
**PROJECT TITTLE**

# CODE OPTIMIZATION FOR COMPILER DESIGN

## A PROJECT REPORT

**Submitted to**

**Department of Computer Science and Information Technology**

**SAVEETHA SCHOOL OF ENGINEERING**

***In partial fulfillment of the requirements for the Bachelor's Degree in Computer Science and Information Technology***

Submitted by

k.vishnu vardhan(192224265)

D.Deepak(192210646)

M.jyothieswar(192211468)

## Table of Contents

## 1. Abstract

This project investigated a number of problems that arise in translating computer programs for execution on embedded computer systems—compiling those programs. Embedded systems are characterized by a number of constraints that do not arise in the commodity computer world. Most of these constraints have an economic basis. Embedded computers typically have limited amounts of memory. They often employ idiosyncratic processors that have been designed to maximize their performance for a limited class of applications. The applications are often quite sensitive to performance.

Some of the problems that arise in compiling code for execution on embedded systems have solutions that are relatively local in their impact within a compiler. For example, teaching the compiler to emit code for specialized instructions on a particular processor is easily handled during instruction selection — a modern code generator, based on pattern matching, can be extended to make good use of special case operations. We investigated several of these local problems. Other problems, however, have solutions that cut across the entire compiler. We tackled several of these cross-cutting problems. In both realms (local problems and cross-cutting problems), we developed an understanding of the issues involved, did some fundamental experimentation, proposed new techniques to address the problem, and validated those techniques experimentally.

To transfer the results of this work into commercial practice, we have published papers, distributed code, communicated with industrial compiler groups, and sent students to work in those groups. The techniques developed in this project are beginning to appear in the systems of other compiler groups — both research and commercial compilers. We expect more of them to be adopted in the future.

*Major Results*

- New methods for reducing the size of compiler-generated code
- New techniques for instruction scheduling — both better schedulers for space constrained environments and stronger schedulers for hard problems
- A new algorithm for scheduling and data placement on processors with partitioned register sets — an increasingly popular feature in embedded processors
- New techniques for reducing the amount of spill code generated by a graph coloring register allocator and for reducing the impact of that spill code (both space and time)
- New techniques for some of the fundamental analyses and transformations used in code optimization for both embedded systems and commodity systems
- A new approach to building self-tuning optimizing compilers, which we call adaptive compilation.

## 2. Introduction

The embedded environment presents unusual challenges to a compiler. These systems are characterized by small memories, aggressive and idiosyncratic microprocessors, performance sensitive applications, and real-time applications. All too often, the available compilers fail to satisfy either the space or performance requirements, and the user must write at least part of the system in assembly code. While this works today, we will soon need better ways of building these systems. The rapid growth in the embedded systems marketplace, both applications and processors, suggests that not enough assembly-code wizards will be available to meet demand. Furthermore, within a hardware generation, the processors used in embedded systems will be complex enough to render effective assembly programming by humans virtually impossible.

Some of the problems that arise in targeting embedded systems have solutions that are relatively local in their impact within the compiler. For example, adding a specialized boolean instruction to the compiler's repertoire is an issue for instruction selection, easily handled by a technique like BURG. The more difficult problems have solutions that cut across the entire compiler. Our particular interest is in these cross-cutting problems: developing an understanding of the issues involved, proposing techniques to address the problems, validating the ideas experimentally, and working to move the solutions into commercial practice.

This project had three primary themes:

1. *Novel optimization paradigms* —

   The resource, performance, and timing constraints of embedded systems suggest that more powerful compile-time techniques could be applied profitably during the final stages of program development if the compiler were allowed a constant factor more time. In this investigation, we looked at ideas that included: pursuing multiple optimization strategies and keeping the best result, using randomized algorithms and restart to explore large, complex solution spaces, and fundamentally rethinking the organization of our compilers.

2. *Resource constraints* —

   The memory systems in embedded systems are almost always too small. Reducing the memory requirements of compiled code requires a concerted effort from parser to code generator. We investigated several schemes for reducing code space as a code optimization problem. We also looked at one technique for reducing data-space requirements (reducing the footprint of spill code).

3. *Architectural idiosyncrasies* —

   The microprocessor architectures used in embedded systems evolve rapidly to improve their performance. We examined several specific issues, including partitioned register sets, predicated instructions, local (non-cache) memories, and branch-delay slots.

The sections that follow describe our major results.

## 3. Methodology

Our goal for this project was to improve compilation techniques in use for embedded systems. To achieve this requires more than simply inventing new techniques that address the problems. It requires careful experimental validation of both the costs and the benefits of new techniques. It requires detailed engineering of the techniques to ensure their implementability and their practicality. (The new methods must fit into the commercial compiler. No commercial group will rewrite their entire compiler to accommodate some academic result.) It requires a mechanism for transmitting the high-level concepts, the low-level engineering details, and the implementation insights to the commercial implementor in a concise and useful form. Finally, it requires an aggressive effort to ensure that commercial implementors are aware of the new work.

Because we understand the difficulty of moving new techniques into commercial practice, we have structured our experimental methodology to help us address each of these concerns.

1. ***Problem identification*** — To find new research problems, we read and profile the output of existing compilers, and we talk to commercial compiler groups (TI, Motorola, Intel, HP, Microsoft, and others).
2. ***Preliminary exploration*** — To understand the importance of a problem and its amenability to solution, we perform an initial round of experiments. This might involve hand simulation of a transformation or the construction of a prototype implementation (often, using inefficient algorithms). If the results are promising, we continue.
3. ***Algorithmic development*** — To refine our ideas, we build a serious prototype that runs in our research compiler. In the prototype, we work out the algorithmic and engineering details required for acceptable compile-time performance. We use the prototype to test effectiveness against a collection of representative codes. This is an iterative process, where testing reveals further opportunities for improvement.
4. ***Publication and distribution***— To make the results of the work widely available, we publicize them on several levels. We publish papers in appropriate journals and conferences. We make the implementation accessible via the web. We visit with commercial compiler groups and discuss their problems and our solutions.

Historically, we have achieved reasonable success in moving ideas and techniques from our lab into commercial compilers from many companies.

## 4. Results and Discussion

This project, which ran from July 1997 through July 2001, investigated a number of issues in code optimzation and code generation for embedded systems. This section summarizes the results of our major research thrusts. The final subsection describes a number of algorithms that we developed as a result of these inquiries that do not fit into any of the major research thrusts. The annotated bibliography provides a running commentary on the various publications and technical reports that we produced.

### *Novel Optimization Paradigms*

Historically, compilers operate by applying a fixed sequence of translation steps in a fixed order. This is true on the macro level; compilers generally run their optimizations in a fixed order. It is also true on a micro level; most individual transformations attack the opportunities for improvement in a deterministic order. The compiler confronts a problem: what is the best code to generate for the source program being translated? The compiler constructs an approximation to the best answer — the code is correct but not optimal. This approach is a sensible response to the constraints under which compilers have historically operated: produce correct code quickly.

As part of this project, we explored what might be possible if we relaxed these constraints. In particular, we relaxed the constraint that the compiler itself runs quickly. This created the option of using techniques that tried multiple approaches, evaluated the results, and kept the best code — an idea accepted in register allocation since the late 1980s. We applied this notion to two problems, with three sets of interesting results.

*Iterative Repair Scheduling* — Traditional instruction schedulers operate by using a greedy list-scheduling algorithm. While the folklore suggests that these schedulers do well in practice, there was little hard data that assessed how often list schedulers produce optimal schedules.

We built a series of schedulers based on an alternative paradigm, called iterative repair, and used these schedulers to understand the space of possible schedules and to measure the effectiveness of list scheduling. Iterative repair schedulers operate by constructing a gross approximation as an initial schedule. (The initial schedule must respect the data dependences, but not the resource constraints.) To transform the initial schedule into a valid schedule, the iterative repair framework chooses a mis-scheduled operation at random and places it in a position where it can legally execute. By restarting the algorithm multiple times, the framework can construct many distinct schedules. (This combination of randomization and restart is a powerful tool for exploring the space of schedules.) It can either gather data about the various schedules or it can simply keep the best schedule. Using different heuristics to select the next repair site produces distinct scheduling regimes.

Our experiments showed that:

1. List scheduling produces schedules of optimal length most of the time (more than ninety percent of the time).
2. A randomized version of list scheduling, run perhaps ten times, outperforms any single version that we tested.

algorithm that is easy to understand and simple to program. It produces, in a single pass, results that are equivalent to multiple applications of the classic Allen-Cocke-Kennedy algorithm (the best prior art). This algorithm has been implemented in several commercial systems. (See reference 11.)

*Dominance Calculations*—A fundamental step in building SSA form for a program involves computing dominators and dominance frontiers. We developed a simple algorithm for the dominance calculation that is, in practice, the fastest known method for this problem. Other algorithms for this problem have lower asymptotic complexity. However, careful engineering of the data structures for our algorithm let us move the cross-over point (where the asymptotically faster algorithms begin to win) out beyond graphs of thirty-thousand nodes. Since most control-flow graphs are orders of magnitude smaller than this, our algorithm outperforms the asymptotically better ones in practice. This work is described in a paper currently in review (see reference 12); the algorithms and data structures are described in Cooper and Torczon's forthcoming book.

*Building Control-flow Graphs* — In our work on partitioned register set machines, we became interested in object-code to object-code translation for Texas Instruments' C6000 series of processors. Unfortunately, those processors allow branches to be scheduled in the delay slots of other branches. Since they have five delay cycles per branch, the TI compiler for the C6000 aggressively uses this feature when it software pipelines small loops. (It stuffs the pipeline with branches, then places the single execute packet in the last delay slot.)

These programs break all previous algorithms for building control-flow graphs. For example, the compiler cannot determine where basic blocks begin and end without performing an iterative computation to discover which branches might be pending on entry to the block. We developed an algorithm that builds correct control-flow graphs for assembly code on machines with this feature. This is a necessary first step in building object-level tools for architectures like the C6000. (See reference 13.)

*Reducing the Impact of Spill Code* — Register allocation continues to be an active research issue. We developed several new strategies for reducing the amount of spill code generated by a graph-coloring register allocator. Some of them depend on specific features of embedded processors. Others have more general application. We continue to work and to publish actively in this area.

## *3.* Conclusion

This project developed a number of novel techniques for improving the code thatcompilers can generate for embedded systems.

- We worked on several aspects of instruction scheduling. This work should findpractical application in compilers for embedded systems and for conventional systems based on commodity microprocessors.
- We developed several new approaches to reducing code size, and did some experiments that used coloring to reduce the program's footprint in data memory.
- We did fundamental new work that combined techniques from other areas, such as randomization and restart, genetic algorithms, and iterative repair, to important problems in optimization and code generation.
- We developed new techniques, like OSR, that have broad application. For example, the CFG reconstruction work is a first step toward object-level optimization on DSP instruction sets — optimization for properties such as powerreduction or code size reduction.

The work on novel optimization paradigms will take longer before it has an impact on commercial practice, although we have seen some groups using techniques like our genetic algorithms to choose compiler option flags in program-specific ways. We have received additional funding to continue our investigation of these id