Dynamic Elimination of Overflow Tests in a Trace Compiler

Rodrigo Sol[1], Christophe Guillon[2], Fernando Magno Quintão Pereira[1] and Mariza A. S. Bigonha[1]

[1] UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil
{rsol,fpereira,mariza}@dcc.ufmg.br
[2] STMicroelectronics – 12 Jules Horowitz St, B.P. 217, 38019, Grenoble, France
Christophe.Guillon@st.com

Abstract. Trace compilation is a technique used by just-in-time (JIT) compilers such as TraceMonkey, the JavaScript engine in the Mozilla Firefox browser. Contrary to traditional JIT machines, a trace compiler works on only part of the source program, normally a linear path inside a heavily executed loop. Because the trace is compiled during the interpretation of the source program the JIT compiler has access to runtime values. This observation gives the compiler the possibility of producing binary code specialized to these values. In this paper we explore such opportunity to provide an analysis that removes unnecessary overflow tests from JavaScript programs. Our optimization uses range analysis to show that some operations cannot produce overflows. The analysis is linear in size and space on the number of instructions present in the input trace, and it is more effective than traditional range analyses, because we have access to values known only at execution time. We have implemented our analysis on top of Firefox's TraceMonkey, and have tested it on over 1000 scripts from several industrial strength benchmarks, including the scripts present in the top 100 most visited webpages in the Alexa index. We generate binaries to either x86 or the embedded microprocessor ST40-300. On the average, we eliminate 91.82% of the overflows in the programs present in the TraceMonkey test suite. This optimization provides an average code size reduction of 8.83% on ST40 and 6.63% on x86. Our optimization increases TraceMonkey's runtime by 2.53%.

1 Introduction

JavaScript is the most popular programming language used in the client-side of web applications [12]. Supporting this statement is the fact that JavaScript is used in 97 out of the 100 most popular websites in the alexa 2010 report [3]. Thus, it is very important that JavaScript programs benefit from efficient execution environments. Web browsers normally interpret programs written in this language. However, to achieve execution efficiency, JavaScript programs can be

[3] http://www.alexa.com

compiled during interpretation – a process called just-in-time (JIT) compilation. There are many ways to perform JIT compilation. *TraceMonkey* [14], the Mozilla Firefox 3.1's JIT compiler, translates the most executed *program traces* into machine code. A program trace is a linear sequence of code representing a path inside the program's control flow graph.

The Firefox JIT compiler, among many optimizations, does type specialization on JavaScript programs. For instance, JavaScript sees numbers as double precision 64-bit values in the IEEE 754 format [8, p.29]; however, TraceMonkey tries to manipulate them as integer values every time it is possible. Dealing with integers is much faster than handling floating-point arithmetics, but it is necessary to ensure that this optimization does not change the semantics of the program. So, every time an operation produces a result that might exceed the precision of the integer type, it is necessary to perform an overflow test. In case the test fails, float-point numbers must replace the original integer operands.

Overflow tests are pervasive in the machine code produced by TraceMonkey, a performance nuisance already acknowledged by the Mozilla community [4]. Overflow tests impose two major problems. First, because each test may force an early exit from tracing mode, these tests complicate optimizations that require code motion, such as partial redundancy elimination and instruction scheduling. Second, instructions that handle overflows increase code size: about 12.3% of the x86 code produced by TraceMonkey per script in this compiler's test suite implement overflow tests. This extra code is a complication on small devices that run JavaScript, such as the ST family of microcontrollers[5]. We have attacked this issue via a flow sensitive range analysis that proves that some overflow tests are redundant, and we present the results of this work in this paper. Our analysis runs in linear time on the number of instructions in the input trace. This analysis is implemented on top of TraceMonkey; however, it does not depend on a particular compiler. On the contrary, it works on any JIT engine that uses the trace paradigm to do code generation.

Our algorithm does range analysis [17, 20], i.e, it tries to estimate the lowest and greatest values that can be assigned to any variable. However, our approach differs from previous works because we use values known only at runtime in order to put bounds on the range of values that integer variables might assume during the program execution. This is a form of *partial evaluation* [18], yet done at runtime [4], because our analysis is invoked by a just-in-time compiler while the target application is being interpreted. By relying on such values we are able to perform much more aggressive range inferences than traditional static analyses. In terms of implementation, our analysis is similar to the ABCD algorithm that eliminates array bound-checks [3]; however, there are important differences. First, our algorithm is simpler, for it runs on a program trace, which is straight-line code. Because it runs on a trace, our algorithm is linear on the number of program variables, and not quadratic, as other analyses that keep def-use chains of variables. Second, we perform the whole analysis at once, in a

[4] `https://bugzilla.mozilla.org/show_bug.cgi?id=536641`
[5] `http://www.st.com/mcu/familiesdocs-51.html`

single traversal of the input trace, whereas ABCD runs on demand. In addition to these differences, the fact that we use the runtime value of variables lets us be less conservative.

We have implemented our analysis on top of TraceMonkey (release from 2010-10-3), the JIT compiler used by the Firefox web browser, and we have targeted two different processors, x86 and ST40. We have correctly compiled and run over one million lines of JavaScript code taken from a vast collection of benchmarks, including the TraceMonkey's test suite, and the top 100 webpages according to the Alexa index. Currently our implementation can only remove overflow tests from arithmetic operations performed on variables declared locally, i.e, we do not handle global variables yet. Nevertheless, we recognize 59% of the overflow tests in the TraceMonkey test suite, which contains over 800 scripts. On average, our algorithm eliminates 91.82% of the overflow tests that we recognize in these scripts, providing an average code size reduction of 6.63% on x86, and 8.83% on ST40. Our research-quality implementation adds 2.53% of time overhead on the core TraceMonkey implementation (time to compile and run the script) without other optimizations enabled. However, we speculate that an industrial-strength implementation of our algorithm will be able to obtain runtime gains. We have instrumented the Firefox browser, to check the behavior of our algorithm in actual webpages: on the average we remove 53.5% of the overflow tests per webpage. This number is lower when compared to the TraceMonkey's test suite because there are more global variables in actual webpages.

The remainder of this paper is organized as follows. Section 2 describes the TraceMonkey JIT compiler. In that section we show, by means of a simple example, how a trace is produced and represented. We describe our analysis in Section 3. Section 4 provides some experimental data that shows that our analysis is both fast and effective. We discuss related work in Section 5. Finally, Section 6 concludes this paper.

2 TraceMonkey in a Nutshell

There exists a number of recent works describing the implementation of trace-based JIT compilers, such as Tamarim-trace [6], HotpathVM [15], Yeti [27] and TraceMonkey [14]. In order to explain this new compilation paradigm, in this section we describe the TraceMonkey implementation. TraceMonkey has been built on top of *SpiderMonkey*, the original JavaScript interpreter used by the Firefox Browser. To produce `x86` machine code from JavaScript sources, TraceMonkey uses the *Nanojit* [6] compiler. The whole compilation process goes through three intermediate representations, a path that we reproduce in Figure 1.

1. **AST:** the abstract syntax tree that the parser produces from a script file.
2. **Bytecodes:** a stack based instruction set that SpiderMonkey interprets.
3. **LIR:** the low-level three-address code instruction representation that Nanojit receives as input.

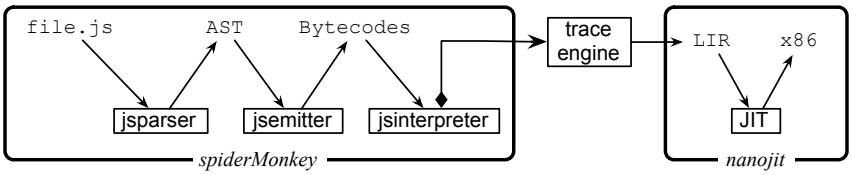


Fig. 1: The TraceMonkey JavaScript JIT compiler.

SpiderMonkey has not been originally conceived as a just-in-time compiler, a fact that explains the seemly excessive number of intermediate steps between the source program and the machine code. Segments of LIR instructions – a *trace* in TraceMonkey's jargon – are produced according to a very simple algorithm [14]:

1. each conditional branch is associated to a counter initially set to zero.
2. If the interpreter finds a conditional branch during program interpretation, then it increments the counter. The process of checking and incrementing counters is called, in TraceMonkey's jargon, the *monitoring phase*.
3. If the counter is two or more, and no trace exists for that counter, then the trace engine starts translating the bytecodes to a segment of LIR instructions, while they are interpreted. Overflow tests are inserted into this LIR segment. The process of building the trace is called *recording phase*.
4. Once the trace engine finds the original branch that started the recording process, the current segment is passed to Nanojit.
5. The Nanojit compiler translates the LIR segment, including the overflow tests, into machine code, which is dumped into main memory. The program flow is diverted to this code, and direct machine execution starts.
6. After the machine code runs, or in case an exceptional condition happens, e.g, an overflow test fails or a branch leaves the trace, the flow of execution goes back to the interpreter.

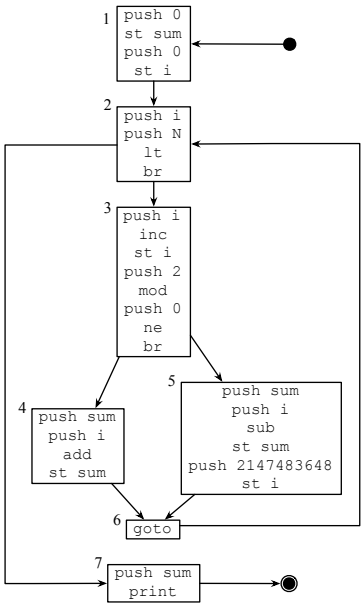
From bytecodes to LIR: we use the program in Figure 2 (a) to illustrate the process of trace compilation, and also to show how our analysis works. This is an artificial program, clearly too naive to find use in the real world; however, it contains the subtleties necessary to put some strain on the cheap analysis that must be used in the context of a just-in-time compiler. This program would yield the bytecode representation illustrated in Figure 2 (b). Notice that we took the liberty of simplifying the bytecode intermediate language used by TraceMonkey.

A key motivation behind the design of the analysis that we present in Section 3 is the fact that TraceMonkey might produce traces for program paths while these paths are visited for the first time. This fact is a consequence of the algorithm that TraceMonkey uses to identify traces. At the beginning of the interpretation process TraceMonkey finds the branch at Basic Block 2 in Figure 2

[6] `https://developer.mozilla.org/en/Nanojit`

```
foo (N) {
  var sum = 0;
  var i = 0;
  while (i < N) {
    i++;
    if (i % 2 != 0) {
      sum += i;
    } else {
      sum -= i;
      i = 2147483648;
    }
  }
  print(sum);
}

foo(10);
```



(a) **(b)**

Fig. 2: Example of a small JavaScript program and its bytecode representation.

(b), and the trace engine increments the counter associated to that branch. The next branch will be found at the end of Basic Block 3, and this branch's counter will be also incremented. The interpreter then will find the `goto` instruction at the end of Basic Block 6, which will take the program flow back to Block 2. At this moment the trace engine will increment again the counter of the branch in that basic block. Once the trace engine finds a counter holding the value two, it knows that it is inside a loop, and starts the recording phase, which produces a LIR segment. However, this segment does not correspond to the first part of the program visited: in the second iteration of the loop, Basic Block 5 is visited instead of Basic Block 4. In this case, the segment that is recorded is formed by Basic Blocks 2, 3, 5 and 6.

Because the trace that is monitored by the trace engine is not necessarily the trace that is recorded into a LIR segment, it is difficult to remove overflow tests during the recording phase. A conditional test, such as $a < N$, where N is constant, helps us to put bounds on the range of values that a might assume. However, in order to know that N is constant, we must ensure that the entire recorded segment does not contain commands that change N's value. Although it is not possible to remove overflow tests directly during the recording phase, it is possible to collect constraints on the ranges of the variables in this step. These constraints will be subsequently used to remove overflow tests at the

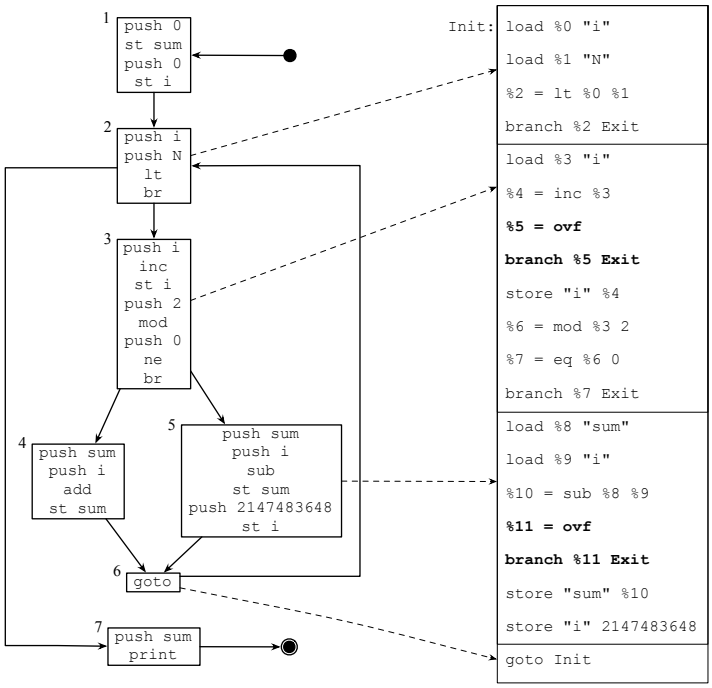


Fig. 3: This figure illustrates the match between the recorded trace and the LIR segment that the trace engine produces for it.

Nanojit level. Thus, we perform the elimination of overflow tests once the whole LIR segment has been produced, right before it is passed to Nanojit. Continuing with our example, Figure 3 shows the match between the recorded trace and the LIR segment that the trace engine produces for it.

From LIR to x86: Before passing the LIR segment to Nanojit, the interpreter augments this segment with two sequences of instructions: the prologue and the epilogue. The prologue contains code to map the values from the interpreter's execution environment to the execution environment where the compiled binary will run. The epilogue contains code to perform the inverse mapping. Nanojit produces machine code, e.g x86, ARM, ST40, from the LIR segment that it receives from the interpreter. The overflow tests are implemented as *side-exit* code: a sequence of instructions that has two functions: (i) it recovers the state of the program right before the overflow took place, and (ii) it jumps to the exit of the trace, returning control back to the interpreter. Figure 4 shows a simplified version of the x86 code that would be produced for our example trace. Notice that our analysis has no role at this point of the compilation process –

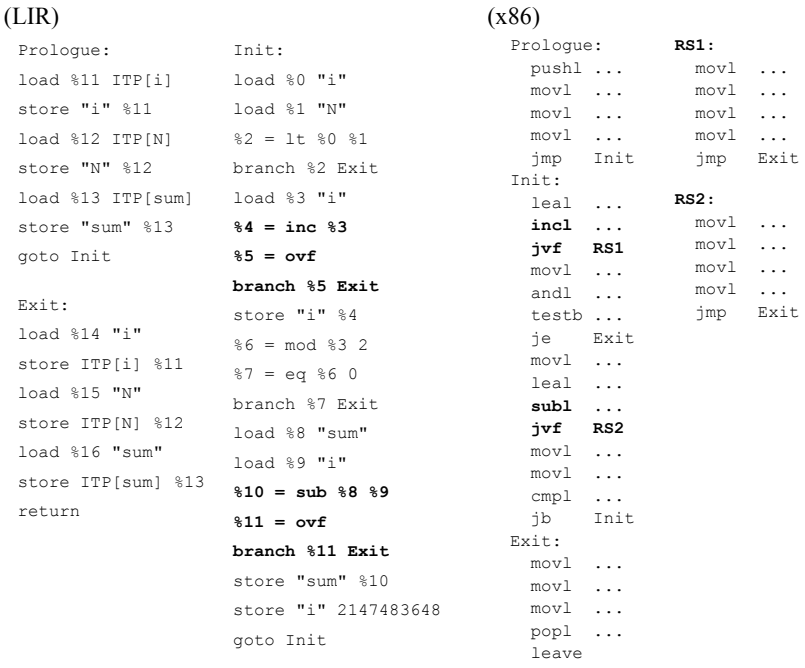


Fig. 4: The LIR code, after the insertion of a prologue and an epilogue, and the schematic x86 trace produced by Nanojit.

we include it in this discussion only to give the reader a full picture of the trace compiler.

3 Flow Sensitive Range Analysis

The flow sensitive range analysis that we use to remove overflow tests relies on a directed acyclic graph to determine the ranges of the variables. This graph, henceforth called *constraint graph*, has four types of nodes:

Name: represent program variables. Each node contains a counter, which represents a particular definition of a variable. These nodes are bound to an integer interval, which represents the range of values that the definition they encode might assume.

Assignment: denoted by `mov`, represent the copy of a value to a variable.

Relational: represent comparison operations, which are used to put bounds on the ranges of the variables. The comparison operations considered are: equals (`eq`), less than (`lt`), greater than (`gt`), less than or equals (`le`), and greater than or equals (`ge`).

Arithmetic: represent operations that might require an overflow test. We have two types of arithmetic nodes: binary and unary. The binary operations are addition (`add`), subtraction (`sub`), and multiplication (`mul`). The unary operations are increment (`inc`) and decrement (`dec`). Division is not handled because it might legitimately produce floating-point results.

The analysis proceeds in two phases: construction of the constraint graph and range propagation. The remaining of this section describes these phases.

3.1 Construction of the Constraint Graph

We build the constraint graph during the trace recording phase of TraceMonkey, that is, while the instructions in the trace are being visited and a LIR segment is being produced. In order to associate range constraints to each variable in the source program we use a program representation called *Extended Static Single Assignment* (e-SSA) [3] form, which is a superset of the well known SSA form [9]. In the e-SSA representation, a variable is renamed after it is assigned a value, or after it is used in a conditional.

Converting a program to e-SSA form requires a global view of the program, a requirement that a trace compiler cannot fulfill. However, given that we are compiling a program trace, that is, a straight line segment of code, the conversion is very easy, and happens at the same time that the constraint graph is built, e.g, during the trace recording step. The conversion works as follows: counters are mantained for every variable. Whenever we find a use of a variable v we rename it to v_n, where n is the current value of the counter associated to v. Whenever we find a definition of a variable we increment its counter. Considering that the variables are named after their counters, incrementing the counter of a variable effectively creates a new name definition in our representation. So far our renaming is just converting the source program into Static Single Assignment form [9]. The e-SSA property comes from the way that we handle conditionals. Whenever we find a conditional, e.g, $a < b$, we learn new information about the ranges of a and b. Thus, we redefine a and b, by incrementing their counters.

There are two events that change the bounds of a variable: *simple assignments* and *conditional tests*. The first event determines a unique value for the variable. The second puts a bound in one of the variable's limits, lower or upper. These are the events that cause us to increment the counters associated to variables. Thus, it is possible to assign unique range constraints to each new definition of a variable. These range constraints take into consideration the current value of the variables at the time the variable is found by the trace engine. We determine these values by inspecting the interpreter's stack. We have designed the following algorithm to build the constraint graph:

1. initialize counters for every variable in the trace. We do this initialization on the fly: the first time a variable name is seen we set its counter to zero, otherwise we increment its current counter. If we see a variable for the first time, then we mark it as *input*, otherwise we mark it as *auxiliary*. If a variable

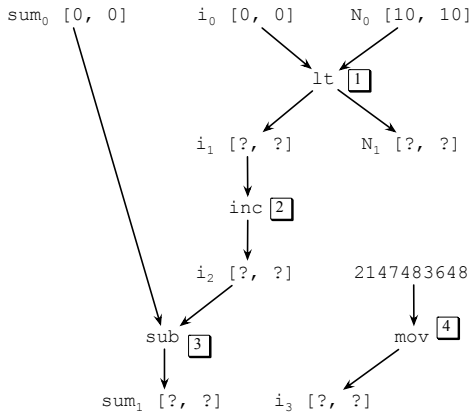


Fig. 5: Constraint graph for the trace in Figure 3. The numbers in boxes denote the order in which the nodes were created.

is marked as *input*, then we set its upper and lower limits to the value that the interpreter currently holds for it. Otherwise, we set the variable boundaries to undefined values, e.g, $]-\infty, +\infty[$.

2. For each instruction i that we visit:
   (a) if i is a relational operation, say $v < u$, we build a relational node that has two predecessors: variable nodes v_x and u_y, where x and y are counters. This node has two successors, v_{x+1} and u_{y+1}.
   (b) For each binary arithmetic operation, e.g, $v = t + u$, we build an arithmetic node n. Let the nodes related to variables t_x and u_y be the predecessors of n, and let v_z be its successor.
   (c) For each unary operation, say $u{+}{+}$, we build a node n, with one predecessor u_x, and one successor u_{x+1}.
   (d) For each copy assignment, e.g, $v = u$, we build an assignment node, which has predecessor u_x, and successor v_{y+1}, assuming y is v's counter.

Figure 5 shows the constraint graph to our running example, assuming that the function `foo` was called with the parameter $N = 10$. Notice that we bound the input variables to intervals: $sum_0 \subseteq [0, 0]$, $i_0 \subseteq [0, 0]$ and $N_0 \subseteq [10, 10]$. When constructing the constraint graph, it is important to maintain a list with the order in which each node was created. This list, which we represent by the numbers in boxes, will be later used to guide the range propagation phase.

3.2 Range Propagation

During the range propagation phase we find which overflow tests are necessary, and which ones can be safely removed from the target code. The propagation of

Arithmetics

$$x + (+\infty) = +\infty + x = +\infty, x \neq -\infty \qquad x + (-\infty) = -\infty + x = -\infty, x \neq +\infty$$

$$x \times (\pm\infty) = \pm\infty \times x = \pm\infty, x > 0 \qquad x \times (\pm\infty) = \pm\infty \times x = \mp\infty, x < 0$$

Increment: $x_1 = x_0 + 1$

$$\frac{x_0.l + 1 > \text{MAX\_INT}}{x_1.l = +\infty} \qquad \frac{x_0.u + 1 > \text{MAX\_INT}}{x_1.u = +\infty}$$

$$\frac{x_0.l + 1 \leq \text{MAX\_INT}}{x_1.l = x_0.l + 1} \qquad \frac{x_0.u + 1 \leq \text{MAX\_INT}}{x_1.u = x_0.u + 1}$$

Addition: $x = a + b$

$$\frac{a.l + b.l < \text{MIN\_INT}}{x.l = -\infty} \qquad \frac{a.u + b.u < \text{MIN\_INT}}{x.u = -\infty}$$

$$\frac{a.l + b.l > \text{MAX\_INT}}{x.l = +\infty} \qquad \frac{a.u + b.u > \text{MAX\_INT}}{x.u = +\infty}$$

$$\frac{\text{MIN\_INT} \leq a.l + b.l \leq \text{MAX\_INT}}{x.l = a.l + b.l} \qquad \frac{\text{MIN\_INT} \leq a.u + b.u \leq \text{MAX\_INT}}{x.u = a.u + b.u}$$

Multiplications: $x = a \times b$

$$\frac{a.l \times b.l < \text{MIN\_INT}}{x.l = -\infty} \qquad \frac{a.u \times b.u < \text{MIN\_INT}}{x.u = -\infty}$$

$$\frac{a.l \times b.l > \text{MAX\_INT}}{x.l = +\infty} \qquad \frac{a.u \times b.u > \text{MAX\_INT}}{x.u = +\infty}$$

$$\frac{\text{MIN\_INT} \leq a.l \times b.l \leq \text{MAX\_INT}}{x.l = \text{MIN}(a.\{l,u\} \times b.\{l,u\})} \qquad \frac{\text{MIN\_INT} \leq a.u \times b.u \leq \text{MAX\_INT}}{x.u = \text{MAX}(a.\{l,u\} \times b.\{l,u\})}$$

Fig. 6: Range propagation for arithmetic nodes. Decrements and subtractions are similar to increments and additions. We use $a.\{l,u\} \times b.\{l,u\}$ as a short form for $(a.l \times b.l, a.l \times b.u, a.u \times b.l, a.u \times b.u)$.

ranges is preceded by a trivial initialization step, when we replace the constraints of the input variables with $]-\infty, +\infty[$ if those variables have been updated inside the trace. This is the case, for example, of variables $\texttt{sum}_0$ and $\texttt{i}_0$ in the example from Figure 5. In order to do the propagation of range intervals, we visit all the arithmetic and relational nodes, in topological order. This ordering is given by the "age" of the node. Nodes that have been created earlier, during

Less than with constant: $(a_1, N_1) \leftarrow (a < N)?$, when $a.l \leq N.l = N.u$	
$a_1.u = \text{MIN}(a.u, N.u - 1)$	$N_1.l = N.l$
$a_1.l = a.l$	$N_1.u = N.u$
General less than: $(a_1, b_1) \leftarrow (a < b)?$, when $[a.l, a.u] \cap [b.l, b.u] \neq \emptyset$	
$a_1.u = \text{MIN}(a.u, b.u - 1)$	$b_1.l = \text{MAX}(b.l, a.l + 1)$
$a_1.l = a.l$	$b_1.u = b.u$
Equal to constant: $(a_1, N_1) \leftarrow (a = N)?$, when $a.l \leq N.l = N.u \leq a.u$	
$a_1.l = N.l$	$N_1.l = N.l$
$a_1.u = N.u$	$N_1.u = N.u$
General equals: $(a_1, b_1) \leftarrow (a = b)?$, when $[a.l, a.u] \cap [b.l, b.u] \neq \emptyset$	
$a_1.l = \text{MAX}(a.l, b.l)$	$b_1.l = \text{MAX}(a.l, b.l)$
$a_1.u = \text{MIN}(a.u, b.u)$	$b_1.u = \text{MIN}(a.u, b.u)$

Fig. 7: Range propagation for two relational nodes. The value of variable N is not updated inside the trace.

the construction of the constraint graph are visited first. Notice that we get this ordering for free, simply storing the arithmetic and relational nodes in a queue while we create them, during the construction of the constraint graph. If every variable is defined before being used, then by following the node creation order, the propagation of ranges guarantees that whenever we reach an arithmetic, relational or assignment node, all the name nodes that point to it have been visited before.

Each arithmetic and relational node causes the propagation of ranges in a particular way, always preserving the invariant that the lower bound of an interval is less than or equal its upper bound. Figure 6 shows the updating rules that we use for some arithmetic nodes. We use standard IEEE extended arithmetics [16], except that, to improve our analysis, we assume that the result of multiplying infinity and zero is zero. Furthermore, we let the sum of $]-\infty, x[$ plus $]+\infty, +\infty[$ to be $]-\infty, +\infty[$, and vice-versa. We denote the interval $[l, u]$ associated to a node a by $a.l$ and $a.u$. For simplicity, we state the range propagation rules using infinity precision arithmetics. Our actual implementation uses wrapping semantics, i.e: if $x + 1 > x$, then $x = +\infty$. Only arithmetic nodes might cause overflows; and each of our five types of arithmetic nodes might

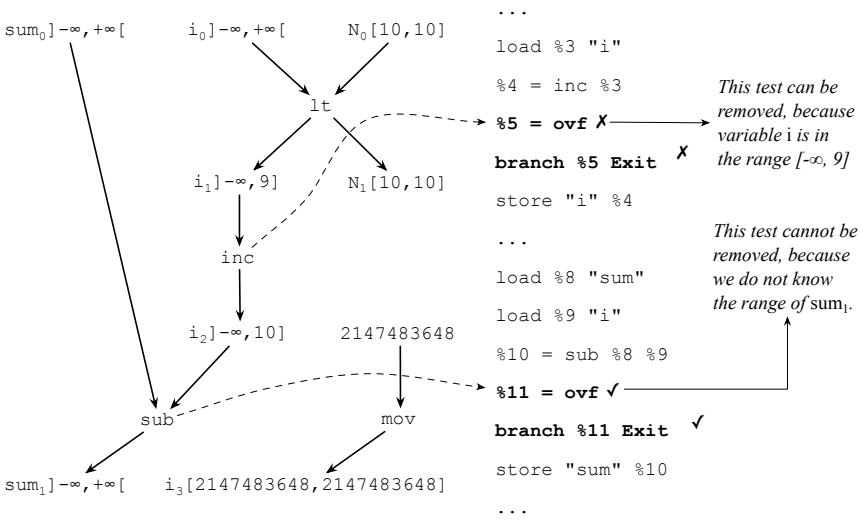


Fig. 8: Once we know the ranges of each variable involved in an arithmetic operation we can remove the associated overflow test, if it is redundant.

produce overflows in different ways. For instance, if we find an increment of $[-\infty, v.u]$, then we keep the overflow associated to this node, as long as $v.u+1 \leq$ MAX_INT. Figure 7 shows the updating rules for some relational nodes. These nodes are not associated to any overflow test, but they help us to constrain the range of intervals bound to program variables. As an optimization, we do not update the ranges of variables that are not defined inside the trace. This is the case, for instance, of Variable N in the program of Figure 2 (a). Notice that we do not perform range updates that break the invariant $v.l \leq v.u$.

After range propagation we have a conservative estimate of the intervals that each integer variable might assume during program execution. This information allows us to go over the LIR segment, before it is passed to Nanojit, removing the overflow tests that our analysis has deemed unnecessary. Figure 8 shows this step: we have removed the overflow test from the `inc` operation, because it receives as input a node bound to the interval $]-\infty, 9]$; hence, its result will never be greater than 10. However, our analysis cannot prove that the test in the `sub` operation is also unnecessary, although that is the case.

3.3 Complexity Analysis

The proposed algorithm has running time linear on the number of instructions of the source trace. To see this fact, notice that the constraint graph has a number of conditional and arithmetic nodes proportional to the number of instructions

in the trace. During our analysis we traverse the constraint graph one time, at the range propagation phase, visiting each node only once. We do not have to preprocess the graph beforehand, in order to sort it topologically, because we get this ordering from the sequence in which instructions are visited in the source trace. Our algorithm is also linear in terms of space, because each type of arithmetic node has a constant number of predecessors and successors. This low complexity is in contrast to the complexity of many graph traversal algorithms, which are $O(E)$, where E is the number of edges in the graph. These algorithms have a worst case quadratic complexity on the number of vertices, because $E = O(V^2)$. We do not suffer this drawback, for in our case $E = O(V)$.

4 Experimental Results

We have implemented our algorithm on top of TraceMonkey, revision 2010-10-01. Our implementation handles the five arithmetic operations described in Section 3.1, namely additions, subtractions, increments, decrements and multiplications. We perform the range analysis described in Section 3 during the recording phase of TraceMonkey, that is, while a segment of JavaScript bytecodes is translated to a segment of LIR. We have also modified Nanojit to remove the overflow tests, given the results of our analysis. Our current implementation has some shortcomings, which are all due to our limited understanding of TraceMonkey's implementation, and that we are in the process of overcoming:

- we cannot read values stored in global variables, a fact that hinders us from removing overflows related to operations that manipulate these values.
- We cannot recognize when TraceMonkey starts tracing constructs such as `foreach{...}`, `while(true){...}` and loops that range on iterators.

The benchmarks. We have correctly compiled and executed over one million lines of JavaScript code that we took from three different benchmark suites:

Alexa top 100: the 100 most visited webpages, according to the Alexa index [7]. This list includes `Google`, `Facebook`, `Youtube`, etc. We tried to follow Richards *et al.*'s methodology [12], manually visiting each of these pages with our instrumented Firefox. Notice that we present results for only 80 of these benchmarks, because we did not find overflow tests in 20 webpages.

Trace-Test: the test suite that is used in TraceMonkey [8]. This collection of scripts includes popular benchmarks, such as Webkit's Sunspider and Google's V8. Many scripts do not contain arithmetic operations; thus, we show results only for the 224 scripts that contains at least one overflow test.

PeaceKeeper: an industrial strength benchmark used to test browsers [9].

[7] `http://www.alexa.com/`

[8] `http://hg.mozilla.org/tracemonkey/file/c3bd2594777a/js/src/trace-test/tests`

[9] `http://service.futuremark.com/peacekeeper/index.action`

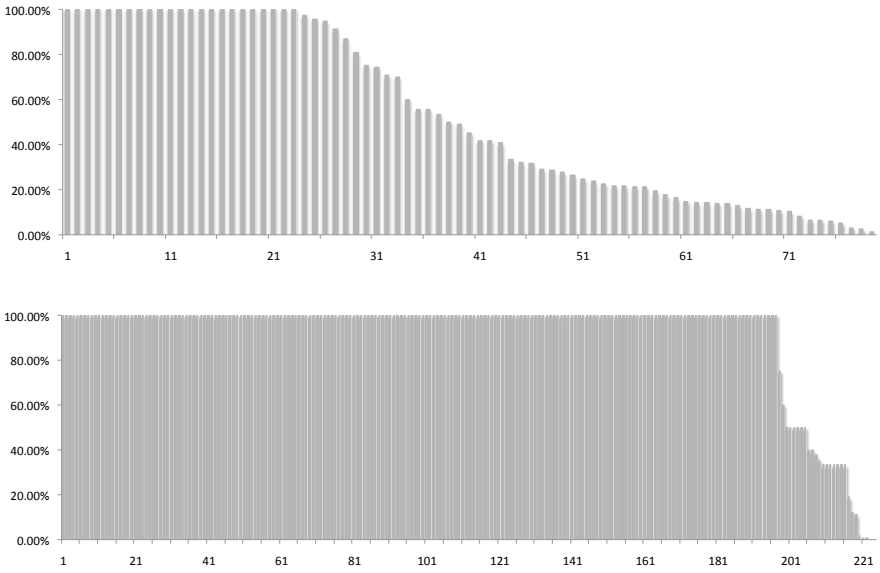


Fig. 9: The effectiveness of our algorithm in terms of percentage of overflow tests removed per script. (Top) Alexa; geo mean: 53.50%. (Bottom) Trace-test; geo mean: 91.82%. Hardware: same results for ST4 and x86. The 224 scripts are sorted by average effectiveness. We removed 700/859 overflow tests for Peace-Keeper (one script).

The hardware. We have used our modified TraceMonkey compiler in two different hardware: (i) x86: 2GHz Intel Core 2 Duo, with 2GB of RAM, featuring Mac OS 10.5.8. (ii) ST40-300: a real-time processor manufactured by STMicroelectronics. The ST40 runs STLinux 2.3 (kernel 2.6.32), has a 450MHz clock, provides 200MB of physical memory and 512MB of virtual memory. This target is a two level cache architecture with separated 32KB intruction and 32KB data L1 caches on top of a 256KB unified L2 cache.

How effective is our algorithm? Figure 9 shows the effectiveness of our algorithm when we run it on the Alexa webpages and the Trace-Test scripts. We have ordered the scripts on the x-axis according to the effectiveness of our algorithm. We present static numbers, in the sense that we have not instrumented the binary traces to see how many times each overflow test is executed. On the average, we remove 91.82% of the overflows tests from the scripts in Trace-Test, and 53.50% of the overflow tests from the Alexa webpages. This is the geometric mean; that is, we are very effective in removing overflow tests from most of the scripts. In particular, we remove most of the overflow tests used in counters that index simple loops. However, the arithmetic mean is much lower, because of the outliners. Three of these scripts, which are part of SunSpider (included in Trace-

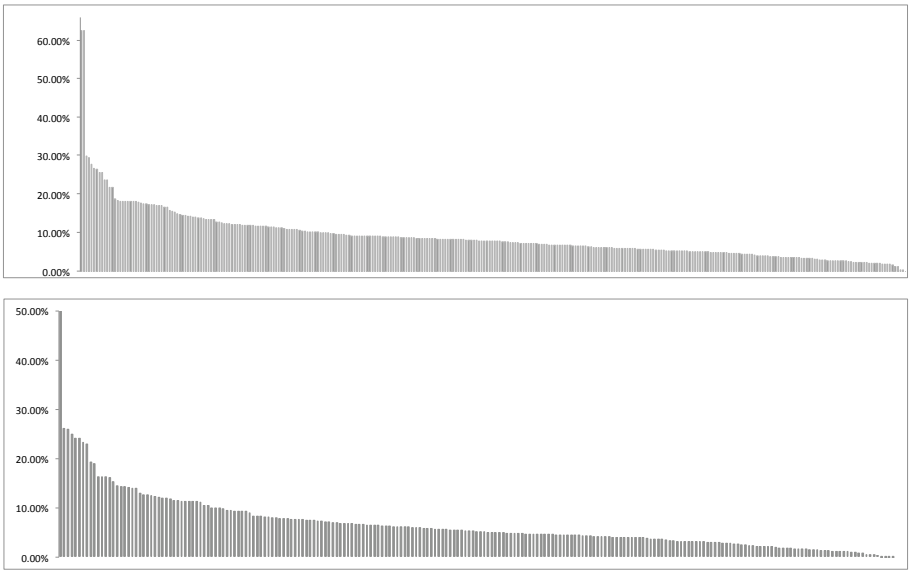


Fig. 10: Size reduction per script in two different architectures. (Top) ST4; geo mean: 8.83%. (Bottom) x86; geo mean: 6.63% Benchmark: Trace-Test. The 224 scripts are sorted by average size reduction.

Test) deal with cryptography and manipulate big numbers. They account for 8,756 tests, out of which we removed only 347. In absolute terms we remove 961 out of 11,052 tests in the Trace-Test benchmark, and 2,680 out of 11,126 tests in the Alexa benchmark. The arithmetic mean, in this case, is 8.7% for Trace-Test, and 24.08% for Alexa. Figure 9 does not contain a chart for PeaceKeeper, because this benchmark contains only one script. We removed 700 tests out of the 859 tests that we found in this script.

What is the code size reduction due to the elimination of overflow tests? On the x86 target, TraceMonkey uses eight instructions to implement each overflow test, which includes the branch-if-overflow instruction, load instructions to reconstruct the interpreter state and an unconditional jump back to interpreter mode. On the ST4 microprocessor, TraceMonkey uses 10 instructions. We eliminate all these instructions after removing an overflow test. Figure 10 shows the average size reduction that our algorithm obtains on each target. On the average, we shrink each script by 8.83% on the ST4, and 6.63% on the x86. On most of the scripts the size reduction is modest; however, there are cases in which our algorithm decreases the binaries by over 60.0%. Notice that by using integers instead of floating-point numbers TraceMonkey already reduces the size of the binaries that it produces. For instance, on the ST4, TraceMonkey obtains an average 31.47% of size reduction with the floating-point to integer optimization.

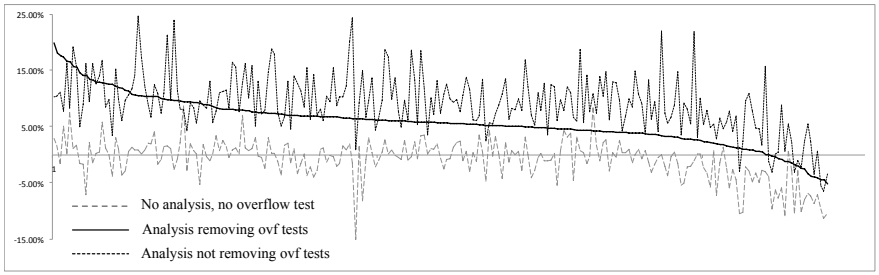


Fig. 11: Impact of our algorithm on TraceMonkey's runtime. Benchmark: Trace-Test. Hardware: x86. The 224 scripts are sorted by average runtime increase. We run each test 8 times, ignoring smallest and largest outliers. Zero is TraceMonkey's runtime without our analysis and with overflow tests enabled.

What is the effect of our algorithm on TraceMonkey's runtime? Figure 11 shows that, on the average, our algorithm has increased the runtime of TraceMonkey by 2.56% on the x86. This time includes parsing, interpreting, JIT compiling and executing the scripts. Our algorithm increases the time of JIT compilation, but decreases the time of script execution. So far the cost, in time, is negative for two reasons: a prototype implementation and a low benefit optimization. Figure 11 shows that running our analysis without disabling overflows tests increases TraceMonkey's runtime by 4.12%. Furthermore, the effects of avoiding the overflows tests are negligible. Most of the time, the impact of an overflow test is the cost of executing the x86 instruction *branch-if-overflow*; however, this instruction is normally predicted as not taken. If we disable every overflow test without running the analysis, then we improve the runtime by 1.56% on the scripts that finish correctly. The noise in the chart is due to short runtimes – the slowest script executes for 0.73s, and the fastest for 0.01s.

Why sometimes we fail to remove overflows? We have performed a manual study on the 42 smallest programs from Trace-Test in which we did not remove overflow tests. Each of these tests contains a loop indexed by an induction variable. Figure 12 explains our findings. In 69% of the traces, we failed to remove the test because the induction variable was bounded by a global variable. As we explained before, our implementation is not able to identify the values of global variables; hence, we cannot use them in the estimation of ranges. In 17% of the remaining cases the trace is produced after a `foreach` control structure, which our current implementation fails to recognize. Once we fix these omissions, we will be able to remove at least one more overflow test in 36 out of these 42 scripts analyzed. There was only one script in which our algorithm legitimately failed to remove a test: in this case the semantics of the program might lead to a situation in which an overflow, indeed, happens.

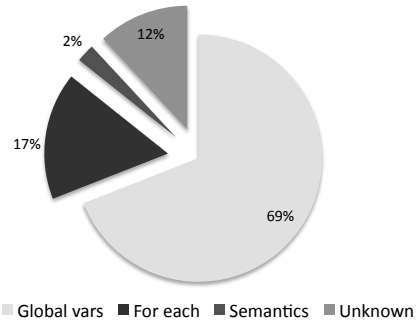


Fig. 12: The main reasons that prevent us from removing overflow tests.

What we gain by knowing the runtime value of variables? We perform more aggressive range estimations than previous range analyses described in the literature, because we are running alongside a JIT compiler, a fact that gives us the runtime value of variables, including loop limits. Statically we can only rely on constants to start placing bounds in variable ranges. Non-surprisingly, a static implementation of our algorithm removes only 472 overflow tests from the Trace-Test, which corresponds to 37% of the tests that we remove dynamically.
Why are effectiveness results so different for Trace-Test and Alexa? Figure 9 shows that our algorithm is substantially more effective when applied on the programs in Trace-Test than on the programs in the Alexa top 100 pages. This happens because most of the scripts in the Trace-Test benchmark suite are small, and contain only simple loops controlled by locally declared variables, which our algorithm can recognize. In order to compare the two test suites, we have used our instrumented Firefox to produce the histogram of assembly instructions that TraceMonkey generates for each benchmark. Figure 13 shows the results of our findings. We have produced this chart by plotting pairs $(i, f(i))$, which we define as follows:

1. let I be the list formed by the 100 most common SpiderMonkey bytecodes found in the Alexa benchmark. We order the bytecodes i according to $a(i)$, the frequency of occurrences of i in the Alexa collection.
2. For each bytecode $i \in I$ and benchmark $B \in \{\text{Trace-Test}, \text{PeaceKeeper}\}$, we let $b(i)$ be i's frequency in B, and we let $f(i) = a(i) - b(i)$.

The closer to zero the values of $f(i)$, more similar is the frequency of i in the Alexa collection and in the other benchmark. Using this rough criterion, we see that PeaceKeeper is closer to Alexa than Trace-Test. Summing up the absolute values of $f(i)$, we find 0.24 for PeaceKeeper and 0.75 for Trace-Test. Coincidentally, our analysis is more effective on Trace-Test than on PeaceKeeper.

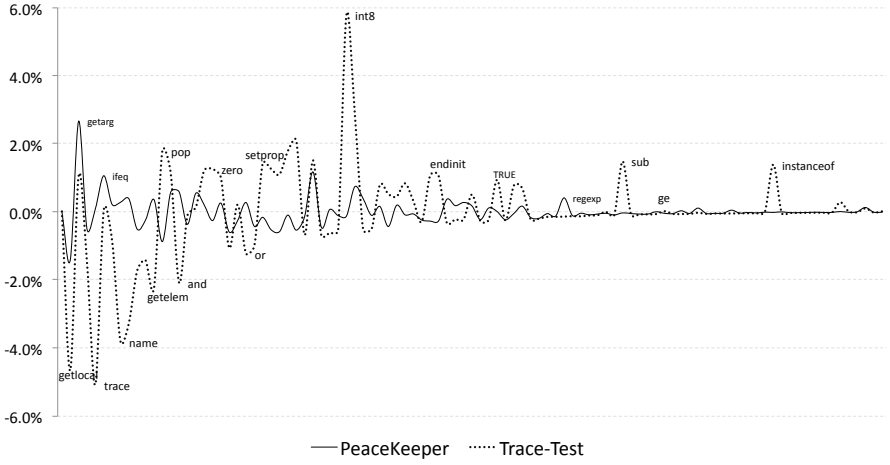


Fig. 13: A histogram of the bytecodes present in Trace-Test and PeaceKeeper, compared to the bytecodes present in the Alexa collection. X-axis: instructions i ordered by their frequency in the Alexa collection.

5 Related Work

Just-in-time compilers are old allies of programming language interpreters. Since the seminal work of John McCarthy [19], the father of Lisp, a multitude of JIT compilers have been designed and implemented. A comprehensive survey on just-in-time compilation is given by John Aycock [1].

The optimization that we propose in this paper is a type of partial evaluation at runtime. Partial evaluation is a technique in which a compiler optimizes a program, given a partial knowledge of its input [18]. Variations of partial evaluation have been used to perform general code optimization [22, 23]. This type of partial evaluation in which the JIT compiler uses runtime values to produce better binaries is sometimes called *specialization by need* [21]. Implementations that use this kind of technique include Python's Psyco JIT compiler [21], Matlab [7, 10] and Maple [4]. Partial evaluation has been used in the context of just-in-time compilation mostly as a form of type specialization [7, 5, 14]. That is, once the compiler proves that a value belongs into a certain type, it uses this type directly, instead of resorting to costly boxing and unboxing techniques.

The competition between popular browsers has brought renewed attention to trace compilation. However, the idea of focusing the compilation effort on code traces, instead of whole methods, has been known at least since the work of Bala *et al.* [2], or even before if we consider trace scheduling [11]. Traces have been independently integrated on JIT-compilers by Gal [13] and Zaleski [27]. Currently, there are many trace-based JIT-compilers [6, 14, 15, 27]. For a formal overview of trace compilation, we recommend the work of Guo and Palsberg [26].

Our algorithm to remove overflow tests is a type of *range analysis* [17, 20]. Range analysis tries to infer lower and upper bounds to the values that a variable might assume during the program execution. In general these algorithms rely on theorem provers, an approach deemed too slow to a JIT compiler. Bodik *et al.* [3] have described a specialization of range analysis that removes array bound checks, the ABCD algorithm, that targets JIT compilation. Zhendong and Wagner [25] have described a type of range analysis that can be solved in polynomial time. Stephenson *et al.* [24] have used a polynomial time analysis to infer the bitwidth of each integer variable used in the source program. Contrary to our approach, all these previous algorithms work on the static representation of the source program. Such fact severely constraints the amount of information that these analysis can rely on. By knowing the runtime value of program variables we can perform a very extensive, yet fast, range analysis.

6 Conclusion

This paper has presented a new algorithm to remove redundant overflow tests during the JIT compilation of JavaScript programs. The proposed algorithm works in the context of a trace compiler. Our algorithm is able to find very precise ranges for program variable by inspecting their runtime values. We have implemented our analysis on top of TraceMonkey, the JIT compiler used by the Mozilla Firefox browser to speed up the execution of JavaScript programs. We have submitted our implementation to Mozilla, as a Firefox patch, available at `http://github.com/rodrigosol/Dynamic-Elimination-Overflow-Test`. In terms of future work, we would like to improve the performance and effectiveness of our implementation. For instance, currently we can only read the runtime values of local variables, a limitation that we are working to overcome.

Acknowledgments: This project has been made possible by the cooperation FAPEMIG-INRIA, grant 11/2009. Rodrigo Sol is supported by the Brazilian Ministry of Education under CAPES and CNPq. We thank David Mandelin from the Mozilla Foundation for helping us with TraceMonkey.

References

1. John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
2. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI*, pages 1–12. ACM, 2000.
3. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
4. Jacques Carette and Michael Kucera. Partial evaluation of maple. In *PEPM*, pages 41–50. ACM, 2007.
5. C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.

6. Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, pages 71–80. ACM, 2009.
7. Maxime Chevalier-Boisvert, Laurie J. Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In *CC*, pages 46–65. Springer, 2010.
8. ECMA Committe. *ECMAScript Language Specification*. ECMA, 5th edition, 2009.
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
10. Daniel Elphick, Michael Leuschel, and Simon Cox. Partial evaluation of matlab. In *GPCE*, pages 344–363. Springer-Verlag New York, Inc., 2003.
11. Joshph A. Fisher. Trace scheduling: A technique for global microcode compaction. *Trans. Comput.*, 30:478–490, 1981.
12. Richards G., Lebresne S., Burg B., and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, pages 1–12, 2010.
13. Andreas Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine, 2006.
14. Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Jess Ruderman, Edwin Smith, Rick Reitmair, Mohammad R. Haghighat, Michael Bebenita, Mason Change, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM, 2009.
15. Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE*, pages 144–153, 2006.
16. David Goldberg. What every computer scientist should know about floating-point arithmetic. *Comput. Surv.*, 23:5–48, 1991.
17. W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250, 1977.
18. Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1st edition, 1993.
19. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of ACM*, 3(4):184–195, 1960.
20. Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.
21. Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, pages 15–26. ACM, 2004.
22. Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *TOPLAS*, 25(4):452–499, 2003.
23. Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. *SIG. Not.*, 40(10):327–343, 2005.
24. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.
25. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.
26. Shu yu Guo and Jens Palsberg. The essence of compiling with traces. In *POPL*, page to appear. ACM, 2011.
27. Mathew Zaleski. *YETI: a gradually extensible trace interpreter*. PhD thesis, University of Toronto, 2007.