Simple, Effective Code-Size Reduction for Functional Programs

Ekaterina Stefanov and Anthony M. Sloane

Department of Computing, Macquarie University
{kate, asloane}@ics.mq.edu.au

Abstract. Code-size reduction is an important area of investigation for computer system developers due to the increasing use of technologies such as communication networks and embedded systems for which program size is an important factor. A new software-based method of program compression for languages with interpreter-based runtime systems is described. The method employs a modified version of a standard dictionary-based text compression algorithm to produce a shorter encoding for a given program and a runtime system tailored for it. In comparison with previous software-based code-size reduction methods the new method is simpler to implement and imposes lower overhead at compile time. Its performance on a representative suite of Haskell programs is analysed. Executable space savings of 16–26% are achieved as a result of code compression, exclusion of unused instructions from the runtime, and inclusion of the standard library in the optimisation. To the best of the authors’ knowledge, this is the first work on running compressed code for a purely declarative and functional language.

1 Introduction

The size of computer programs is important in many modern computing environments. For example, communication networks are often used to transport programs for execution on remote platforms such as Web browsers or mobile phones. Cost and latency considerations demand that developers trade off program functionality (and hence size) with communication bandwidth. Similarly, embedded computer systems are increasingly part of our day-to-day lives. The cost of embedded systems is closely related to the amount of code and data that must be included.

These sorts of concerns have lead to a growing interest in code-size reduction methods that can be applied automatically to any program. Thus a developer can gain the benefits of reduced program size without having to artificially limit the functionality of their program as much.

Code-size reduction methods based on hardware modifications have arguably the most potential for success. However, they are also the most costly to implement, at least until such hardware becomes available in most machines. Hence, we focus on software-based approaches in this paper.

© Springer-Verlag Berlin Heidelberg 2005
Our method targets the embedded systems and small device application areas, where traditionally the language of choice has been C due to its ability to produce small fast executables. Alternatives to C for these platforms are desirable due to reasons such as enhanced safety, developmental efficiency, and higher-level language features.

We are primarily interested in code-size reduction approaches for languages whose execution models are based on bytecoded abstract machines. Prominent examples include Java and the Java Virtual Machine [1], Prolog and the Warren Abstract Machine [2], and Haskell and the G-machine [3, 4]. These sorts of languages are not traditionally used for size-critical applications. We are investigating ways in which the compiled bytecode size of programs can be reduced and how the associated runtime systems can be tailored to specific programs to further reduce program size.

In this paper we describe a new, simple and effective method of code-size reduction called LZW-based Code Compression (LZW-CC) that uses a slightly modified version of the well-known Lempel-Ziv-Welch (LZW) text compression algorithm [5, 6, 7]. LZW was chosen as the basis of our approach because of its simplicity and its use of a fixed-size codeword which matches the encodings of most bytecode languages.

Input to LZW-CC is the bytecode for both a program and the standard library routines used by that program. The output is equivalent but shorter bytecode for the program and library routines using a customised instruction set. The added instruction codes stand for sequences of instructions from the original code as identified by the modified LZW algorithm.

As well as compressing the bytecode, we also modify the interpreter component of the run-time system so that is capable of executing programs that use the extended instruction set but doesn’t have support for instructions that are not used in the program or library routines under consideration.

Our experience so far with LZW-CC shows that it compares favourably with existing software-based approaches to code-size reduction. It is relatively straightforward to implement, yields reasonable compression ratios for typical programs, and should not impose a high runtime overhead. Compilation times for individual modules are slightly slower than usual. When the entire standard library is included in the compression phase, overall compilation times are naturally longer than a regular compilation that can use precompiled library modules. However, LZW-CC imposes less compile-time overhead than other similar methods. It is also simple to implement.

The rest of the paper is structured as follows. The next section places LZW-CC in context with previous work by introducing the main terminology of the area. Section 3 describes LZW-CC in detail by way of an artificial example. In Section 4 we describe a limited experimental evaluation of LZW-CC based on a prototype implementation in the nhc98 compiler [8] [9] applied to a suite of typical Haskell programs. We also undertake a brief analysis of the theoretical optimal performance of compression on this data. Section 5 compares the performance of the method with other notable software code-size reduction methods. Finally, in Section 6 we sum up and point toward future work.