Program Analysis and Optimization for Embedded Systems

Mike Jochen and Amie Souter

{jochen, souter}@cis.udel.edu

Computer and Information Sciences

University of Delaware

Newark, DE 19716

(302) 831-6339

1 Introduction

A proliferation in use of embedded systems is giving cause for a rethinking of traditional methods and effects of program optimization and how these methods and optimizations can be adapted for these highly specialized systems. This paper attempts to raise some of the issues unique to program analysis and optimization for embedded system design and to address some of the solutions that have been proposed to address these specialized issues.

This paper is organized as follows: section 2 provides pertinent background information on embedded systems, delineating the special needs and problems inherent with embedded system design; section 3 addresses one such inherent problem, namely limited space for program code; section 4 discusses current approaches and open issues for code minimization problems; section 5 proposes a new approach to the code minimization problem; and finally section 6 provides areas of related work to embedded systems.

2 Background

Consumers' hunger for smaller, faster, more powerful gadgets in the telecommunication, multimedia, and consumer electronics industries is pushing the integration of complete systems onto a single chip [Lie97]. The demand for consumer oriented, wireless communication and multimedia devices directly impacts the design of the underlying architecture for such gadgets. Embedded systems are quickly becoming the means to meet these consumer demands. Embedded systems are much more specialized in nature than general computing systems (e.g. desktop computers, laptop machines, servers, and workstations). Embedded systems are designed to serve dedicated functions (e.g. anti-lock brake systems, digital cellular phones, video coding for HDTV, audio coding for surround sound, etc.).

The special nature of use of these personal digital assistants (PDA's) imposes several critical criteria on their design. These systems must:

- have low cost
- have low power consumption
- require as little physical space as possible
- meet rigid time constraints for computation completion

These requirements place constraints on the underlying architecture's design, which together affect compiler design and program optimization techniques. Each of these criteria must be balanced against the other, as each will have an effect on the other (e.g. making a system smaller and faster will typically increase its cost and power consumption). A typical embedded system, as illustrated in figure 1, consists of a processor core, a program ROM (Read Only Memory), RAM (Random Access Memory), and an ASIC (Application Specific Integrated Circuit). The cost of developing an integrated circuit is linked to the size of the system. The largest portion of the integrated circuit is often devoted to the ROM for storing the application. Therefore, developing techniques that reduce code size are extremely important in terms of reducing the cost of producing such systems.

Unfortunately, the current state of the art in compilation for embedded systems is somewhat left to be desired. We live in an era where memory for the general use computer is abundant and processing speed is remarkably fast. Thus issues such as limiting code size for compiled code have taken a back seat to other optimizations. Compiler technology for embedded system design must support:

- compilation for low-cost, irregular architectures (instruction set programmable architectures) like microcontroller units (MCU'S), Digital Signal Processors (DSP's), and application specific instruction set processors (ASIC's)
- rich data structures to support complex instruction sets for the above
- extensive searching (helpful in register allocation, scheduling, & code selection)

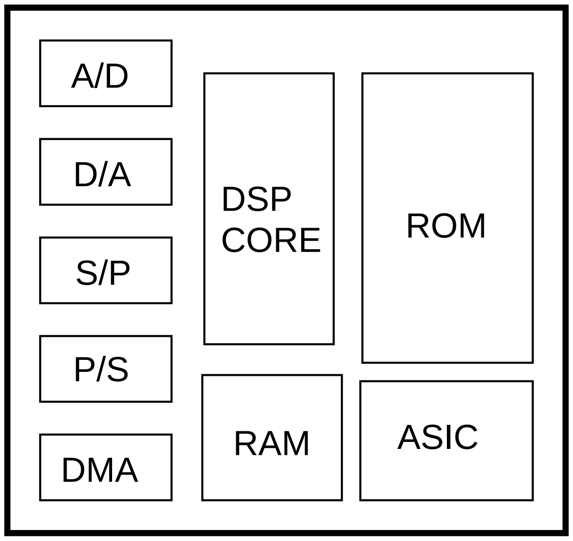

Figure 1: A typical embedded system.

- methods for capturing architecture specific optimizations easily

All of these new requirements render current compiler technologies at least inefficient and at most unusable for embedded systems. In fact, most source codes for embedded systems today are written in assembly, despite all the benefits that come with today's high level languages. The costs of using these high level languages via traditional compilation are too great in terms of executable code size, execution time, and number and type of registers used. Further, a compiler for embedded systems must be:

- retargetable - able to compile to a number of changing instruction sets for different architectures
- able to handle register constraints - most embedded processors have a number of special-purpose rather than general-purpose registers (allows for tighter instruction coding)
- able to handle specialized arithmetic - DSP's often have specialized math operators which require more than 3 operands (thus rendering 3-address-code useless)

The complexity of these systems are forcing designers who were once able to implement functions in hardware to switch to implementing the functions in software and then burning the program onto a chip. There is now a motivation to switch to high-level languages to program embedded systems, assembly languages are being phased out because development costs are lower when using high-level languages. One disadvantage of implementation via high-level languages is increased code size. This is a problem that must be addressed because embedded systems are usually constrained to relatively small memories.

Thus, with the onslaught of embedded processors and embedded systems, we are now forced to re-examine many of today's optimizations and their effect on the target code in terms of code size, register use, instruction scheduling, and garbage collection, among other issues.

3 Code Minimization for Embedded Systems

The whole premise of an embedded system demands that it be small; no one wants to carry around a portable phone that is as large as a toaster. This constraint on size means there can be no wasted space. Put more succinctly, everything from chip design to on-board memory must take as little space as possible. To this end, a few approaches to limiting the amount of memory needed to hold executable code have surfaced. Some techniques look at instruction scheduling [CS98, CSS99] and its effect on target code size. Other techniques employ some form of pattern matching [CM99, LDK99] to eliminate regions of repeated code. A third genre of technique attempts to physically compress the target code [LW98] to save on precious memory space.

An attempt to limit target code growth via efficient instruction scheduling was introduced by [CS98]. This technique merely tries to repair the damage (the increase in code space requirements) normally done by instruction scheduling optimization (e.g. finding another sequence of instructions that minimizes the number of 'nop' instructions). This is done by scheduling larger regions of code which allows a higher degree of fine-grain parallelism (ILP) to be achieved, thus eliminating 'nop' or stall instructions. This technique adds a good deal of overhead (in terms of time) to the scheduling phase of compilation. Somehow finding a way to decrease instruction scheduling time while increasing the size of the block of instructions to be scheduled would be welcomed improvements.

In [CSS99], Cooper et al. expanded the above approach to include a series of optimizations in which genetic search algorithms find a sequence of optimizations that produce small codes. Genetic algorithms are search algorithms designed to mimic the process of natural selection and evolution in nature. The goal of the algorithm in this case is to search for the best optimizing sequence that creates minimal codes. Cooper at al. validates their approach with experiments using C and FORTRAN codes. The genetic algorithm approach produces code sizes that range from 20 to 75 percent smaller than unoptimized code. They also show that the GA produces codes with 0 to 40 percent fewer static operations, and that the GA can produce codes with 25 to 26 percent fewer dynamic operations. Since code for embedded systems is often compiled once and burned onto a ROM, the software designer will tolerate longer compile times. Genetic algorithms do take a long time to run. For their results, they ran the GA for 10 hours to find the best optimization sequence. The GA approach typically produces small object codes, but not always. Also, the code produced may sometimes run more slowly than the original code, but it is stated that code speed comes secondary to code size.

The goal of Liao et al's work is to achieve maximal code compression while incurring very little performance penalty [LDK99]. The basic technique uses a data compression approach where common sequences of code are extracted to form dictionary entries and are replaced by mini subroutine calls to the dictionary. Liao et al. formulate the dictionary entry selection and substitution problem as a set-covering problem. Two methods are introduced, one based on software and the second based on hardware/software. The automatic minimization of code size relieves the programmer from worrying about making programs small and allows them to write programs in a high level language. Their results show that the hardware/software approach outperforms the software only approach by 2-5%. The drawback of the hardware/software approach is the requirement for specialized hardware. The disadvantage of both techniques is the size/speed trade-off. The time needed to uncompress code at run-time can be a hindrance to the benefit of the small code size. But slow speeds are tolerated when small code size is required due to system specifications.

Liao et al. and Cooper and McIntosh have mildly similar approaches in code minimization techniques. Liao et al. finds common sequences of code and inserts them into a dictionary. The code sequence is replaced with a mini subroutine call to the dictionary. Cooper and McIntosh's approach differ in that it finds regions of code that are similar and deletes all subsequently similar regions. Control is then funneled through the first region [CM99]. This technique also employs some interesting ways to identify larger and a greater number of regions of similar code. Here register names and the names for the targets of branches are abstracted (removed and given a generic name). For example, if two pieces of code were similar except that they used different register names (and those registers could be re-ordered without causing harm to the semantic content of the code) then those regions could be made to use the same registers and thus be identified as similar regions. This technique did increase dynamic instruction count, however, which increased runtime slightly. This needs to be avoided. Also, instruction reordering should be considered to permit more regions of code to be identified as similar.

The only approach to try to physically compress the target code is [LW98]. Here the target code is compressed cache line, by cache line. The non-sequential nature of object code (ability to branch and jump to different regions of code) precludes compression at the file level, it must be done at some smaller granularity. Decompression occurs when there is a cache miss. With this technique, on each cache miss, the cache line is fetched from memory and sent through a decoder before it is given to the cache. Thus, decompression is done on the fly at run time. They present two algorithms for compression using such an approach. Their results show that these approaches achieve good code compression results, comparable to UNIX Compress and gzip. The difference is that their approach can be performed separate from compilation. This approach does require special hardware, and there is some increase in cache miss penalty as decompression must also be included with each cache miss.

An application of code compression is shown in the work by [YRE98]. They are trying to compress the Java machine in order to make it feasible to run on the Inferno OS (an OS for distributed devices). They do not introduce any new techniques, but show that Java programs are capable of running on small devices. This paper addresses the issues of code size and memory needed for execution in terms of building a Java implementation to run on the Inferno OS. Their design had several goals, namely, the size of the OS should not be affected in order to run Java, running Java should use as little memory as possible and run as fast as possible, and the resulting code should be completely portable. They implemented their design which resulted in the OS size increasing by only 11Kb. This is not the fastest Java engine, but it is small both statically and dynamically and it allows for Java to be used in places it otherwise could not.

4 Current Approaches and Their Open Issues

Cooper and Schielke attempt to reduce code size during compiler instruction scheduling by finding larger regions of code to schedule [CS98]. Scheduling larger regions of code enables greater instruction level parallelism, which reduces the number of NOP instructions needed. Cooper and Schielke describe several possible improvements to their work. One such improvement would be to quickly generate larger acyclic code regions which may lead to reduced run times while still limiting code growth. Another possible improvement, concluded from their results is to find a speedier implementation for dominator path scheduling (DPS) as DPS yields the best improvement for minimizing code size.

Lekatas and Wolfe reduce code size by compressing code in cache-line size chunks in memory [LW98]. This technique utilizes specialized hardware to perform code decompression between memory and cache, but there is no insight as to how costly the decompression phase is in terms of run time. With each cache miss, a cache-line goes through decoding first, then it is read into cache. Their insights into improving their technique are to find a way to generate the best Markov model for a given code (for SAMC), or producing better dictionary entries (for SADC), thus improving compression efficiency for the two methods. Building better/faster decompression hardware would improve runtime performance but again, since runtime performance is not mentioned in the paper, we are not sure of the costs (thus the desirability) of this method.

Cooper and McIntosh reduce code size by coalescing repeated sequences of code identified via a pattern matching mechanism [CM99]. This technique is similar in nature to Liao et al. who reduce code size by replacing repeated sequences of code with dictionary entries [LDK99]. The key open issues discussed in [CM99], include applying the same register name abstraction technique to constants, this should allow more code to be identified as repeat code. Instruction reordering before pattern matching is applied could also prove to be useful. For example, if two code segments are mostly the same except for the order of some operations and the two code segments are semantically equivalent, then the current algorithm will not mark these segments as repeat code. If these instructions are reordered to match between the two segments, then these segments could be identified as repeat code. Performing compression before register allocation would also permit more code to be identified as repeat code (since registers could be freely assigned/changed at that point). But this approach could complicate register assignment quite a bit. Liao et al. also discuss the idea of improving the code generator in order to create more compressible codes. The trade-offs of using different code generation techniques and applying different heuristics to these techniques should be explored in order to improve compression methods.

Finally, Cooper et al. explore techniques that utilize genetic algorithms to search sequences of optimizations to find the particular sequence that yields the smallest code size [CSS99]. They conclude that more experimentation needs to be performed by using different optimizations, changing the number of optimizations available, as well as applying different optimizations to different sections of the program.

5 Proposed Approach and Evaluation

All the above approaches to code minimization suffer from the same flaw, they all address the problem of shrinking code size after it has been generated with (potentially) flawed optimizations as measured in terms of producing small target code. Historically, optimization has focused on producing fast code; code size has largely been neglected.

Our general approach focuses on how solutions to various optimization problems affect code size and to make these solutions cognizant of these effects. This would be a pro-active approach to code minimization rather than reactively fixing the code expansion produced by previous optimizations. This multi-tiered framework is broken into three separate strategies: a reconsideration of the effects of program optimization, a look at the usefulness of different program representations, and the impact that various programming language features have on code size.

5.1 Program Optimizations

Current optimizations need to be re-examined in terms of their effect on code size, not just on runtime efficiency. We will examine these optimizations to determine portions that can be attributed to increased code size. These portions of any problematic optimizations will be tweaked to yield less offensive results. After each optimization has been repaired, methods from the above papers can be combined to produce even smaller target code. For example, the genetic search algorithm can be applied to the new and improved optimizations to find the best sequence for minimized code. The output of this process could then be sent through the compression methods of [LDK99], [CSS99], [CM99], or some combination of these three.

This first approach will be evaluated in the following manner. Care must be taken to explore the possibility that gains in one optimization could adversely affect other optimizations.

1. Generate a generic target code. This code is our base case, which contains no optimizations (other than dead code elimination, and any other optimizations that trim code from a given program), assumes an infinite number of available registers, and no data, structural, or control hazards. This yields an artificially small target code that can be used to compare target codes and identify the positive and negative effects of each optimization (and combination of optimizations).

2. Look at optimizations to determine which optimizations affect code size in a positive or negative manner (via comparison to the above artificially small target code). Once an optimization is found to be code size unfriendly, some heuristic can be applied to attempt to change that optimization without harming the intended effects of that optimization. Part of this evaluation will consider various heuristics that yield the best results.

3. Use the genetic algorithm approach to further reduce code size.

5.2 Program Representations

Another area of exploration is to examine various program representation forms in order to determine if some particular program representation can bring more power to any code minimization technique. For example, any methods that utilize pattern matching techniques on the target code, may benefit from and create more opportunities for code minimization by utilizing an intermediate program representation. Different program representation may be beneficial in finding regions of repeated code, or larger basic blocks.

This second approach will be evaluated in the following manner.

1. Study the advantages and disadvantages of program representations applicable to code minimization.
2. After determining which program representations to use, or augmenting or creating a new representation, build old optimizations on top of these new program representations in hopes to discover new opportunities for code minimization.

5.3 Programming Style

Perhaps the features of the source code language also contribute to target code growth. We will explore the effects of different high-level programming languages and language features with respect to how they affect code size. It should be determined if certain features of a language that explode code size can be removed or avoided without impairing an otherwise desirable language.

The third approach will be evaluated in the following manner.

1. Identify features of a language that create opportunities for code growth. For example, which features are most the costly in terms of space, and which are the most efficient in terms of space.
2. Identify alternatives for costly features.
3. Identify alternative languages should the above approach prove unsuccessful by studying the features of different languages.

5.4 Summary

All of our approaches try to identify problems that help in reducing code size at an earlier stage in the compilation process. Each approach tries to identify areas where code growth occurs and subsequently to alleviate the problem at that point. We think this is a more efficient approach rather than trying minimize the code at the end of the compilation process.

6 Related Work

Other work related to embedded systems has also been investigated. Such work includes other optimizations geared towards performance and code minimization. Also, run-time issues have been investigated such as garbage collection for embedded systems as well as performance analysis.

6.1 Other Code Optimizations

Register allocation is another challenging task for code generation of embedded systems. Embedded systems typically have a small number of registers, of which several are designated for special use. The focus of the work by Kolson et al. is to find optimal register assignments in loops in order to profit from the fact that statements in a loop execute more often than other portions of the code [KNDK96]. Their work focused on embedded system architectures that are characterized by a single register file and then extended the work for architectures that are characterize by distributed memories. Liao et al. also address the problems of register allocation for embedded systems [LDK$^+$95]. They focus on processors which exhibit highly irregular data-paths, such as Texas Instruments' TMS320C25 which is an accumulator-based machine. Such processors display different characteristics than RISC based machines. Scheduling and register allocation have typically been solved as separate problems. They present optimal and heuristic algorithms that determine an instruction schedule while simultaneously optimizing accumulator spilling and mode selection.

Data partitioning optimization has also been investigated in embedded systems [AP98], particularly data partitioning for arrays on limited memory embedded systems. Many embedded applications are memory intensive. The data segment of code is often split between the on-chip and off-chip memory, therefore remote memory references which occur due to this situation can lead to significant degradation in real time response to these systems. The goal of this compiler optimization is to analyze loops and partition data efficiently within the loop based on the frequency of generated references. They use an efficient algorithm based on the 0/1 knapsack problem where they map partitions on local/remote memory. The main contributions is the identification of the data footprint of each reference (memory demands) and the framework for its 0/1 solution. The disadvantage of this approach is that it does not always work well when there is extremely low memory available.

Another use of partitioning in embedded systems is for defining the boundaries of hardware and software functionalities. Agrawal and Gupta use data-flow analysis within a graph partitioning framework to more efficiently estimate communication costs between software and hardware partitions within an embedded system [AG97].

6.2 Run-Time Issues

The runtime performance or analysis of embedded systems bring about new issues concerning include garbage collection and performance analysis.

Garbage collection methods designed for traditional hardware/software systems are not necessarily suited for embedded systems. Unique factors of embedded systems such as limited memory and hardware support drive the decision of which garbage collection algorithm to choose for the specific embedded system. In [PBT98], they discuss factors that must be considered when choosing garbage collection algorithms for embedded systems. In particular, they discuss the issues necessary for garbage collection in Embedded Java. Garbage collection is an essential part of the Java language. In order for such a language to be used in embedded systems the unpredictability of garbage collection needs to be addressed. Persson presents techniques for predicting the maximum amount of live memory in object-oriented languages [Per99]. The reason for predicting the maximum amount of live memory stems from the need to predict the worst-case execution time of a program in order to guarantee that the system will fulfill timing constraints for a real-time embedded system. Kim et al. also address the challenges due to garbage collection of real-time embedded systems [KCKS99]. They propose a new algorithm that schedules both a garbage collector and real time mutator tasks. This approach attempts to reduce the amount of system memory requirements.

The problem of determining the extreme (worst and base) case bounds on the running time of a program is addressed in [LM97]. This problem is important in embedded systems due to real-time constraints that must be satisfied on a large number of embedded systems. Li and Malik use implicit path enumeration to determine which paths in the program are exercised. Previous techniques have used explicit paths.

Benchmarks are commonly used to evaluate the performance and functionality of program analysis tools. The same benchmarks used for general use desktop systems are used for embedded systems, whose characteristics differ vastly from such systems. In particular the SpecInt95 benchmark suite is often used. Englom addresses "the gut feeling" in the embedded system community that the SpecInt95 benchmark suite is not appropriate for evaluating applications running on embedded systems [Eng99]. The study concludes that the static properties of real embedded programs are quite different than those found in SpecInt95 programs and should not be used to evaluate application used for embedded systems.

References

[AG97] Samir Agrawal and Rajesh K. Gupta. Data-flow assisted behavioral partitioning for embedded systems. In *Proceedings of the 34th annual conference on Design Automation Conference*, pages 709–712, 1997.

[AP98] Sundaram Anantharaman and Santosh Pande. An efficient data partitioning method for limited memory embedded systems. In *ACM Proceedings of the SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 208–222, June 1998.

[CM99] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. In *ACM Proceedings of the SIGPLAN 1999 Conference on Programming Language Design and Implementation*, volume 34, pages 139–149, May 1999.

[CS98] Keith D. Cooper and Philip J. Schielke. Non-local instruction scheduling with limited code growth. In *ACM Proceedings of the SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume ?, ? 1998.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM Proceedings of the SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 34, pages 1–9, July 1999.

[Eng99] Jakob Engblom. Why specint95 should not be used to benchmark embedded systems tools. In *ACM Proceedings of the SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 34, pages 96–103, July 1999.

[KCKS99] Taehyoun Kim, Naehyuck Chang, Namyun Kim, and Heonshik Shin. Scheduling garbage collector for embedded real-time systems. In *ACM Proceedings of the SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[KNDK96] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal register assignment to loops for embedded code genereation. In *ACM Transaction on Design Automation of Electronic Systems*, pages 251–279, April 1996.

[LDK$^+$95] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Code optimization techniques for embedded dsp microprocessors. In *ACM Design Automation Conference*, 1995.

[LDK99] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. In *ACM Transactions on Design Automation of Electronic Systems*, volume 4, pages 12–38, January 1999.

[Lie97] Clifford Liem. *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications*, chapter 1. Kluwer Academic Publishers, 1997.

[LM97] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, volume 16, pages 1477–1487, Dec 1997.

[LW98] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *ACM Design Automation Conference*, pages 516–521, June 1998.

[PBT98] Alexandre Petit-Bianco and Tom Tromey. Garbage collection for java in embedded systems. In *Industrial Conference on Embedded Systems*, volume 468, 1998.

[Per99] Patrik Persson. Live memory analysis for garbage collection in embedded systems. In *ACM Proceedings of the SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[YRE98] C. F. Yurkoski, L. R. Rau, and B. K. Ellis. Using inferno to execute java on small devices. In *ACM Proceedings of the SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 108–118, June 1998.