There was a time when programmers worked with assembly code. Computers were rare, expensive machines and a good programmer could pack a lot of functionality into a few kilobytes of memory. During this period, being a good programmer or designer mainly meant being a good compactor [2]. Since that bygone era, several technological advances have occurred. Very large scale integration has made memory large and inexpensive, and advances in programming languages have brought us high-level programming tools to develop extremely large and complex applications.

As a result of this progress, programming has become a different engineering discipline. Component-based, object-oriented programming [1] focuses on handling the complexity of large applications—one of its fundamental purposes is to increase programmer productivity by enabling code reuse, not to produce efficient applications. And that’s the way we want it to be: programmers and software designers should concentrate on the program logic and correctness, and not on speed or size issues of the resulting application. But is this really the case today? For those working in the embedded domain, the answer is often a firm “No” [5].

The first important reason for this response is that hardware remains expensive in our world of mobile, wireless networked consumer electronics. Because of mass-scale production, the per-unit software cost of consumer electronics is overshadowed by the hardware cost. The most dominant factor in the latter is total die size, in which memory plays an important role. Often a large fraction of the die area is devoted to memory, either to store code and data permanently (ROM) or temporarily (RAM or caches).

A simple way to make systems cheaper is to reduce the size of on-chip memories. Of course ROM or RAM memory can only be made smaller if the programs stored in it become smaller as well, and reducing cache size will typically degrade performance. These are important reasons to compact programs. Conversely, given a fixed hardware budget, reducing the size of some part of a program might free up memory space to add additional functionality.

The second important reason why things...
are different in the embedded world is that, unfortunately, most modern programming paradigms do not support the fact that smaller programs result in cheaper and possibly better performing systems. In contrast, these paradigms and today’s programming environments result in ever larger programs. This is not only due to the increasing complexity such environments allow us to handle, but also to the overhead they introduce in order to hide much of the lower-level issues from the programmer. Code libraries, for example, are written with general applicability in mind. As a result, they provide more functionality than is typically needed by any single application. Unless the unused functionality can be filtered out, an application will be larger than is needed to encode its functionality.

Because of the overhead they introduce, modern software engineering paradigms are often not considered viable for developing embedded applications. And when such paradigms are being adopted, the C code or assembly code generated from a higher-level specification often still has to be optimized and compacted manually. In other words, often good embedded programmers still need to be good compactors. Due to increasing software complexity and shrinking time-to-market constraints, manual code compaction becomes more cumbersome. Thus, during the last decade, automated compaction techniques have been an increasingly important subject for research and development.

This special section presents some of the most promising techniques to automate the process of program compaction and to overcome much of the program size overhead resulting from the use of contemporary software engineering paradigms. Our intent in this section is to give designers in different domains a larger design space to explore for their systems by presenting insights into some of the available compaction techniques. Toward that goal, five software techniques are presented in the articles that follow. The techniques differ in applicability, resulting program size reductions, and side effects on performance.

The first two articles discuss techniques to overcome the disadvantages of code libraries and separate compilation of source code files. These techniques have the advantage that they do not degrade performance but instead often improve it. The first article, on application extraction for Java programs, explains techniques that operate on program representations providing high-level semantic information such as type information. The second article, “Post-Pass Compaction Techniques,” describes methods that can be used when such information is not available. Those techniques are typically used to compact programs written in assembly, C, or C++-like languages that are compiled directly into machine code.

The third article in this section discusses how programs (from which all superfluous code has been removed) can be compacted using mixed-width instruction sets. These instruction sets are exploited automatically without having to compromise between program size and program performance.

The “Cold Code Compression at Runtime” and “Grammar-based Compression of Interpreted Code” articles take compaction a step further, abandoning the requirement that all code in an application must be directly executable on some native or virtual machine. Instead they rely on (partial) decompression and properties of interpretable languages to obtain even smaller programs. This comes at a price though, as the use of these techniques can degrade performance significantly.

It is important to note that the techniques presented here do not cover all existing methods for code compaction. In particular, this section does not cover an important body of work on hardware-controlled decompression, whereby code is decompressed when it is transferred between different levels of a memory hierarchy or when it is fetched from memory by the processor [3, 6]. Neither does this section cover (data) compression techniques that one can apply to entire programs to limit the amount of required storage space or network bandwidth (so-called wire-formats [6]). Before such a compressed program can be executed, it must be decompressed to its original representation, which, together with the decompressor, requires not less but more memory. Our choice of techniques is not an arbitrary one. Other existing techniques, such as hardware-controlled decompression and the use of wire-formats, are largely orthogonal to the software compaction techniques presented here.

References


Bjorn De Sutter (bdsutter@elis.ugent.be) is a post-doctoral research fellow of the Fund for Scientific Research–Flanders, Belgium.

Koen De Bosschere (kdb@elis.ugent.be) is a professor in the Electronics and Information Systems department at Ghent University in Belgium.

© 2003 ACM 0002-0782/03/0800 $5.00