Removing Overflow Tests via Run-Time Partial Evaluation

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

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{rsol,fpereira,mariza}@dcc.ufmg.br

Abstract. *Trace compilation is a new 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 use only part of 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 the values manipulated at run-time, a fact that opens the doors to very aggressive optimizations. This fact gives the compiler writer the possibility of improving code via partial evaluation techniques at run-time. In this paper we rely on this observation to present an analysis that removes unnecessary overflow tests from JavaScript programs. Our optimization uses a variation of range analysis to estimate the lower and upper limits of the values assumed by variables during the execution of the program. Such information allows us to prove that some operations cannot produce overflows. Our analysis runs in linear time, in terms of both time and space, and is much more effective than traditional range analyses, given that we have access to values known only at execution time. We have implemented our analysis on top of TraceMonkey, and our experiments show that our non-optimized implementation removes 56% of all the overflow tests that we found, reducing the code size by 2.6%, while increasing the overall time in 6.3%.*

1. Introduction

JavaScript is a dynamically and weakly typed language, heavily used in the client side of web applications [Chugh et al. 2009, Gal et al. 2009]. Web browsers normally interpret JavaScript programs. However, to achieve execution efficiency, JavaScript programs can be compiled during interpretation – a process called just-in-time (JIT) compilation. There are many ways to perform JIT compilation. One of them is the V8 approach, which compiles each JavaScript method before the method is called [1]. Another is the approach used in the Mozilla Firefox 3.1's JIT compiler, the *TraceMonkey* [Gal et al. 2009]. In this second strategy, bytecodes are interpreted, and the program *traces* often executed are directly compiled to machine code. A program trace is a linear sequence of code representing a path inside the program's control flow graph. The Firefox method, proposed by Andreas Gal and Michael Franz [Gal 2006, Gal and Franz 2006, Gal et al. 2006], is the approach discussed in this paper.

The Firefox JIT compiler, among many optimizations, tries to perform some simple type specialization on JavaScript programs. This means, for instance, that, although JavaScript sees numbers as float-point values, TraceMonkey tries to manipulate them as

[1]http://code.google.com/apis/v8/design.html

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. For instance, JavaScript assumes arbitrary precision arithmetics, a property that cannot be guaranteed with 32-bit integer values. 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 flaw already acknowledged by the Mozilla community [2]. A major problem of overflow tests is the increase in code size: about 3.2% of the binaries produced by TraceMonkey are overflow tests. This extra code, often unnecessary, is a complication on mobile applications that feature JavaScript capabilities [3]. We have attacked this bug via dynamic analysis, and we present the results of this work in this paper. We have designed and implemented a flow sensitive analysis that proves that some overflow tests are redundant. Our analysis runs in linear time on the number of instructions in the source program. This analysis is implemented on top of TraceMonkey; however, we emphasize that the analysis does not depend on a particular compiler. On the contrary, it works on any JIT engine that uses the trace paradigm of code generation. Our experiments show that this analysis is fast enough to be used in the context of a JIT compiler. A non-optimized implementation of our algorithm adds 6.3% of time overhead on the core TraceMonkey implementation, running without other optimizations enabled. Our implementation finds about 59% of the overflow tests in the TraceMonkey test suite. Our algorithm eliminates 54% of these tests, reducing the size of the binaries in 2.6%.

Our analysis is a variation of range analysis [Harrison 1977, Patterson 1995]. However, our approach differs from previous work because we use values known only at run-time 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* [Jones et al. 1993], yet done at run-time [Carette and Kucera 2007], 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.

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

Trace compilation is a new technique, and the literature contains the description of only two implementations: one is Tamarim-trace [Chang et al. 2009], a JIT compiler implemented on top of Tamarim-central, the Adobe's Flash 9 engine. The other is TraceMonkey. 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

[2] `https://bugzilla.mozilla.org/show_bug.cgi?id=536641`
[3] `http://opensource.nokia.com/projects/S60browser/`

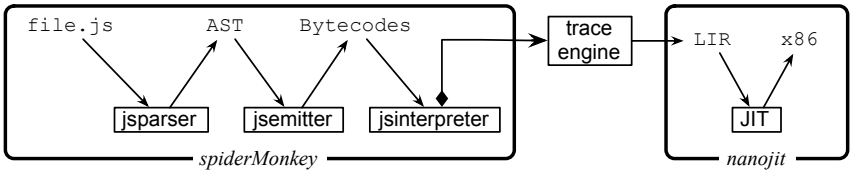


Figure 1. The TraceMonkey JavaScript JIT compiler.

original JavaScript interpreter used by the Firefox Browser. In order to produce `x86` machine code from JavaScript sources, TraceMonkey uses the *nanojit* [4] 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 JavaScript source file.
2. **Bytecodes:** a stack based instruction set that is directly interpreted by the spiderMonkey interpreter.
3. **LIR:** the low-level three-address code instruction representation that nanojit receives as input.

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 [Gal et al. 2009]:

1. each conditional branch is associated to a counter initially set to zero.
2. If the interpreter finds a conditional branch during the interpretation of the program, then it increments the counter. The process of checking and incrementing counters is called, in the TraceMonkey nomenclature, the *monitoring phase*.
3. If the counter is two or more, then the trace engine starts translating the bytecodes to LIR instructions, at the same time that the byte codes are interpreted. Overflow tests are inserted into the LIR segment that the trace engine produces. 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.

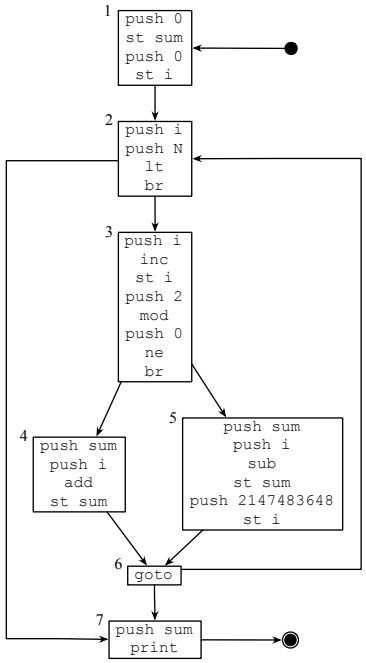
2.1. The Running Example

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

[4] `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);
```

(a)



(b)

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

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.

2.1.1. Trace Construction

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 (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 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, as showed by the dashed arrows in Figure 3 on the right side.

Because the trace that is monitored by the trace engine is not necessarily the trace that is recorded into a LIR segment, we cannot remove overflow tests during the recording phase. That is, the trace engine might record code that has not been seen by the in-

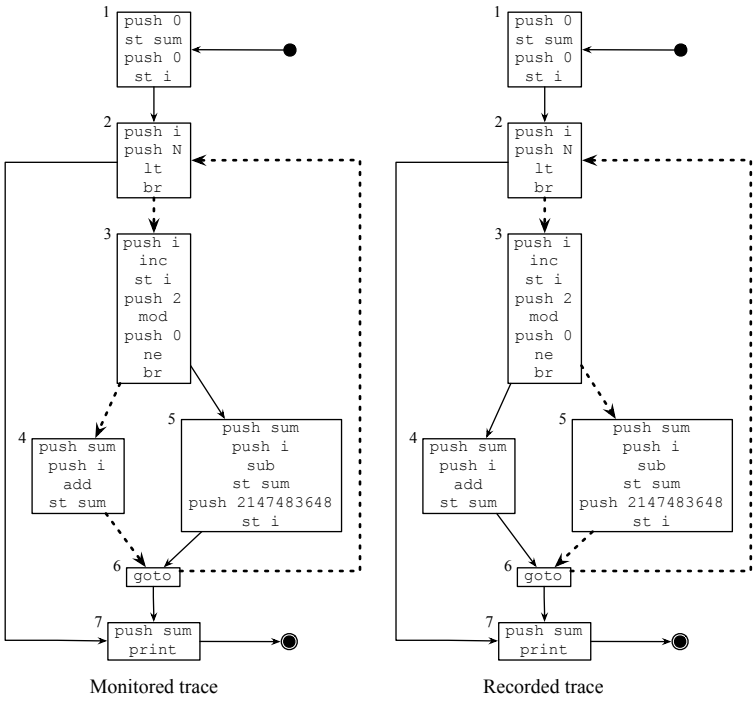


Figure 3. (Left) Segment monitored by the trace engine (Right) Segment effectively recorded by the trace engine.

terpreter, what would yield the information gathered during interpretation useless. Thus, we perform the elimination of overflow tests when nanojit translates LIR into x86 code. Continuing with our example, Figure 4 shows the match between the recorded trace and the LIR segment that the trace engine produces for it.

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 nanojit level.

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. Variable nodes are further divided into *input* nodes and *auxiliary* nodes.

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

Conditional: represent conditional operations, which are used to put bounds on the ranges of the variables. The conditional 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

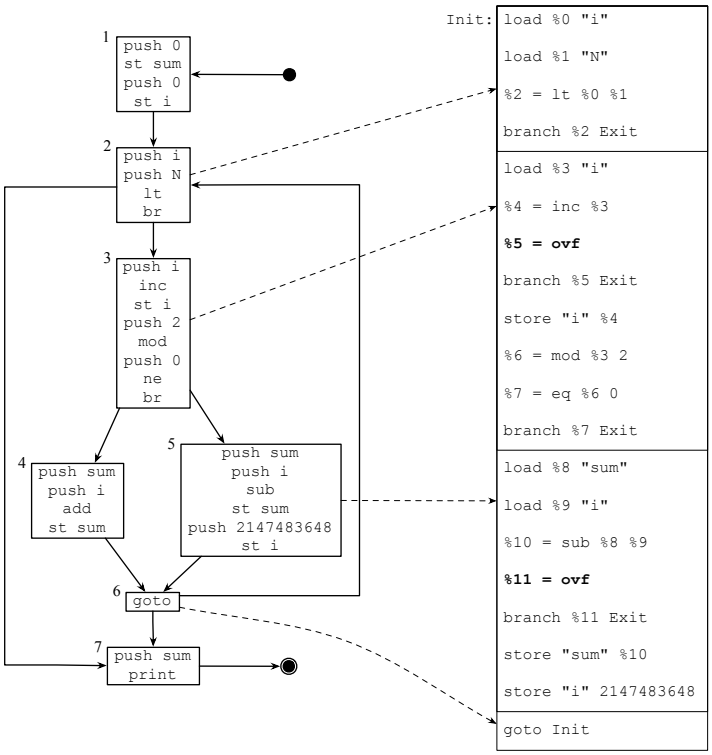


Figure 4. This figure illustrates the match between the recorded trace and the LIR segment that the trace engine produces for it.

(`inc`) and decrement (`dec`). Division operator are not handled once they can produce float point values as result.

The analysis proceeds in two phases: construction of the constraint graph and range propagation. This last phase is further divided in the initialization and the propagation steps. 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) [Bodik et al. 2000] form, which is a superset of the well known SSA form [Cytron et al. 1991]. In the e-SSA representation, a variable is renamed after it is assigned a value, or after it is used in a conditional.

Normally, 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 [Cytron et al. 1991]. 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 interpretation 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 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.
2. For each instruction i that we visit:
   (a) if i is a conditional operation, say $v < u$, we build a conditional 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 the counter of v.

Figure 5 shows the constraint graph to our running example. When constructing the constraint graph, it is important to maintain a list with the order in which each node was created. This list will be later used to guide the range propagation phase.

3.2. Range Propagation

It is during this phase that we find which overflow tests are necessary, and which ones can be safely removed from the target code. This step is further divided into *range initialization* and *range propagation*. In the initialization phase, we simply replace the constraints of the input variables with $[-\infty, +\infty]$ if those variables have been updated inside the trace. We indicate the need of an update by adding back-edges to the constraint graph, from the last occurrence of a modified variable to the corresponding input variable.

After the initialization phase, we remove the back-edges; thus, guaranteeing that our constraint graph is acyclic. The propagation of range intervals happens according to the algorithm given in Figure 6. We visit all the arithmetic and conditional nodes, in

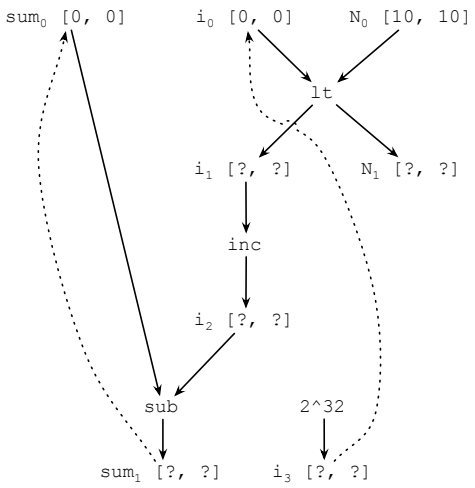


Figure 5. Constraint graph for the trace in Figure 4. The dotted edges mark variables that have been updated inside the trace.

```
class ConstraintGraph() {
  // This queue contains only arithmetic and conditional nodes,
  // inserted in the order in which the instructions that
  // represent them have been found in the trace:
  private Queue<Node> workList;
  // This set holds the variables that have been updated inside
  // the trace:
  private Set<VarNode> targetBackEdges;
  public void constraintPropagation() {
    // Initialization phase:
    for (VarNode v : targetBackEdges) { v.setLimits(-INF, +INF);
    }
    // Propagation phase:
    while (!workList.empty()) {
      Node n = workList.removeFirst();
      n.update();
    }
  }
}
```

Figure 6. The propagation algorithm.

topological order. This ordering is given by the "age" of the node. Nodes that have been created earlier, during the construction of the constraint graph are visited first.

Each arithmetic and conditional node causes the propagation of ranges in a particular way. Figure 7 shows the updating methods that we use for the arithmetic nodes `add` and `inc`, and the conditional node `lt`. Only arithmetic nodes might cause overflows. Thus, while we are propagating range intervals we verify, for each arithmetic node, if the operation that the node encodes might produce an overflow. In the affirmative case we indicate that the node demand an overflow test by calling the function `signalOverflow` in Figure 7.

After the range propagation phase we have a conservative estimate of the intervals that each integer variable might assume during the program execution. This information allows us to go over the LIR segment, before it is passed to nanojit, removing the overflow

```
// Addition: a[la?, ua?] := b[lb, ub] + c[lc, uc]
public void update() {
  // Find the lower boundary:
  la = lb + lc;
  if (la <= 0 && lb > 0 && lb > 0) { la = +INF; signalOverflow();
  } else if (low >= 0 && lb < 0 && lc < 0) {
    la = -INF; signalOverflow();
  }
  // Find the upper boundary:
  int ua = ub + uc;
  if (ua <= 0 && ub > 0 && uc > 0) {
    ua = +INF; signalOverflow();
  } else if (up >= 0 && ub < 0 && uc < 0) {
    ua = -INF; signalOverflow();
  }
}
// Inc Node: a'[la', ua'] := a[la, ua] ++;
public void update() {
  la' = la();
  if (la' + 1 < la) { la' = +INF; signalOverflow();
  }
  ua' = ua;
  if (ua' + 1 < ua) { ua' = +INF; signalOverflow();
  }
}
// Conditional less than:
// (a[la, ua] < b[lb, ub])? => a'[la', ua'], b'[lb', ub']
public void update() {
  ua' = min(ua, lb - 1);
  lb' = max(lb, ua + 1);
}
```

Figure 7. Range propagation for additions, increments and less than's.

tests that our analysis has deemed unnecessary. Thus, we remove the tests associated to each arithmetic node, as long as the function `signalOverflow` has not been called for that node. Figure 8 shows this step: in this case we have been able to remove the overflow from the `inc` operation. But our analysis could not 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 or arithmetic nodes which is proportional to the number of instructions in the trace, and the `update` method is invoked one time for each of these nodes. Notice also that during our analysis we traverse the constraint graph only once, at the range propagation phase. We do not have to preprocess the graph beforehand, in order to sort it topologically, because we get this ordering for free, 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, all of them are variable nodes. 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, given that $E = O(V^2)$. We do not suffer this drawback, because, in our case $E = O(V)$.

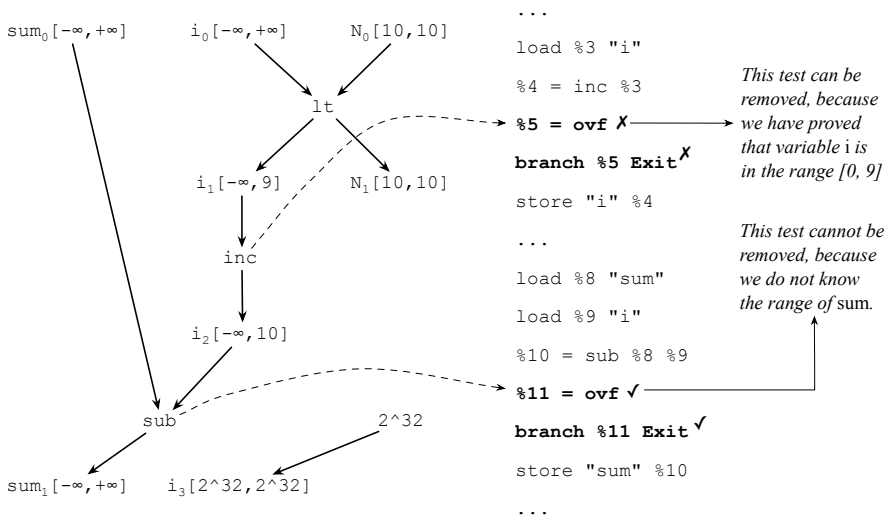


Figure 8. Once we know the ranges of each variable involved in an arithmetic operation we can remove the associated overflow test, if it is redundant.

4. Experimental Results

We have implemented our algorithm on top of TraceMonkey. Our implementation handles the five arithmetic operations described in Section 2.1.1, namely additions, subtractions, increments, decrements and multiplications. Our current implementation performs 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 limitations, which are all due to our lack of understanding of the TraceMonkey's internals, and that we are in the process of overcoming:

- we cannot read 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 used the TraceMonkey test suite to validate our implementation. This testing set contains 253 scripts, which provide a total of 8,469 lines of JavaScript code. TraceMonkey produces 1,792 traces during the interpretation of these scripts. Such traces are translated to x86 binaries by nanojit, yielding a total of 3,958 overflow tests. We can recognize 2,356 out of these tests. The remaining 1,602 tests are due to operations on global variables, which our implementation cannot see.

How effective is our algorithm? Considering the overflow tests that our implementation can recognize, we have deleted 1,281 out of 2,356 tests. Figure 9 is a histogram that shows the effectiveness of our algorithm. In 71% of the scripts we have been able to remove all the possible overflow tests. On the other hand, there are 17% of programs where we do not remove tests. In this case we failed to remove 122 out of the 2,356 possible tests, or 5% of the total. Considering all the 253 programs, we have been able to remove 54% of the total number of tests.

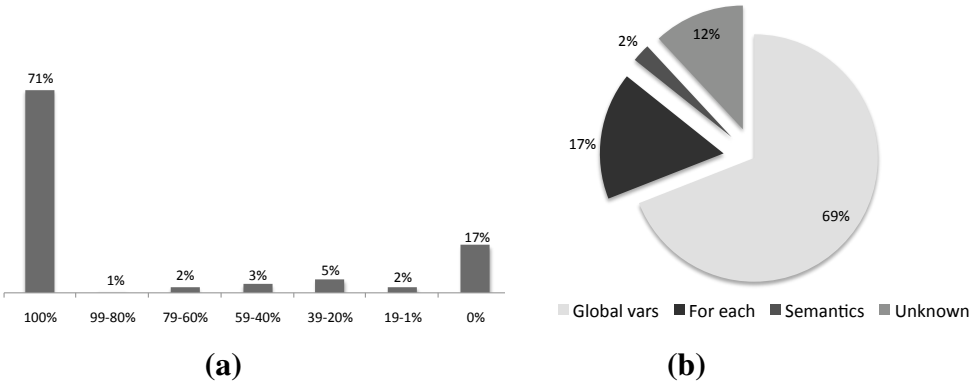


Figure 9. (a) Histogram showing the effectiveness of our analysis. We have been able to remove 100% of all the overflow tests in 71% of the scritps. (b) The reasons that have prevent us to remove some overflow tests.

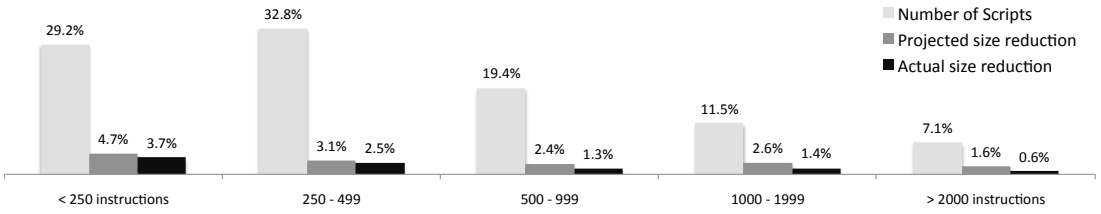


Figure 10. Size reduction due to the elimination of overflow tests.

What is the code size reduction due to the elimination of overflow tests? The elimination of overflow tests reduces the size of the x86 binaries produced by TraceMonkey. TraceMonkey uses six x86 instructions to implement an overflow test: a branch, and five copies to pass information back to the interpreter; all these instructions are removed with the elimination of the test. On the average, our algorithm removes 2.6% of all the instructions. Once we start dealing with global variables, we hope to increase this amount to 3.2%. We have observed that we tend to remove more tests in smaller scripts. Figure 10 illustrates this fact with a histogram, where we show the number of scripts under a certain size, the percentage of the code that is used to implement overflow tests, and the average size reduction that we have produced. Notice that our analysis is more effective in reducing the size of smaller scripts, because larger programs contain less overflow tests in proportion to the total code size.

What is the effect of our algorithm in the running time of TraceMonkey? On the average, our algorithm has increased the running time of TraceMonkey in 6.3%, as the chart in Figure 11 shows. The run-time includes the time to parse and interpret the script, the time to JIT compile it to x86 binaries and execute the binaries. We run these tests on a 2GHz Intel Core 2 Duo, with 2GB of RAM, featuring Mac OS 10.5.8 . Our algorithm increases the time of JIT compilation, but decreases the time of script execution. So far the cost, in time, is negative. This fact happens due to two reasons: (i) the implementation of our algorithm is just a prototype, yet to be optimized for performance. (ii) Most of the scripts run for a very short time, and the effects of avoiding the overflows are negligible.

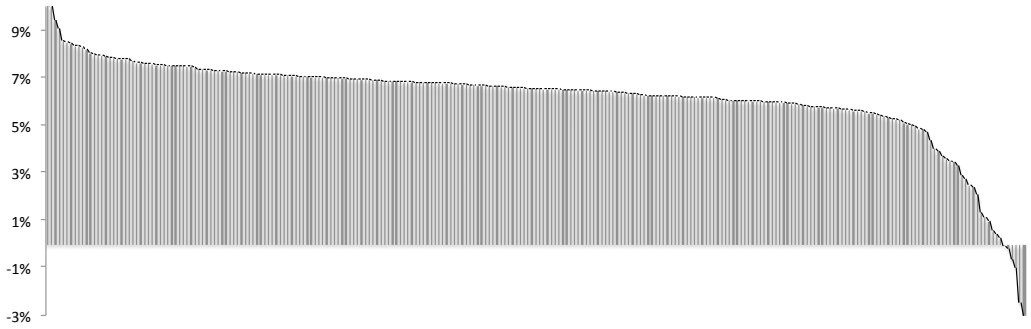


Figure 11. Run-time of TraceMonkey (interpretation + JIT-compilation + script execution). We obtained each number by feeding each script to TraceMonkey 16 times, and removing the smallest and largest outliers. Average 6.3%

Concerning this last point, the only impact of an overflow test, on the script, is the cost of executing a special x86 instruction *branch-if-overflow*. However, this instruction is naturally predicted as not taken adding only 0.5 execution cycles to the program time.

Why sometimes we fail to remove overflows? Each trace contains at least one overflow test. This test is associated to the updating of the induction variable that controls the number of times that the trace executes. We have performed a manual study on each of the 42 programs in which we did not remove overflow tests, focusing on the test on the induction variable. Figure 9 (b) 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.

What we gain by knowing the run-time 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 give us the run-time 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, which corresponds to 37% of the tests that we remove dynamically.

5. Related Work

Just-in-time compilers are old allies of those who advocate interpreted languages. Since the seminal work of John McCarthy [McCarthy 1960], 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 [Aycock 2003].

The optimization that we propose in this paper is a type of partial evaluation at run-time. Partial evaluation is a technique in which a compiler optimizes a program, given a partial knowledge of its input [Jones et al. 1993]. Variations of partial

evaluation have been used to perform general code optimization [Shankar et al. 2005, Schultz et al. 2003]. This particular type of partial evaluation, in which run-time values are used to improve the quality of the code produced by the JIT compiler, is sometimes called *specialization by need* [Rigo 2004]. Examples of running environments that employ this kind of technique include Python's Psyco JIT compiler [Rigo 2004], Matlab [Elphick et al. 2003, Chevalier-Boisvert et al. 2010] and Maple [Carette and Kucera 2007]. Partial evaluation has been used in the context of just-in-time compilation mostly as a form of type specialization. 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. This type of specialization has been recently used in JavaScript [Gal et al. 2009] and Matlab [Chevalier-Boisvert et al. 2010]; however, the technique itself is much older, having seen use, for instance, in the run-time specialization of Self programs [Chambers and Ungar 1989].

Although the concept of just-in-time compilation is old, and well grounded in Computer Science, the compilation of dynamically generated program traces is a new idea. The first trace compiler was described by Andreas Gal in his Ph.D dissertation [Gal 2006, Gal and Franz 2006, Gal et al. 2006]. A detailed description of a trace compiler is given by Chang *et al.* [Chang et al. 2009], and a brief overview of the use of type specialization during trace compilation is given by Gal *et al* [Gal et al. 2009].

Our algorithm to remove overflow tests is a type of *range analysis* [Harrison 1977, Patterson 1995]. Range analysis tries to infer lower and upper bounds to the values that a variable might assume through out the execution of the program. In general the algorithms rely on theorem provers, an approach deemed too slow to a JIT compiler. Bodik *et al.* [Bodik et al. 2000] have described a specialization of range analysis that removes array bound checks, the ABCD algorithm, which is intended to be used by a JIT compiler. Zhendong and Wagner [Su and Wagner 2005] have described a type of range analysis that can be solved in polynomial time. Stephenson *et al.* [Stephenson et al. 2000] 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 run-time value of program variables we can perform a very aggressive, 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, an aggresive variation of range analysis, works in the context of a trace compiler. By relying on the values of variables, an information known only at run-time, our algorithm is able to find very precise ranges for these variables. 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. Currently we are in the process of turning our implementation into an official Firefox patch. In terms of future work, we would like to improve our implementation in terms of performance and effectiveness. For instance, currently we only find the ranges of variables defined locally, a limitation that we are working to overcome. Besides, we

are planing to test our analysis with real scrips, using the 100 sites ranked from alexa.[5]

References

Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113.

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Carette, J. and Kucera, M. (2007). Partial evaluation of maple. In *PEPM*, pages 41–50. ACM.

Chambers, C. and Ungar, D. (1989). Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160.

Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., and Franz, M. (2009). Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, pages 71–80. ACM.

Chevalier-Boisvert, M., Hendren, L. J., and Verbrugge, C. (2010). Optimizing matlab through just-in-time specialization. In *Compiler Construction*, pages 46–65. Springer.

Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. (2009). Staged information flow for Javascript. In *PLDI*, pages 50–62. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *POPL*, pages 25–35.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Elphick, D., Leuschel, M., and Cox, S. (2003). Partial evaluation of matlab. In *GPCE*, pages 344–363. Springer-Verlag New York, Inc.

Gal, A. (2006). *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine.

Gal, A., Eich, B., Shaver, M., Anderson, D., Kaplan, B., Hoare, G., Mandelin, D., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmair, R., Haghighat, M. R., Bebenita, M., Change, M., and Franz, M. (2009). Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM.

Gal, A. and Franz, M. (2006). Incremental dynamic code generation with trace trees. Technical Report 06-16, University of California, Irvine.

Gal, A., Probst, C. W., and Franz, M. (2006). Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE*, pages 144–153.

Harrison, W. H. (1977). Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250.

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

Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1st edition.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of ACM*, 3(4):184–195.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM.

Rigo, A. (2004). Representation-based just-in-time specialization and the psyco prototype for Python. In *PEPM*, pages 15–26. ACM.

Schultz, U. P., Lawall, J. L., and Consel, C. (2003). Automatic program specialization for Java. *TOPLAS*, 25(4):452–499.

Shankar, A., Sastry, S. S., Bodík, R., and Smith, J. E. (2005). Runtime specialization with optimistic heap analysis. *SIGPLAN Not.*, 40(10):327–343.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138.