Compacting Arm Binaries with the Diablo Framework

Dominique Chanet and Ludo Van Put

22 september 2003

1 Introduction

On most embedded systems, memory space and power resources are limited. As a consequence, significant effort is spent in creating resource-efficient programs for embedded systems. Compilers for embedded systems usually try to generate the most compact code possible, and libraries for embedded systems are usually quite a lot smaller than those for general purpose computer systems.
One area of improvement that has in general been overlooked, is the linking process. Recent research [2] has shown that it is possible to achieve considerable program compaction at link time, as the linker has a view of the complete program, including libraries, and therefore it is not bound to a number of the conservative assumptions a compiler has to make. These techniques were however developed for general-purpose architectures, and have not yet been evaluated in the context of an actual embedded platform, using actual embedded toolchains.
Our aim is to evaluate these techniques for the ARM platform, which is widely used in cell phones, PDAs and other embedded devices. We use Diablo, a retargetable framework for binary rewriting at link time, that has recently been developed at the ELIS research group of Ghent University.

2 The ARM architecture

The ARM architecture is very widely used in the embedded market space (according to ARM, 76% of all RISC processors shipped in 2001 implemented the ARM instruction set). Some examples of widely used embedded processors based on the ARM instruction set are the Intel XScale and StrongArm processors (mostly used in PDAs) and the Texas Instruments OMAP (mostly used in cell phones).

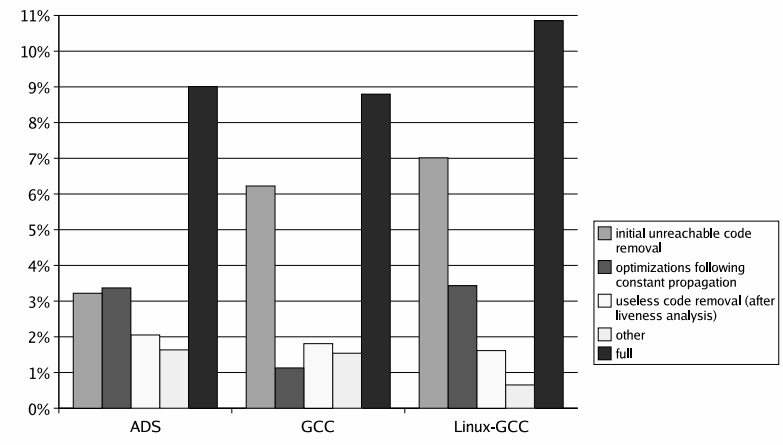

Figure 1: Impact of the optimizations

Because of this focus on the embedded market, the ARM architecture has a number of features that facilitate the generation of compact programs, like conditional execution of all instructions (this eliminates the need for jump instructions with "small" if tests). Furthermore, in the more recent revisions of the architecture an extra instruction set, called Thumb, has been introduced. Thumb instructions are only 16 bits wide (half the size of regular ARM instructions), which results in even more compact programs. The downside is that a number of ARM operations cannot be expressed in a single Thumb instruction, so the execution time of an application increases slightly with the use of the Thumb instruction set. Fortunately the two instruction sets can interoperate, so it is possible to use the ARM instruction set for time-critical pieces of code and the Thumb instruction set for all other parts.

3 Link time compaction with Diablo

Diablo is designed to operate on the information a linker has at its disposition. It reads in all object files and libraries the program consists of, and links these together. In the process, Diablo gets to know the exact contents of all data and code sections, how these sections are composed of smaller subsections from

the different object files and libraries, and all relocation information that was stored in the object files and libraries.
Using this information, an interprocedural control flow graph of the complete program is built. This graph is the highest-level representation of the program in Diablo. On this representation all analyses and optimizations are performed.

4 Main sources of link time compaction

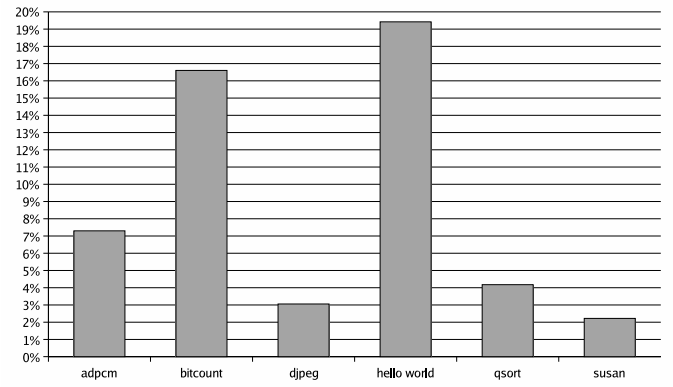

Figure 2: Compaction results with arm-elf-gcc

Using Diablo a link time compaction prototype was built for the ARM platform. With this prototype binaries produced with the ARM ADS toolchain (with the ARM C library), the `arm-elf-gcc` toolchain (with the `newlib` C library) and the `arm-linux-gcc` toolchain (with the `glibc` library) were compacted. All binaries were generated for the ARM instruction set, we have not yet implemented support for the Thumb instruction set in Diablo. However, it is our belief that the results will not notably change for mixed ARM/Thumb programs. The benchmarks we used were taken mainly from the MediaBench, a benchmark suite that contains programs that are typically used on embedded systems. We've also added results for the ' Hello World' - program, as a 'minimal' application. Overall compaction results are shown in figure 1, as well as the amount of compaction contributed by different optimizations. The compaction for each benchmark is shown per toolchain in figures 2, 3 and 4.
The major source of optimization is the removal of unreachable code. This is done immediately after the construction of the interprocedural control flow graph, using the following simple algorithm:

1. add the program entry point to the list of reachable blocks.
2. for each block in the list of reachable blocks:
   (a) add all successors of this block to the list of reachable blocks
   (b) for all data blocks that can be accessed from this block: add all code blocks that can be reached from this data block to the list of reachable blocks
3. remove the blocks that cannot be found in the list of reachable blocks from the graph

Step 2a marks all code that can be reached through direct control flow transfers (direct jumps and function calls), while step 2b marks all code that could possibly be reached through indirect control transfers from already reachable code.
The second major source of compaction are the results of interprocedural constant propagation. This analysis is performed on registers only, constants are not propagated through memory, as it is very hard at link time to gather enough aliasing information to do a propagation through memory. Using the results of this analysis one can do the obvious things like constant folding and removing idempotent instructions (i.e. instructions that do not change the state of the program in any way). Much more interesting however are the possibilities for refining the control flow graph: for a number of conditional branches it is possible to determine which branch will be taken, so we can remove the other branch. For some indirect branches we will also be able to determine the exact destination, which allows us to remove a number of unrealisable paths from the control flow graph. Furthermore, because constant propagation at link time can also propagate addresses, which isn't possible at compile time, a number of address calculations can be simplified by reusing the results of previous, unrelated address calculations. This is something a compiler cannot do, as it does not know the final relation between addresses that lie in different subsections (the order of the subsections is determined by the linker).
The third significant contribution to code compaction comes from interprocedural liveness analysis. Again, this analysis is only performed on the values stored in registers, not memory. Instructions that are found to produce no live values at all are useless and can safely be removed. Compilers usually do a similar optimization already, but thanks to the whole-program view at link time a number of new possibilities arise. Furthermore, optimizations like the previously mentioned constant folding also render previously useful instructions useless.

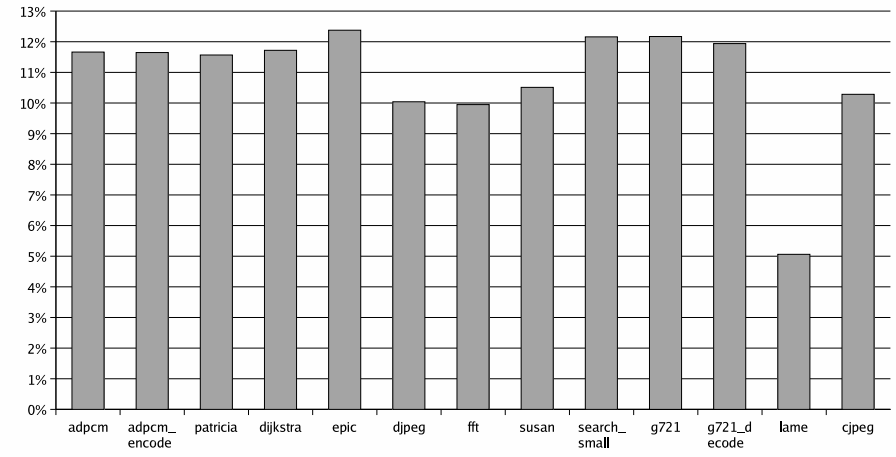

Figure 3: Compaction results with arm-linux-gcc

We've also implemented some other optimizations like branch forwarding and copy propagation. These are roughly equivalent to their compiler counterparts and do not achieve any significant compaction. Their primary purpose is to tie up some "loose ends" left behind by the major optimisations.
Whereas the maximal compaction results for the ADS and arm-elf-gcc toolkits seem to vary a lot, the maximal compaction achieved for the arm-linux-gcc benchmarks almost all lie in the relatively small window between 10% and 12%. The reason for this is that the glibc library used with this toolchain is several times larger than the ADS and newlib C libraries of the other toolchains. Consequently, the ratio of application-specific code versus library code in the linux binaries is almost constant. It is precisely in this library code that the link-time compaction techniques achieve most gain, which explains the small variance in the compaction results for the linux binaries.

5 Future work

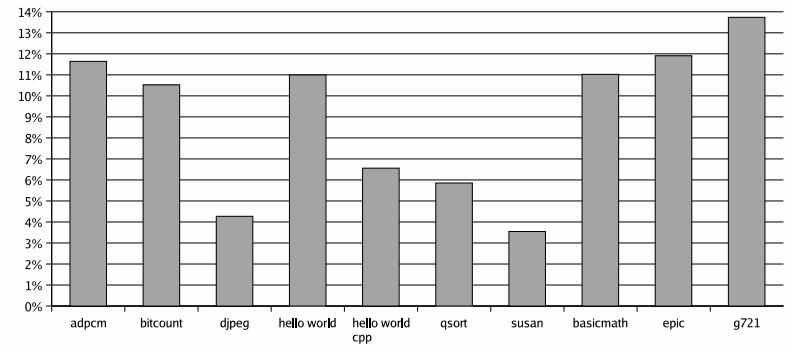

Figure 4: Compaction results with arm ADS

Not all possible compaction techniques have yet been implemented for the ARM architecture, so there is still room for improvement to the compaction results already achieved. The impact of code abstraction and detection and removal of dead data blocks from the program [1] is one of the things that should certainly be evaluated in the future. There also remains some overhead due to the calling conventions that have to be respected when using library code. This overhead is partly reduced by Diablo, but additional efforts can be done. In order to remove more of this overhead, more detailed knowledge about the program's stack behaviour is needed. An analysis that produces this information is not yet implemented in Diablo. The impact of the use of Thumb code in programs and its implications on link-time compaction also deserve some further investigation.

6 Conclusions

Thanks to the interprocedural nature of the analyses and optimizations performed at link time, compaction at link time can remove some of the overhead introduced by the separate compilation of program source files. The achieved compaction ranges from 2% to 19% and is for the most part the result of the removal of unreachable code (both superfluous library code linked in by traditional linkers and code that becomes unreachable as a result of other optimizations performed at link time).

References

[1] Bjorn De Sutter. Compactie van programma's na het linken. Doctoraatsthesis, Universiteit Gent, 2002.

[2] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Saumya Debray. Combining global code and data compaction. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.