pp. 18–25.

[14] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu, "The HipHop Compiler for PHP," in *Proc. of OOPSLA*, 2012, pp. 575–586.

[15] "JRuby," http://jruby.org/.

[16] "Safari," http://www.apple.com/safari/.

[17] L. P. Deutsch and A. M. Schiffman, "Efficient Implementation of the Smalltalk-80 System," in *Proc. of POPL*, 1984.

[18] C. Chambers, D. Ungar, and E. Lee, "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes," in *Proc. of OOPSLA*, 1989.

[19] A. Holkner and J. Harland, "Evaluating the Dynamic Behaviour of Python Applications," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, 2009, pp. 19–28.

[20] B. Akerblom and T. Wrigstad, "Measuring Polymorphism in Python Programs," in *Proc. of DLS*, 2015, pp. 114–128.

[21] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How (and why) developers use the dynamic features of programming languages: The case of Smalltalk," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, May 2011.

[22] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proc. of PLDI*, 2010.

[23] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf, "Improved Type Specialization for Dynamic Scripting Languages," in *Proc. of DLS*, 2013, pp. 37–48.

[24] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the IBM POWER8 processor," *IBM Journal of Research and Development*, pp. 8:1–8:14, 2015.

[25] "SunSpider Benchmarks," http://www.webkit.org/perf/sun- spider/sunspider.html.

[26] "Kraken Benchmarks," http://krakenbenchmark.mozilla.org/.

[27] C. Lattner and V. Adve, "LLVM: a Compilation Framework for Lifelong Program Analysis Transformation," in *Proc. of CGO*, March 2004.

[28] U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," in *Proc. of PLDI*, July 1992, pp. 32–43.

[29] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," in *Proc. of CGO*, March 2003.

[30] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles, "Hardware Atomicity for Reliable Software Speculation," in *Proc. of ISCA*, June 2007.

[31] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff, "BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support," in *Proc. of ISCA*, December 2009.

[32] M. Arnold and B. G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," in *Proc. of PLDI*, 2001, pp. 168–179.

[33] T. Würthinger, C. Wimmer, and H. Mössenböck, "Array Bounds Check Elimination for the Java HotSpot™ Client Compiler," in *Proc. of PPPJ*, 2007, pp. 125–133.

[34] R. Bodík, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array Bounds Checks on Demand," in *Proc. of PLDI*, 2000, pp. 321–333.

[35] *IBM Power ISA Version 3.0B*. Open POWER Foundation, March 2017.

[36] *Intel Architecture Instruction Set Extensions Programming Reference*. Intel Corporation, February 2012.

[37] *IBM System Blue Gene Solution: Blue Gene/Q Application Development Manual*. IBM Corporation, March 2012.

[38] C. Click, "Azul's Experiences with Hardware Transactional Memory," in *HP Labs Bay Area Workshop on Transactional Memory*, January 2009.

[39] C. Jacobi, T. Slegel, and D. Greiner, "Transactional Memory Architecture and Implementation for IBM System Z," in *Proc. of MICRO*, 2012, pp. 25–36.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proc. of PLDI*, 2005.

[41] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," in *Proc. of ASPLOS*, October 1998, pp. 58–69.

[42] C. G. Ritson and F. R. Barnes, "An Evaluation of Intel's Restricted Transactional Memory for CPAs," in *Communicating Process Architectures*, 2013.

[43] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The Hiphop Virtual Machine," in *Proc. of OOPSLA*, 2014, pp. 777–790.

[44] G. Ottoni, "HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack," in *Proc. of PLDI*, 2018, pp. 151–165.

[45] C. Wimmer and S. Brunthaler, "ZipPy on Truffle: A Fast and Simple Implementation of Python," in *Proc. of SPLASH*, 2013, pp. 17–18.

[46] "V8 JavaScript Engine," https://developers.google.com/v8/.

[47] "SpiderMonkey Project," https://developer.mozilla.org/en-US/docs/SpiderMonkey, Mozilla.

[48] W. Ahn, J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas, "Improving JavaScript Performance by Deconstructing the Type System," in *Proc. of PLDI*, 2014, pp. 496–507.

[49] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications," in *the USENIX Conference on Web Application Development*, 2010.

[50] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of JavaScript benchmarks," in *Proc. of OOPSLA*, 2011.

[51] "Safari," https://web.archive.org/web/20170607191021/-https://www.apple.com/safari/, online; accessed 7-June-2017.

[52] S. Patel and S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Transactions on Computers*, June 2001.

[53] N. Neelakantam, D. R. Ditzel, and C. Zilles, "A Real System Evaluation of Hardware Atomicity for Software Speculation," in *Proc. of ASPLOS*, March 2010.

[54] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust Architectural Support for Transactional Memory in the POWER Architecture," in *Proc. of ISCA*, 2013, pp. 225–236.

[55] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta, Tech. Rep., January 2000.

[56] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *Proc. of HPCA*, 2011.

[57] M. Mehrara and S. Mahlke, "Dynamically accelerating client-side web applications through decoupled execution," in *Proc. of CGO*, 2011.

[58] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked Load: Architectural support for JavaScript type-checking on mobile processors," in *Proc. of HPCA*, 2011.

[59] L. Su and M. H. Lipasti, "Speculative Optimization Using Hardware-monitored Guarded Regions for Java Virtual Machines," in *Proc. of VEE*, 2007, pp. 22–32.