Computer Languages, Systems & Structures 39 (2013) 123–141

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Context-sensitive trace inlining for Java ☆

Christian Häubl [a,*], Christian Wimmer [b], Hanspeter Mössenböck [a]

[a] Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Altenbergerstrasse 69, 4040 Linz, Austria
[b] Oracle Labs, 500 Oracle Parkway, Redwood Shores, CA 94065, USA

ARTICLE INFO

Available online 19 April 2013

Keywords:
Java
Just-in-time
Trace-based
Compilation
Inlining

ABSTRACT

Method inlining is one of the most important optimizations in method-based just-in-time (JIT) compilers. It widens the compilation scope and therefore allows optimizing multiple methods as a whole, which increases the performance. However, if method inlining is used too frequently, the compilation time increases and too much machine code is generated. This has negative effects on the performance.

Trace-based JIT compilers only compile frequently executed paths, so-called traces, instead of whole methods. This may result in faster compilation, less generated machine code, and better optimized machine code. In the previous work, we implemented a trace recording infrastructure and a trace-based compiler for Java™, by modifying the Java HotSpot VM. Based on this work, we evaluate the effect of trace inlining on the performance and the amount of generated machine code.

Trace inlining has several major advantages when compared to method inlining. First, trace inlining is more selective than method inlining, because only frequently executed paths are inlined. Second, the recorded traces may capture information about virtual calls, which simplify inlining. A third advantage is that trace information is context sensitive so that different method parts can be inlined depending on the specific call site. These advantages allow more aggressive inlining while the amount of generated machine code is still reasonable.

We evaluate several inlining heuristics on the benchmark suites DaCapo 9.12 Bach, SPECjbb2005, and SPECjvm2008 and show that our trace-based compiler achieves an up to 51% higher peak performance than the method-based Java HotSpot client compiler. Furthermore, we show that the large compilation scope of our trace-based compiler has a positive effect on other compiler optimizations such as constant folding or null check elimination.

1. Introduction

Method-based just-in-time (JIT) compilation translates whole methods to optimized machine code, while trace-based compilation uses frequently executed paths, so-called traces, as the compilation unit [1]. This can increase the peak performance, while reducing the amount of generated machine code. Fig. 1 shows the control flow graphs (CFGs) of three methods as well as three possible traces through them. The start of a trace is called a *trace anchor*, which is block 1 for all

* Corresponding author. Tel.: +43 732 2468 4341; fax: +43 732 2468 4345.
E-mail addresses: haeubl@ssw.jku.at (C. Häubl), christian.wimmer@oracle.com (C. Wimmer), moessenboeck@ssw.jku.at (H. Mössenböck).
URLS: http://www.ssw.jku.at/ (C. Häubl), http://www.christianwimmer.at/ (C. Wimmer), http://www.ssw.jku.at/ (H. Mössenböck).

1477-8424/$ - see front matter © 2013 The Authors. Published by Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cl.2013.04.002

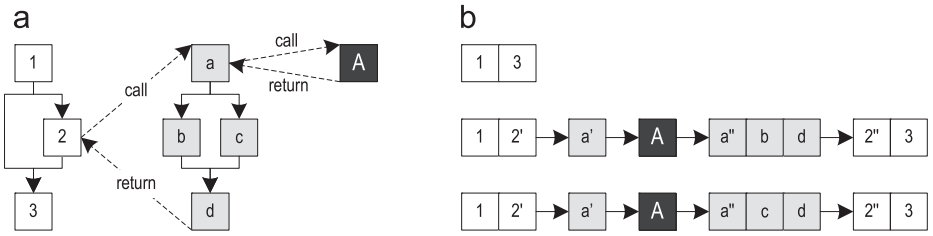

Fig. 1. Possible traces through three methods: (a) control flow graphs and (b) possible traces.

traces in the example. It highly depends on the specific trace recording implementation which blocks are chosen as trace anchors.

In a virtual machine (VM), traces can be recorded by instrumenting bytecode execution. Those traces are then compiled to optimized machine code. If a method part that was not compiled has to be executed, it is common to fall back to the interpreter.

Most existing trace recording implementations allow traces to cross method boundaries [1,2,10,12,18]. This may result in large traces that must be compiled together.

In the previous work [14,15], we implemented a trace-based JIT compiler based on Oracle's Java™ HotSpot client compiler [19]. Our earlier conference paper [15] focused on trace inlining and contributed the following:

- We described how to perform trace inlining and discuss its advantages compared to method inlining.
- We presented multiple trace inlining heuristics implemented for our trace-based JIT compiler.
- We evaluated the impact of our trace inlining heuristics on compilation time, peak performance, and amount of generated machine code for the DaCapo 9.12 Bach [3] benchmark suite.

This paper is an extended version of our earlier conference paper [15], and contributes the following new aspects:

- We present our trace recording and our trace inlining approaches in more detail.
- We describe how compiler intrinsics for native methods can profit from the larger compilation scope that is achieved by our trace inlining.
- We additionally evaluate our inlining heuristics on the benchmark suites SPECjbb2005 [23] and SPECjvm2008 [24]. Furthermore, we also compare the peak performance of our best trace inlining heuristic to the Java HotSpot server compiler.
- We evaluate which high-level compiler optimizations do benefit from trace inlining due to the widened compilation scope.

The remaining paper is organized as follows: Section 2 gives a short overview of our trace-based Java HotSpot VM. In Section 3 we illustrate our trace recording system, and in Section 4 we explain how we perform trace inlining. Section 5 presents different trace inlining heuristics. Section 6 discusses the benchmark results. In Section 7 we discuss related work, and Section 8 concludes the paper.

2. Overview

In the previous work, we implemented a trace recording infrastructure and a trace-based JIT compiler for Java [14,15]. Fig. 2 shows the structure of our VM. Execution starts with the class loader that loads, parses, and verifies the class files. The class loader provides run-time data structures such as the constant pool and method objects to other parts of the VM. After class loading, a bytecode preprocessing step is performed that detects loops and creates tracing-specific data structures.

For trace recording, the Java HotSpot VM template interpreter [13] is duplicated and instrumented. This results in a normal and a trace recording interpreter. The normal interpreter executes bytecodes with nearly the same speed as the interpreter of the unmodified VM and is used for the initial executions. Whenever the normal interpreter encounters a trace anchor, it increments the invocation counter of that trace anchor. When the counter overflows, the trace anchor is marked as hot and execution switches to the trace recording interpreter. The current implementation supports two different kinds of traces: loop traces anchored at loop headers, and method traces anchored at method entries.

Oracle's Java HotSpot VM ships with two different JIT compilers that share most parts of the VM infrastructure. The *client compiler* is designed for startup performance and implements basic optimizations to achieve a decent peak performance [19]. Upon compilation, the compiler generates the high-level intermediate representation (HIR), which is in static single assignment (SSA) form [7] and represents the control flow graph. During and after building the HIR, optimizations such as constant folding, null check elimination, and method inlining are applied. The optimized HIR is translated to the low-level intermediate representation (LIR), which is close to machine code but still mainly platform independent. The LIR is then used for linear scan register allocation [27] and code generation.

The *server compiler* performs significantly more optimizations than the client compiler and produces highly efficient code to reach the best possible peak performance [21]. It is designed for long-running server applications where the initial JIT