Link-time Compaction of MIPS Programs

Matias Madou, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere

***Abstract*—Embedded systems often have limited amounts of available memory, thus encouraging the development of compact programs. This paper presents a link-time program compactor for the embedded MIPS architecture. The application of several important data flow and control flow analyses and the related program transformations at link-time are discussed and evaluated for a collection of typical embedded applications compiled against the uClibc library targeted at the embedded market. With the presented link-time compactor, code size reductions of up to 27% are obtained, and speedups of up to 17%.**

***Index Terms*—compiler, linker, compaction, optimization**

I. Introduction

The MIPS architecture is intended for high performance, low-power, system-on-a-chip applications, such as smart cards, point of deployment devices, digital cameras, set-top boxes, GPS-systems, etc. The production cost and power consumption of such mass-production embedded systems are becoming increasingly important, which results in limited amounts of memory on such systems. Of course, memories can only be made as small as the programs that need to be stored in them.

To reduce the program size of programs written in statically bound programming languages such as C and C++, we have proposed link-time code compaction [3, 1, 2] in the past. By applying an additional program optimization pass at link-time, code overhead resulting from separate compilation (of source code files and of application and library code) can be eliminated to a large extent. Until recently, research into link-time program compaction remained in the proof-of-concept phase: the Squeeze++ prototype compactor [3, 2], e.g., was evaluated on the Alpha Tru64Unix platform. This workstation and server platform can hardly be called an embedded platform, as neither its architecture, nor its compilers or system libraries are oriented towards the embedded market.

In this paper, we present our link-time compactor for the embedded MIPS R3000 platform. This optimizer is developed in the Diablo [1] link-time code editing framework, and it applies aggressive whole-program optimizations such as constant propagation, address computation optimizations, unreachable code elimination and useless code elimination on binary MIPS code at link time.

This paper is organized as follows. Section II discusses those parts of the MIPS architecture that require special attention when editing binary MIPS code. Section III discusses the actual whole-program optimizations applied in our link-time optimizer. An experimental evaluation is the subject of section IV, after which related work is discussed in Section V and conclusions are drawn in Section VI.

Presenting author: Matias Madou; Postal Address: Ghent University, Department of Electronics and Information Systems, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium; phone: +32 9 264 33 67; fax: +32 9 264 35 94; email: mmadou@elis.ugent.be

II. The MIPS Architecture

The MIPS architecture [9] is a RISC architecture with fixed-width 32 bits instructions. This, combined with the fact that there is no data mixed in between the code in MIPS programs, makes it trivial to disassemble the binary code in a program into an intermediate program representation that is suitable for link-time optimization. Some important pipeline effects however, such as delay slots, are programmer-visible and need to be taken into account when binary MIPS code is manipulated at link-time. These effects are the subject of this section.

A. Delayed Branches

Some versions of the MIPS ISA provide delayed and non-delayed branches. With delayed branches, the instruction following a branch (i.e. *the delay slot*) is always executed, whether or not the (conditional) branch is taken, to reduce the branch-misprediction penalty. Because non-delayed branch instructions are quite expensive in terms of branch-misprediction penalty, programmers are encouraged not to use them. But of course, when no suitable instruction can be found to fill the delay slot, code size increases because a no-op instruction has to be inserted in the code.

Since the primary focus of our link-time optimizer is code compaction, we might have chosen to use non-delayed branches to decrease the number of no-ops inserted. We have not chosen to do so, because non-delayed branches will probably be removed from the next revision of the MIPS architecture. Therefore introducing non-delayed branches in a program would endanger forward compatibility of the generated code. Furthermore, we have not found a single compiler that generates code with non-delayed branches. Introducing non-delayed branches at link-time to decrease code size would therefore not only be unwise for compatibility reasons, it would also be unfair: the code size reduction obtained at link-time would no longer result from typical link-time optimization opportunities, but from a compiler's deliberate choice not to exploit some (dubious) part of the architecture.

B. Delayed Loads

Besides branches, load instructions may also have a delay slot in MIPS processors. Since the data loaded by an instruction is not available during the next cycle, the processor must either be able to lock the pipeline when the next instruction consumes the loaded value, or such dependencies must simply be forbidden. While implementing interlocked loads increases the complexity of a processor's circuitry, thus increasing the power consumption of a processor, disallowing load dependencies in two consecutive