Sifting out the Mud: Low Level C++ Code Reuse

Bjorn De Sutter Bruno De Bus Koen De Bosschere
Electronics And Information Systems Department
Ghent University, Belgium

{brdsutte,bdebus,kdb}@elis.rug.ac.be

ABSTRACT

More and more computers are being incorporated in devices where the available amount of memory is limited. This contrasts with the increasing need for additional functionality and the need for rapid application development. While object-oriented programming languages, providing mechanisms such as inheritance and templates, allow fast development of complex applications, they have a detrimental effect on program size. This paper introduces new techniques to reuse the code of whole procedures at the binary level and a supporting technique for data reuse. These techniques benefit specifically from program properties originating from the use of templates and inheritance. Together with our previous work on code abstraction at lower levels of granularity, they achieve additional code size reductions of up to 38% on already highly optimized and compacted binaries, without sacrificing execution speed. We have incorporated these techniques in SQUEEZE++, a prototype link-time binary rewriter for the Alpha architecture, and extensively evaluate them on a suite of 8 real-life C++ applications. The total code size reductions achieved post link-time (i.e. without requiring any change to the compiler) range from 27 to 70%, averaging at around 43%.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*C++*; D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4. [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms

Experimentation, Performance

Keywords

Code compaction, code size reduction

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

1. INTRODUCTION

More and more computers are being incorporated in devices where the available amount of memory is limited, such as PDAs, set top boxes, wearables, mobile and embedded systems in general. The limitations on memory size result from considerations such as space, weight, power consumption and production cost.

At the same time, ever sophisticated applications have to be executed on these devices, such as encryption and speech recognition, often accompanied by all kinds of eye-candy and fancy GUIs. These applications have to be developed in shorter and shorter design cycles. More complex applications, i.e. providing more functionality, generally mean larger applications. Additional functionality is however not the only reason why applications are becoming bigger. Another important reason is the use of modern software engineering techniques, such as OO-frameworks and component-based development, where generic building blocks are used to tackle the complexity problem. These building blocks are primarily developed with reusability and generality in mind. An application developer often uses only part of a component or a library, but because of the complex relation between these building blocks and the straightforward way linkers build a program (only using symbolic references), the linker often links a lot of redundant code and data into an application. Even useful code that is linked with the application will often involve some superfluous instructions, since it was not programmed with that specific application context in mind.

During the last decade, the creation of smaller programs using compaction and compression techniques was extensively researched. The differences between the two categories is that, while compressed programs need to be decompressed before being executed, compacted programs are directly executable.

At the hardware side, the techniques used range from architectural design to hardware supported dynamic decompression. Examples are the design and wide-spread use of the code size efficient Thumb ISA [36] and dynamic decompression when code is loaded into higher levels of the memory hierarchy [22, 35].

At the software side, a wide range of techniques is developed. Whole-program analysis and code extraction avoid to some extent the linking of redundant code with a program [1, 32, 33]. Application-specific, ultra compact instruction sets are interpreted [18, 19] and frequently repeated instruction sequences within a program are identified and abstracted into procedures [6, 17] or macro-operations [4]. Dynamic