

Real-world loops are easy to predict : a case study

Raphael E. Rodrigues¹, Péricles R. O. Alves², Fernando M. Q. Pereira³, and Laure Gonnord⁴

1..3 Department of Computer Science, UFMG

6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil

{raphael,periclesrafael,fernando}@dcc.ufmg.br

4 Université Lyon1 & Laboratoire d'Informatique du Parallélisme

(UMR CNRS / ENS Lyon / UCB Lyon 1 / INRIA)

46 allée d'Italie, 69 364 Lyon, France

laure.gonnord@ens-lyon.fr

Abstract

In this paper we study the relevance of fast and simple solutions to compute approximations of the number of iterations of loops (*loop trip count*) of imperative real-world programs. The context of this work is the use of these approximations in compiler optimizations: most of the time, the optimizations yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones.

In this particular work, we argue that, although predicting exactly the trip count of a loop is undecidable, most of the time, there is no need to use computationally expensive state-of-the-art methods to compute (an approximation of) it.

We support our position with an actual case study. We show that a fast predictor can be used to speedup the JavaScript JIT compiler of Firefox - one of the most well-engineered runtime environments in use today.

We have accurately predicted over 85% of all the interval loops found in typical JavaScript benchmarks, and in millions of lines of C code. Furthermore, we have been able to speedup several JavaScript programs by over 5%, reaching 24% of improvement in one benchmark.

1998 ACM Subject Classification D.3.4 [Processors]: Compilers, F.3.2 [Semantics of Programming Languages]: Program Analysis

Keywords and phrases Just-in-time Compilation, Loop Analysis, Trip Count Prediction

1 Introduction

The *Trip Count* of a loop determines how many times this loop iterates during the execution of a program. The problem of estimating this value before the loop executes is important in several ways. In critical real-time systems, a (safe) overapproximation of the actual number of loops is required to compute safe Worst-Case Execution Times (WCETs). Therefore, academia has spent substantial effort in the development of accurate methods to estimate the trip count of loops [1, 4, 7]. These usual techniques rely on expensive deduction systems, typically based on SAT solvers, Linear Programming or costly relational analyses.

In compilers however, there is no need for such precise and costly solutions. But there is still a need for a *trip count* analysis, because loops that tend to run for long time are good candidates for unrolling and automatic parallelization. Thus, walking in the opposite direction, we make a case for a fast trip count predictor in this paper.

The contributions of the paper are thus :

- A simple and easy-to-implement heuristic for approximating the number of loops during the execution of a given program;
- The use of this simple heuristic inside the Mozilla Javascript Just in Time Compiler and the application to speed up the execution of some javascript programs from the literature.

With this case study, we want to argue that :

- Apart from classical applications for termination and safe precise predictions of trip counts, compilers and JIT compilers are potential clients for fast (and sometimes non safe) methods;
- Most of the loops of classical benchmarks are easy to predict;
- There is still a place for much clever heuristics or methods for the remaining loops !

2 Fast Trip Count Prediction

In this section we explain our motivation to only focus on simple loops, coming from previous work on invariant generation. We also give an algorithm to dynamically infer an approximation of the *trip count* of a given loop just before the actual execution of the first loop in the code (thus, dynamically).

2.1 Motivation and inspiration: simple loops invariant generation techniques

Previous work on numerical invariant generation with abstract interpretation techniques [2, 6] led us to ask ourselves how complicated are the loops in actual programs.

In these works, the authors locally use (an adaptation of the so-called) *acceleration techniques* [3] that compute exact fixpoints of (a small class of) numerical transitions in the more general context of abstract interpretation. These *abstract acceleration* techniques have shown their effectiveness in terms of precision at a minimum supplementary cost. The experiments show that in many of the analyzed programs, static analyzers get more precise results just because they are able to precisely deal with loops of the form `for (int i = M; i < N; i++)` (which is an example of an *acceleratable loop* [6]) in the most precise way.

Following these experiments, we decided to implement a light and fast heuristic to dynamically compute an approximation of the number of executions of loop to be executed, whose main specification is to be as precise as possible in the case of such simple loops.

2.2 A fast trip count prediction for “simple loops”

We apply our heuristic dynamically. In other words, we instrument a program - or its interpreter - to estimate the trip count immediately before the first iteration of the loops. Our instrumentation inspects the state of the variables used in the stop condition of each loop.

In the sequel, we only consider perfect loops with a single “interval” exit-condition, *i.e.* loops of the form : `while (e1  $\bowtie$  e2) { some computation }`, where $\bowtie \in \{<, \leq, >, \geq\}$ and `e1` and `e2` are numeric variables. For this kind of loops, we assume that the trip count will be the absolute difference between `e1` and `e2`. For instance, in a loop such as `for (int i = M; i < N; i++)`, we say that its trip count will be $|\text{val}(N) - \text{val}(i)|$ ($\text{val}(x)$ is the runtime value of `x` when the test is performed).

For each loop of this form, we insert a new instruction before the loop, according to Algorithm 1¹. Let us point out that this algorithm only performs a single $O(1)$ operation per loop, without actually looking inside the body of the loop.

However, for loops of the form `for (int i = M; i < N; i=i+s)` (`s` and `N` invariants in the loops), this heuristic gives an overapproximation of the total number of loops, and the most precise result if `s = 1`. There is no guarantee of precision for the general case, of course, but the experiments will show that this simple-blind instrumentation fits our needs in practice. Because our heuristic is so simple, we can execute it quickly. This perfectly suits the needs of a JIT compiler.

¹ Obvious adaptations are necessary to handle $\leq$ and $\geq$.

Algorithm 1 Trip Count Instrumentation Heuristic**Input:** Loop L **Output:** Loop L' with new instructions that estimate its minimum trip count

```

1: if comparison  $e_1 < e_2$  controls loop exit then
2:   Insert instruction  $tripcount = |e_1 - e_2|$  before  $L$ , giving  $L'$ .
3: end if

```

3 Use in Just-in-time compilation, implementation and results**3.1** Hot code detection in JIT compilers

Virtual environments that combine interpretation and compilation face a difficult question: when to invoke the JIT compiler [5]? Premature compilation might produce binaries that do not run long enough to amortize the cost of the JIT transformation. On the other hand, late compilation might delay the optimization of critical parts of the program. As an example, the Firefox browser separates native execution in two parts. After a few rounds of interpretation, the *baseline compiler* translates the program into non-optimized native code. Once this basic native code is deemed hot, it is re-compiled, this time by *IonMonkey*, an optimizing compiler. Figure 1 illustrates this behavior.

The moment when any of these code transformations happens is determined by *thresholds*, which count *discrete events*. A discrete event is either an invocation of a function, or an iteration of a loop [5]. Once a threshold is reached, the execution environment sends that code unit to the JIT compiler. We use our trip count predictor to identify that a threshold will be reached and call the JIT compiler earlier.

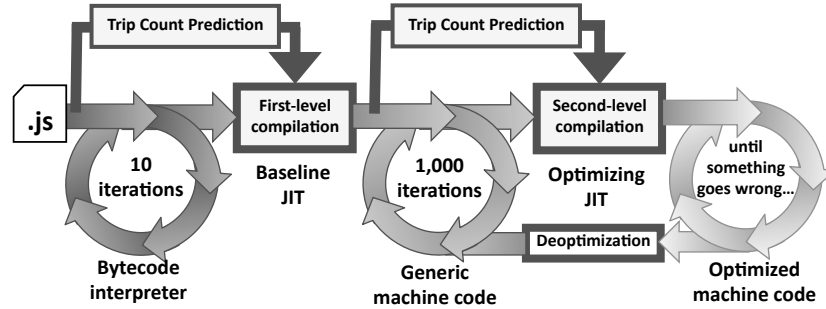


Figure 1 Life cycle of a JavaScript program in our runtime environment. We can perform trip count prediction at two different execution stages.

We have deployed our predictor in the interpreter and in the baseline compiler used in Firefox, as we illustrate in Figure 1. The current distribution of Firefox calls the baseline compiler after 10 events, and the optimizing compiler after 1,000 events. If we predict that a loop will run for more than 10 iterations, we call the baseline compiler immediately, bypassing the warm-up period. Similarly, once in native mode, we call the optimizing compiler immediately upon finding a loop that we estimate to run more than 1,000 times.

3.2 Experiments

We measure the precision of our heuristic by comparing its results against the actual trip count observed during the concrete execution of programs. We have implemented our algorithm both in the LLVM compiler and in Mozilla Firefox. LLVM gives us the opportunity to test our approach in very large programs; Firefox lets us demonstrate its effectiveness in one of the most well-engineered JIT compilers in use today.

Precision. We group results in *interval orders* to measure the precision of our predictor. N denotes the number of iterations of a loop observed via profiling, so the interval $[N, N]$ gives

C programs	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2 * N]$	$]2 * N, N^2]$	$]N^2, +\infty]$
433.milc	14	0	0	435,514,912	38,360	9,984	1,032,930
444.namd	0	0	0	21,602,688	8,065	3,174	0
450.soplex	1,851	367	112	186,939	12,784	10,231	43,338
470.lbm	0	0	0	53,333	0	64	0
401.bzip2	5,270,006	2	311,724	14,386,219	15,987,072	1,502,759	28,939,274
403.gcc	420,390	17	326	17,252,944	1,841,701	283,373	343,054
429.mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
445.gobmk	8,392	20	400	651,081	70,492	117	20,141
456.hmmmer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
458.sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
462.libquantum	0	0	0	8,182,095	0	1	0
464.h264ref	367,010	0	0	302,394,768	12,636,622	5,636,387	10,703,859
473.astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Total	6,171,386	1,113	2,877,982	948,179,441	45,777,639	11,984,924	47,709,177
Total (%)	0.58%	0.00%	0.27%	89.22%	4.31%	1.13%	4.49%

JavaScript	$]1, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N]$	$[N, N]$	$]N, 2N[$	$]2N, N^2[$	$]N^2, \infty[$
SunSpider 1.0	0.0%	0.0%	0.0%	89.2%	2.0%	4.7%	4.1%
V8 v6	0.6%	1.7%	0.0%	94.8%	2.3%	0.0%	0.6%
Kraken 1.1	0.8%	0.0%	3.2%	83.9%	1.6%	2.4%	8.1%

■ **Figure 2** Hit rate of our simple heuristic for C (top) and JavaScript (Bottom).

us the exact predictions. The intervals $[0, \sqrt{N}]$, $]\sqrt{N}, N/2]$, and $]N/2, N[$ represent loops that iterate less times than the prediction. The intervals $]N, 2 * N]$, $]2 * N, N^2]$, and $]N^2, +\infty]$ represent loops that iterate more times than the prediction. The interval $[N, N]$ is marked in gray in Figure 2. The farther from the center column, the worse is the precision. We only produce estimates for interval loops, that account for 71% of the loops in our benchmarks. We collect results for each execution of the loops; hence, the same loop might contribute several times to our final averages.

Figure 2 compares estimated and actual trip counts that we have collected with our profiler. We have correctly predicted 89.2% of the interval loops in the SPEC benchmarks and approximately 90% in the JavaScript benchmarks. To put these results in perspective, we compare them against the numbers presented recently by Tetzlaff and Glesner [8]. These authors also propose a heuristic to estimate the trip count of loops, based on dynamic profiling. Our precision is similar to the one they report. However, we do not need any sort of profiling, multiple compilation phases, nor annotations. Thus, we can be equally as precise, even though we run a much simpler algorithm. Furthermore, because our approach is simpler, it can be used to identify hot spots of programs in both real-world static compilers and JIT compilers.

Speeding up Just-In-Time Compilers. Figure 3 shows the speedup that we obtain using our trip count predictor to perform earlier invocation of the JIT compiler. Each number is the average of 100 runs. We call the baseline compiler immediately once we predict that a loop will iterate 10 times, and we call IonMonkey immediately once we predict that a loop will iterate 1,000 times. Figure 3, reveals that our technique has been able to speed up some benchmarks by a substantial factor. We have also detected slowdowns in a few scripts. This negative behavior happens in benchmarks that iterate for a very short time. In this case, early compilation is not able to pay off the cost of code generation.

4 Conclusion

In this paper, we have presented a heuristic to predict the trip count of loops. We have performed experiments in well-known public benchmarks showing that our technique is able to achieve a good precision, despite of the simplicity of our approach. Our source code is publicly available at <https://code.google.com/p/dynamic-loop-prediction/>

Moreover these experiments show that interval loops represent 70% of our benchmarks and a simple heuristic is the most precise in 90% of these interval loops. These results advocate the use of simple preprocessing for proving the termination (or counting the number of loops) of real-world programs that may include proper slicing and simple pattern-matching.

SunSpider 1.0	
3d-cube	-1%
3d-morph	-1%
3d-raytrace	1%
access-binary-trees	0%
access-fannkuch	2%
access-nbody	-1%
access-nsieve	2%
bitops-3bit-in-byte	0%
bitops-bits-in-byte	1%
bitops-bitwise-and	3%
bitops-nsieve-bits	3%
ctrflow-recursive	-1%
crypto-aes	0%
crypto-md5	3%
crypto-sha1	10%
date-format-tofte	2%
date-format-xparb	2%

SunSpider 1.0	
math-cordic	2%
math-partial-sums	-1%
math-spectral-norm	-1%
regex-dna	0%
string-base64	24%
string-fasta	-7%
string-tagcloud	1%
string-unpack-code	0%
string-validate-input	-5%

V8 version 6.0	
crypto	0%
deltablue	2%
earley-boyer	0%
raytrace	0%
regex	3%
richards	-1%
splay	1%

Kraken 1.1	
ai-astar	1%
audio-beat-detect	0%
audio-dft	-4%
audio-fft	0%
audio-oscillator	1%
img-gaussian-blur	-3%
imaging-darkroom	-3%
imaging-desaturate	0%
json-parse-financial	1%
json-stringify-tinder	1%
crypto-aes	6%
crypto-ccm	2%
crypto-pbkdf2	7%
crypto-sha256-itrv	6%

■ **Figure 3** Speedup due to trip count predictor for benchmarks distributed with Firefox.

Future work may include the extraction of the remaining challenging loops for the community because there is still an open space to be explored.

References

- 1 C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, pages 117–133. Springer, 2010.
- 2 C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3 – 16, 2010.
- 3 S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.
- 4 G. Bhargav and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384. Springer, 2008.
- 5 E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS*, pages 202–211. ACM, 2000.
- 6 L. Gonnord and P. Schrammel. Abstract Acceleration in Linear Relation Analysis. *Science of Computer Programming, under press*, 2013.
- 7 J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, pages 57–66. IEEE, 2006.
- 8 Dirk Tetzlaff and Sabine Glesner. Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 300–307. IEEE, 2013.