Information Processing Letters 115 (2015) 146–150

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A case for a fast trip count predictor

Péricles R.O. Alves, Raphael E. Rodrigues, Rafael Martins de Sousa, Fernando Magno Quintão Pereira *

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

ARTICLE INFO

Article history:
Received 16 April 2014
Received in revised form 7 July 2014
Accepted 19 August 2014
Available online 6 September 2014
Communicated by J.L. Fiadeiro

Keywords:
Compilers
Programming languages
JIT Compilation
Loop Analysis
Trip Count Prediction

ABSTRACT

The *Trip Count* of a loop determines how many iterations this loop performs. Predicting this value is important for several compiler optimizations, which yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones. However, finding an exact prediction, in general, is an undecidable problem. Such problems are usually approached via methods which tend to be computationally expensive. In this paper we make a case for a cheap trip count prediction heuristic, which is $O(1)$ on the size of the loop. We argue that our technique is useful to just-in-time compilers. If we predict that a loop will iterate for a long time, then we invoke the JIT compiler earlier. Even though straightforward, our observation is novel. We show how this idea speeds up JavaScript programs, by implementing it in Mozilla Firefox. We can apply our heuristic in 79.9% of the loops found in typical JavaScript benchmarks. For these loops, we obtain exact predictions in 91% of cases. We get similar results when analyzing the C programs of SPEC CPU 2006. A more elaborate technique, linear on the size of the loop, improves our $O(1)$ technique only marginally. As a consequence of this work, we have been able to speed up several JavaScript programs by over 5%, reaching 24% of improvement in one benchmark.

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. For instance, loops that tend to run for long time are good candidates for unrolling and vectorization. Therefore, the academia has spent substantial effort in the development of accurate methods to estimate the trip count of loops [1–3,7–9].

Trip count prediction is usually approached via expensive deduction systems, typically based on SAT solvers [7], ranking functions [3] or recurrence relations [2,10]. Walking in the opposite direction, this paper makes a case for a fast trip count predictor. We instrument a program – or its interpreter – to estimate the trip count of loops immediately before these loops are visited by the execution flow. Our heuristic checks the stop condition of a loop, inspecting the current state of the variables used in that conditional. As an example, an iterator such as `for(int i = M; i < N; i++)` is guarded by the stop condition `i < N`. We assume that the difference `val(N) − val(i)` gives us the trip count of the loop, where `val(x)` is the runtime value of `x` when the test is performed. This trip count predictor is so simple that its effectiveness and accuracy can be regarded as compiler's folklore.

The goal of this paper is to support this intuition with concrete evidence demonstrating that our $O(1)$ heuristic is useful and precise. To establish the first point, we have used it in the JavaScript engine of Mozilla Firefox. If our heuristic predicts a large trip count for a loop, then

* Corresponding author.
E-mail addresses: periclesrafael@dcc.ufmg.br (P.R.O. Alves), raphael@dcc.ufmg.br (R.E. Rodrigues), rafaelms@dcc.ufmg.br (R.M. de Sousa), fernando@dcc.ufmg.br (F.M. Quintão Pereira).

http://dx.doi.org/10.1016/j.ipl.2014.08.008
0020-0190/

Algorithm 1 Trip Count Instrumentation Heuristic.

Input: Loop L
Output: Loop L' with new instructions that estimate its maximum trip count
1: **if** L has stop condition "$v_1 < v_2$", **where** v_1 and v_2 are variables in registers **then**
2: Insert statement "$tripcount = |v_1 - v_2|$" in the pre-header of L, giving L'.
3: **end if**

Algorithm 2 Trip Count Instrumentation Based on Colliding Vectors.

Input: Loop L
Output: Instrumented Loop L' with new instructions that estimate its maximum trip count
1: **if** If L has stop condition "$v_1 < v_2$", **where** v_1 and v_2 are variables in registers **then**
2: $s_1 t + i_1 = indVar(v_1)$, $s_2 t + i_2 = indVar(v_2)$,
3: **where** s_j is the maximum step and i_j is initial value of basic induction variable v_j, $j \in \{1, 2\}$.
4: **if** $\exists$ *step* **then**
5: Insert statement "$tripcount = |(i_2 - i_1)/(s_1 - s_2)|$" in the pre-header of L, giving L'.
6: **end if**
7: **end if**

we call the JIT compiler before its warm-up period. This early compilation lets us vary the execution time of public benchmarks from −7% up to 24%, as we will discuss in Section 4. Even though this technique is straightforward, we believe that we are the first group to test it in an industrial-strength virtual machine.

Our heuristic is surprisingly accurate, given its simplicity. We can apply it in 79.9% of the loops found in typical JavaScript benchmarks, and in 67.0% of the loops found in SPEC CPU 2006. We call these structures *interval loops*. We have predicted exactly the bounds of 91.1% of the JavaScript interval loops, and of 89.2% of SPEC's interval loops. We have compared our approach against a heavier heuristic, which demands a holistic view of the loops. The gains that we get with the extra complexity are negligible.

Related works This paper touches two fields in computer science: complexity analysis and just-in-time compilation. Most complexities analyses are not heuristics, but – incomplete – proof techniques, whose result is true, if there is one. For the reader interested in knowing more about the state-of-the-art approaches in complexity analysis, we recommend the Related Works section of Brockschmidt et al.'s recent paper [3]. The cheapest technique that we are aware of is due to Blanc et al. [2]. Blanc et al.'s method is similar to our Algorithm 2, when applied on chains of nested loops. None of these approaches have been tested in the context of a Just-In-Time compiler before. Namjoshi and Kulkarni [11] have demonstrated via simulation that it is possible to speed up a JIT compiler via loop prediction; however, they have not implemented any particular heuristic in a virtual machine. Popular runtime environments, such as Java Hotspot, Google V8, PyPy or Mozilla's IonMonkey have never used loop prediction. An exception is LuaJIT, which checks iterator bounds in a specific high-level construct, e.g., `for i=start,stop,step do`. LuaJIT does it to avoid compiling a loop with a very low remaining iteration count.

2. Fast trip count prediction

The *stop condition* of a loop is a boolean test that, when true, forces the program flow to exit the loop. We say that a loop is an *interval loop* if its stop condition is a comparison of two variables v_1 and v_2 using an inequality ($<$, $\leq$, $>$, or $\geq$). The variable i is a *basic induction variable* of a loop if the only definitions of i within the loop are of the form $i = i + s$ or $i = i - s$, and s is loop invariant. We say that s is the *step* of i. A loop is *single-exit* if it has only one stop condition. We call a single-exit interval loop *trivial* if its stop condition is like "i *op* N", where (i) $op \in \{<, \leq, >, \geq\}$; (ii) i is a basic induction variable; (iii) the step of i is 1; and (iv) N is loop invariant. As an example, `for (int i = M; i < N; i++)` is a trivial interval loop, with stop condition `i < N`, and basic induction variable `i`.

Because trivial loops are common in practice; this paper studies the effectiveness of an $O(1)$ trip count predictor that works **exactly** for them. We assume that the trip count of a loop is the absolute difference between the variables used in its stop condition. For instance, considering our previous loop example, we assume that its trip count is $|\texttt{val(N)} - \texttt{val(i)}|$, where $\texttt{val}(v)$ is the runtime value of variable v when the loop is first visited by the program flow. We generate the commands to perform this subtraction statically, upon compiling the program, but the actual estimation happens, naturally, at runtime. Algorithm 1 shows the code generation for conditions such as $v_1 < v_2$. Obvious adaptations are necessary to handle $>$, $\leq$ and $\geq$.

Because our heuristic is so uninvolved, it can be implemented quickly. Our code generation does not require any global analysis of the loop. We do not try to infer the step of the induction variable, neither we try to find out which limits of the stop condition are invariant. We simply perform the subtraction of limits. Thus, our code generator runs in $O(1)$ per loop. As we will explain shortly, it suits perfectly the needs of a just-in-time compiler.

A more elaborate analysis Many loops found in real-world applications are trivial, yet, there are several others which are not. To demonstrate that our heuristic is still precise even in more complex cases, we compare it against a more elaborate one. This new heuristic treats induction variables used in loop conditions as rectilinear vectors such as $s \times t + i$, where s is the step and i the initial value of the induction variable. The trip count t of the loop is the time when these vectors collide. Algorithm 2 describes our second heuristic.

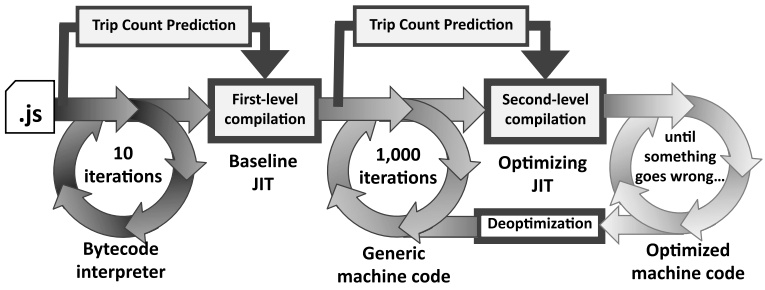

Fig. 1. Life cycle of a JavaScript program in our runtime environment.

Category	%I	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2*N]$	$]2*N, N^2]$	$]N^2, +\infty]$
SPEC	67	6,171,386	1113	2,877,982	948,179,441	45,777,639	11,984,924	47,709,177
SPEC (%)		0.58%	0.00%	0.27%	89.22%	4.31%	1.13%	4.49%
SunSpider 1.0	81	0.0%	0.0%	0.0%	89.2%	2.0%	4.7%	4.1%
V8 v6	77	0.6%	1.7%	0.0%	94.8%	2.3%	0.0%	0.6%
Kraken 1.1	80	0.8%	0.0%	3.2%	83.9%	1.6%	2.4%	8.1%

Fig. 2. Hit rate of our predictor (Algorithm 1). %I: percentage of interval loops.

The function *indVar* finds the initial value (s), and the maximum increment (i) over all paths through which an induction variable can be modified. We use the maximum step because, as we will discuss in Section 3, we want to avoid the fee of invoking the JIT in a situation that does not pay off. This function is $O(N)$, where N is the number of instructions in the loop. We can compute i via Dijkstra's algorithm, which solves the single-source shortest path problem for a graph with non-negative costs assigned to edges [4]. We apply the algorithm on the program's dependence graph [6]. In our case, the cost of each edge is the increment that the induction variable can suffer along that path. If we find paths with negative and positive weights, then we have a *non-monotonic* induction variable. In this case, we assume that its step is 1 to perform a prediction. Except for the use of Dijkstra's, Algorithm 2 is similar to Blanc's ABC test applied on an outermost loop [2].

3. Use in just-in-time compilation

Virtual machines 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. However, 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 loop is interpreted 10 times, the program is compiled to non-optimized native code, as we illustrate in Fig. 1. This first translator is called the *baseline compiler*. If a loop in this program iterates 1000 times, it is re-compiled, this time by *IonMonkey*, an optimizing compiler. The program might be re-compiled again, if "something goes wrong", e.g., some runtime speculation, such as type inference, happens to fail.

We want to shorten these threshold, e.g., 10 or 1000 iterations in the case of Firefox, for code that our dynamic trip count predictor considers hot. We have deployed our predictor in the interpreter and in the baseline compiler used in Firefox v28.0, as we illustrate in Fig. 1. Our heuristic allows us to bypass the warmup periods. If we predict that a loop will run for more than 10 iterations, we call the baseline compiler immediately. Similarly, once in native mode, we call the optimizing compiler immediately upon finding a loop that we estimate to run more than 1000 times. Because our heuristic is so simple, its overhead is minimum. Thus, we can insert code for it as soon as the original Firefox interpreter or JIT compiler identifies loops.

4. 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. Contrary to Firefox, LLVM gives us the opportunity to test our approach in very large programs; Firefox let us demonstrate its effectiveness in one of the most well-engineered just-in-time compilers in use today.

Precision To measure the precision of our predictor we have grouped results in *interval orders*. The interval order of a prediction tells us how far from the actual result that prediction is. If N denotes the number of iterations of a loop observed via profiling, then the interval $[N, N]$ gives us the percentage of exact predictions. Some intervals, such as $]N/2, N]$, represent predictions that are less than the actual result. For instance, if we guess that a loop will iterate 10 times, but it iterates 16, then we classify the prediction in the interval $]N/2, N]$, because $N/2 = 8 < 10 \leq 16 = N$. Intervals such as $]N, 2N]$ contain predictions that are greater than the observed number of iterations. As an example, if we predict that a loop will iterate 10 times, but we observe only 8 iterations, then we are in this category, given that $N = 8 \leq 10 \leq 16 = 2N$. Fig. 2 shows the comparison between the estimated trip count and the actual trip count that we have collected with our profiler. While running the programs, each time a loop stops, we collect the actual trip count and compare it with the estimated trip count. Thus, the numbers that we present are the numbers of dynamic *instances* of loops.

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%
ctrlflow-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%
regexp-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%
regexp	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%

Fig. 3. Speedup due to trip count predictor for benchmarks distributed with Firefox.

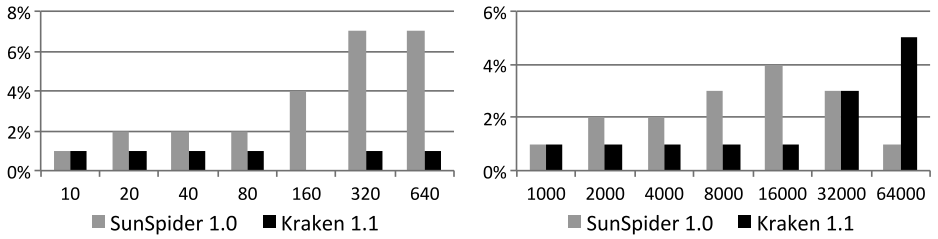

Fig. 4. Speedup obtained by our trip count predictor when IonMonkey uses different thresholds. The baseline compiler is called after 10 loop iterations or function invocations.

Fig. 2 tells us that we correctly predicted 89.2% of the interval loops of SPEC. We get similar results for the JavaScript benchmarks. We compare these results against those produced by Algorithm 2, for the C programs. Vectors allowed us to predict correctly 90.0% of the SPEC loops visited – an improvement of 0.8% in relation to our simple test. We believe that this extra-precision is not worth the linear complexity of the more elaborate test, at least in the world of JIT compilers.

Speeding up just-in-time compilers Fig. 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 1000 times. These compilers are still invoked due to the original threshold policy, i.e., when a function is invoked many times, or when a loop – which we have predicted as cold – ends up iterating many times. Fig. 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. In particular, for `string-fasta`, our early invocation prevents IonMonkey from inlining a specific function, `rand`. Once IonMonkey's profiler concludes that the function must be inlined, recompilation takes place.

Fig. 4 shows how the runtime of programs varies as we change the invocation threshold. We show results for the benchmark suite that runs for less time, SunSpider, and for Kraken, the suite that runs for the longest time. In Fig. 4(a), we vary the thresholds used to invoke the baseline compiler, from 10 to 640 loop iterations, and keep IonMonkey's threshold fixed in 1000 iterations. Fig. 4(b) shows a similar experiment: we fix the threshold used by the baseline compiler in 10 events, and vary the threshold used by IonMonkey.

The figure lets us draw three conclusions. First, our gains become better once we increase the invocation threshold. This happens because, without the predictor, the virtual machine spends more time running code in slow modes. Second, programs having loops with lower iteration counts benefit more from earlier invocation. If a program will run for too long, our predictor ends up speeding a few rounds, e.g., 990, of a loop that will iterate for thousands of times. Thus, we observe larger gains in SunSpider than in Kraken because SunSpider's loops have smaller trip counts. Notice, however, that for very high thresholds, neither trip count estimation, nor traditional warm up, will trigger compilation. That explains why, after 16,000 events in Fig. 4(b), the optimizing compiler is no longer called for many of the SunSpider's scripts, and our predictor tends to deliver the same speedup as traditional profiling. Third, trip count prediction gives the JIT compiler the chance to use longer invocation thresholds. We have observed gains of 5% in Kraken, once we wait 64K iterations to invoke the JIT compiler.

5. 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 the simplicity of our approach. With this work, we have demonstrated that there is still an open space to be explored involving the design and implementation of fast and reliable trip count predictors for just-in-time compilers.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, Damiano Zanardini, Cost analysis of object-oriented bytecode programs, Theor. Comput. Sci. 413 (1) (2012) 142–159.

[2] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, Laura Kovács, ABC: Algebraic bound computation for loops, in: LPAR, Springer, 2010, pp. 103–118.

[3] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, Jürgen Giesl, Alternating runtime and size complexity analysis of integer programs, in: TACAS, Springer, 2014, pp. 140–155.

[4] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.

[5] E. Duesterwald, V. Bala, Software profiling for hot path prediction: less is more, in: ASPLOS, ACM, 2000, pp. 202–211.

[6] J. Ferrante, K. Ottenstein, J. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.

[7] Sumit Gulwani, Krishna K. Mehra, Trishul Chilimbi, Speed: precise and efficient static estimation of program computational complexity, ACM SIGPLAN Not. 44 (1) (2009) 127–139.

[8] J. Gustafsson, A. Ermedahl, C. Sandberg, B. Lisper, Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution, in: RTSS, IEEE, 2006, pp. 57–66.

[9] N. Halbwachs, Y. Proy, P. Roumanoff, Verification of real-time systems using linear relation analysis, Form. Methods Syst. Des. 11 (2) (1997) 157–185.

[10] Jens Knoop, Laura Kovács, Jakob Zwirchmayr, Symbolic loop bound computation for WCET analysis, in: PSI, Springer, 2012, pp. 227–242.

[11] Manjiri A. Namjoshi, Prasad A. Kulkarni, Novel online profiling for virtual machines, in: VEE, ACM, 2010, pp. 133–144.