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

A R T I C L E I N F O

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

A B S T R A C T

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/© 2014 Elsevier B.V. All rights reserved.

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.