Combining Offline and Online Optimizations: Register Allocation and Method Inlining

Hiroshi Yamauchi and Jan Vitek

Department of Computer Sciences, Purdue University
{yamauchi, jv}@cs.purdue.edu

Abstract. Fast dynamic compilers trade code quality for short compilation time in order to balance application performance and startup time. This paper investigates the interplay of two of the most effective optimizations, register allocation and method inlining for such compilers. We present a bytecode representation which supports offline global register allocation, is suitable for fast code generation and verification, and yet is backward compatible with standard Java bytecode.

1 Introduction

Programming environments that support dynamic loading of platform-independent code must provide supports for efficient execution and find a good balance between responsiveness (shorter delays due to compilation) and performance (optimized compilation). Thus, most commercial Java Virtual Machines (JVM) include several execution engines. Typically, there is an interpreter or a *fast compiler* for initial executions of all code, and a profile-guided optimizing compiler for performance-critical code.

Improving the quality of the code of a fast compiler has the following benefits. It raises the performance of short-running and medium length applications that exit before the expensive optimizing compiler fully kicks in. It also benefits long-running applications with improved startup performance and responsiveness (due to less eager optimizing compilation). One way to achieve this is to shift some of the burden to an offline compiler. The main question is what optimizations are profitable when performed offline and are either guaranteed to be safe or can be easily validated online.

We investigate the combination of offline analysis with online optimizations for the two most important Java optimizations [12]: register allocation and method inlining, targeted for a fast compiler. The first challenge we are faced with is the choice of intermediate representation (IR). Java bytecode was designed for compactness, portability, and verifiability and not for encoding offline program optimizations. We build on the previous work [16,18,17,2,9,11,10] and propose a simplified form of the Java bytecode augmented with annotations that support offline register allocation in an architecture independent way. We call it SimpleIR (or SIR). A SIR program is valid Java bytecode and can thus be verified and used in any JVM. We then evaluate offline register allocation heuristics [2,10]

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 307–322, 2006.

and propose novel heuristics. Another challenge is that performing method inlining offline is not always effective because of separate compilation (e.g., dynamic class loading over network and dynamic bytecode generation), architecture independence (e.g., platform-dependent (standard) library modules), and access restriction (e.g., inter-class inlining of methods that access private fields). Thus, we ask the question: can we combine offline register allocation with online method inlining? The contributions of this paper are as follows:

- **Backward-compatible IR for offline register allocation:** *We propose a simplified form of Java bytecode with annotations which supports encoding offline register allocation, fast code generation and verification, and backward compatibility.*
- **Evaluation of offline register allocation heuristics:** *We directly compare two previously known register allocation heuristics and two new heuristics.*
- **Register allocation merging technique:** *which quickly and effectively computes register allocation for inlined methods based on offline register allocation for individual methods.*
- **Empirical evaluation:** *We have implemented our techniques in a compiler and report on performance results and compilation times for different scenarios.*

2 Intermediate Representations

Alternative intermediate code representations have been explored in the literature. They can be categorized into three groups according to their level of abstraction and conceptual distance from the original format. The first category is annotated bytecode using the existing features of Java bytecode format. This approach is backward compatibile as any JVM can run the code by simply ignoring the annotations. The work of Krintz et al. [11], Azevedo et al. [2], and Pominville et al. [14] are some examples. The second category can be described as optimization-oriented high-level representations. These representations do not necessarily bear any resemblance to Java bytecodes. An example is SafeTSA [1] which is a type safe static single assignment based representation. The last category is that of fully optimized low-level architecture dependent representations with certain safety annotations, such as the typed assembly language (TAL) [13].

2.1 An IR for Offline Register Allocation

We propose an IR for offline register allocation which is a simplified form of the Java bytecode (called SIR). We motivate our design choices and contrast them with previous results.

Backward compatibility with Java. SIR is a subset of Java bytecode and thus backwards compatibile. This is important: Any JVM can run SIR code with the expected semantics. Existing tools can be used to analyze, compile, and transform SIR code. This is in contrast to [1,18] which proposes an incompatible register-based bytecode or a SSA form. Offline register allocation results are encoded in annotations following [2,9,10,11].