Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation

Bruce Childers[+], Jack W. Davidson[*], Mary Lou Soffa[+]

[+]Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{childers, soffa}@cs.pitt.edu

[*]Department of Computer Science
University of Virginia
Charlottesville, VA 22904
jwd@virginia.edu

Abstract

Over the past several decades, the compiler research community has developed a number of sophisticated and powerful algorithms for a varierty of code improvements. While there are still promising directions for particular optimizations, research on new or improved optimizations is reaching the point of diminishing returns and new approaches are needed to achieve significant performance improvements beyond traditional optimizations. In this paper, we describe a new strategy based on a continuous compilation system that constantly improves application code by applying aggressive and adaptive code optimizations at all times, from static optimization to online dynamic optimization. In this paper, we describe our general approach and process for continuous compilation of application code. We also present initial results from our research with continuous compilation. These initial results include a new prediction framework that can estimate the benefit of applying code transformations without actually doing the transformation. We also describe results that demonstrate the benefit of adaptively changing application code for embedded systems to make trade-offs between code size, performance, and power consumption.

1. Introduction

Much of the past and recent research in program optimization has focused on developing new algorithms to perform a particular optimization or transformation. Indeed, over the previous decade the compiler research community has developed sophisticated, powerful optimization algorithms for a variety of code improvements: register allocation and assignment, common subexpression elimination, partial redundancy elimination, loop optimizations (e.g., loop fusion, loop unrolling, loop interchange, etc.), code scheduling, and function inlining to name a few. While there are still avenues of promising research for particular optimizations, we are at the point where the performance gains of a new or improved optimization algorithm is usually small—an improvement of a few percent is typical.

Today's challenge for optimization research is to develop new techniques and approaches that yield performance improvements that go beyond today's small single digit improvements. In our work, we are addressing this challenge by investigating and developing an innovative framework and system for continuously and adaptively applying optimizations. Our system, the Continuous Compiler (CoCo), applies optimizations both statically at compile-time and dynamically at run-time using optimization plans developed at compile time and adapted at run time.

Rather than focusing on developing new optimization algorithms (e.g., a new register allocation algorithm, a new loop interchange algorithm) or improving existing optimizations (e.g., better coloring heuristics, better placement algorithms), our work focuses on understanding the interaction of existing optimizations and the efficacy of static and dynamic optimizations. Using this knowledge along with information about the application gathered by static analysis, profile information and monitoring, CoCo will determine how to apply a suite of optimizations so that the optimizations work in concert to yield the best improvements. The result of CoCo's analyses are compile-time and run-time plans that specify what optimizations to apply to the application, the order to apply them, and the conditions under which the optimizations should be applied to achieve maximum benefit. Applying compile-time plans assures that very high-quality code for the application is generated initially, while applying plans dynamically ensures that the executing application can adapt to changing user behavior and program behavior.

In this paper, we describe our preliminary work with CoCo. We first present our general approach and strategy to continuous compilation. We also describe a prediction framework that is included in CoCo to estimate the benefits of applying code optimizations. With this prediction framework, the code optimizer can make

0-7695-1926-1/03/$17.00 (C) 2003 IEEE

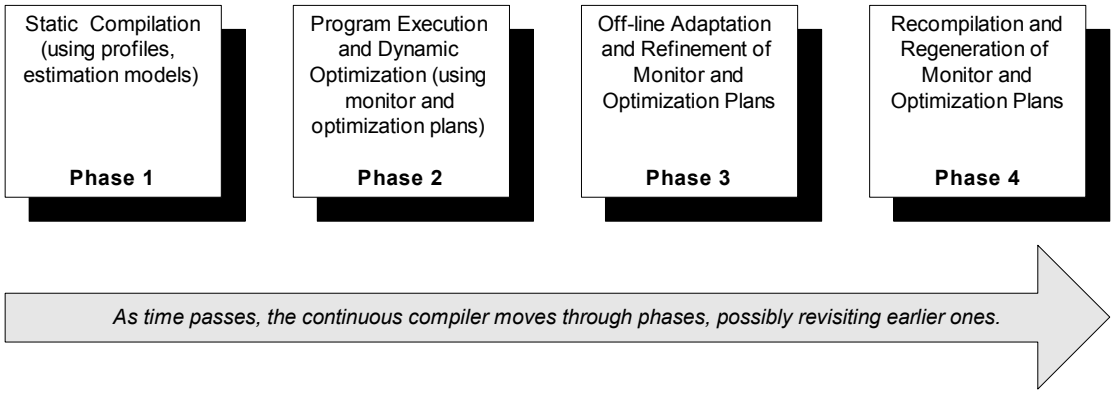

Figure 1: Phases of Continuous Compilation

decisions about when to apply optimizations and potentially in what order to apply them. We also describe work that makes trade-offs between different constraints in embedded systems. This work shows how small code footprints can be achieved while getting good performance on the ARM processor.

2. Continuous Compilation

The primary objective of our research is the development of a system that continously monitors and optimizes an application program to improve its performance. This system will monitor and observe application behavior and adapt its optimization strategies to match the current characteristics being exhibited by the application. It will also capture the continous nature of applications by not only applying optimizations statically and dynamically, but also by applying optimizations across multiple runs.

Our continuous and adaptive compiler will go through several phases as it improves an application. Figure 1 shows the compilation phases. In the **first phase,** static optimization plans are generated using the program, extended profiles from previous training runs (see section 2.3.1) and estimation models of the benefit and cost of applying particular optimizations given the target machine architecture. Using the static optimization plans, an initial optimized binary is generated for the application. In CoCo, the application of optimizations is separate from the planning of optimizations. In this first phase, dynamic optimization plans and monitor plans are also generated for use by a *dynamic optimizer*.

In the **second phase,** a dynamic optimizer applies code transformations at run-time in response to changes in program behavior as guided by the dynamic optimization plans. In our system, the planning of dynamic optimizations can be done statically to reduce the overhead of applying dynamic optimizations and to improve their effectiveness by using more powerful planning techniques (e.g., AI planning, integer linear programming) than what could be used at run-time. The application of code transformations is controlled by the dynamic optimizer based on the optimization and monitor plans. The optimization plans indicate how to transform the binary program and the monitor plans indicate what application and machine events to watch to determine when and where to apply a particular optimization plan.

A **third phase** refines and adapts the monitor and optimization plans generated by the initial compilation phase. The plans are improved using information about the actual execution of the application binary. Thus, in the third phase, more up-to-date information about the actual execution of the binary can be used to improve the initial plans. Information about past runs of the application is kept and constantly updated, building up a repository of details about the binary program's dynamic behavior that can be used to guide optimization planning. The third phase will be invoked in two ways. First, the phase can be invoked after each successive run of the application. Second, a separate *plan refinement server* can be invoked concurrently with the application's execution to refine optimization and monitor plans. For long-running applications that may take days to execute, such a separate server is critical to ensuring that the overhead of collecting detailed continuous state can be amortized immediately while the application is executing (rather than waiting until successive runs to amortized the overhead cost). The plan refinement server will inject its refined plans into the dynamic compiler's current plan portfolio whenever a newly refined plan is generated. By continuously adapting and updating the optimization and monitor plans, we expect that over time the plans will become simpler and more specific and apply "the right optimizations at the right time."

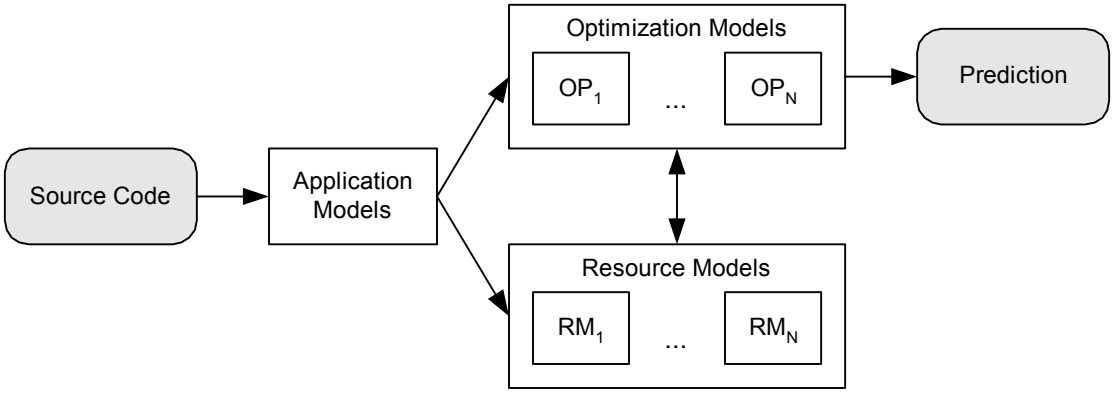

Figure 2: Prediction Framework for CoCo

Although continuously updating optimization plans may lead to steady-state behavior, there are likely to be cases where convergence is never achieved. Programs whose behavior varies with different data input sets are cases where convergence is unlikely (and, perhaps, undesirable as the system should continuously optimize the application based on the current data set). Furthermore, it is advantageous to statically reapply optimizations to the binary as better and more complete information is collected about a program's behavior. To address the problem of convergence and to capture better information in the statically optimized program, a **fourth phase** will fully recompile and optimize the application. This phase uses all information about the application—estimation models, profiles, and details about actual program runs—to optimize the program and generate monitor and dynamic optimization plans. This fourth phase will recompile the application offline and apply the full weight of its optimization and planning techniques.

3. Predicting the Impact of Optimizations

Although code improvement optimizations have been applied by optimizing compilers for almost 40 years, many performance problems remain. In particular, when applying optimizations, a number of decisions are made using fixed strategies, such as always applying an optimization if it is applicable, applying optimizations in a fixed order, and assuming a fixed configuration for optimizations. While it is widely recognized that these fixed strategies or plans may not be the most appropriate for producing high-quality code, there are no practical and automatic strategies that do otherwise. It is well known that optimizations may degrade performance in certain circumstances, but there is no analytic way to determine when this might happen and choose not to apply an optimization to get better performance. Also, there is no effective way to determine the best order of applying optimizations, although the order can have an impact on performance as optimizations can interact with one another by creating or destroying the potential for further optimizations. These problems have become particularly important in recent years with the tremendous growth in cost-sensitive embedded systems, where achieving the very best performance is paramount. What is needed is an effective way to uniformly express the variations among optimizations, their impact on different objectives, and their interactions to make predictions about when it is beneficial to apply an optimization.

3.1. Prediction Framework

Our approach in CoCo is to develop a framework that lets us predict the impact of applying an optimization without actually applying it. In this way, decisions can be made about what optimizations to apply and in what order to apply them in a particular code context. CoCo's prediction framework consist of three types of models: optimization models, code application models, and resource models. The structure of our prediction framework, as shown in Figure 2, includes (1) optimization models that represent the characteristics of the optimizations in terms of how they will impact an objective, both qualitatively and quantitatively, (2) resource models that parameterize the target machine configuration, and (3) application code models that abstract information about the application. By integrating the models, a "benefit" value is produced that represents the benefit of applying an optimization in a code context with the objective represented by the resource.

Using the prediction framework, decisions can be made about whether it is beneficial to apply an optimization given a particular code context. The optimization models (with different configurations or different optimizations) can be combined and we can predict the benefit of combining optimizations rather than applying them one at a time. When more than one optimization

can be applied in a code segment, the framework can be used to predict the best one to apply. Lastly, the prediction value can be used as an objective function when using search techniques such as genetic algorithms and AI planning for developing optimization plans.

3.2. Prediction for Loop Optimizations

As the disparity between processor and main memory speed increases by approximately 50 percent per year, the use of caches with high hit rates has become critical for performance. Data caches are designed to exploit locality, and naturally they work best for programs that have high locality. Some optimizations are designed to improve cache performance by rearranging the code to have better locality. However, other optimizations are not designed specifically for this purpose and may negatively impact cache performance and the overall performance.

We have used CoCo's prediction framework to estimate the impact of applying optimizations on data cache performance. Since loop behavior tends to dominate cache performance, we are initially focusing on loop optimizations. Our prediction framework is tailored to represent the characteristics of loops and optimizations that impact cache performance. We also use a model of cache behavior for the array referencing patterns that estimates the cache cost of executing a code segment. After determining the impact of an optimization on cache performance with the models, the code optimizer can decide whether it is beneficial to apply the optimization. Below we briefly describe each of our models. More detail about the models are in Zhao, Childers, and Soffa [20].

3.2.1. Code Model

To predict the impact of optimizations on cache performance, we need to express code characteristics that affect the cache. For the optimizations that we are initially considering, the code model represents the loop's header and the sequence of array references in a loop body. The model captures several aspects of a loop nest: (1) the loop header with its lower and upper bounds and iteration step; (2) all array references and their type (includes read and writes and their affine expression); and (3) an array reference sequence that consists of all array references in a loop body in the order that they appear in the intermediate code. The model can also capture a loop nest sequence that represents the order of loops that they appear in the input code. Importantly, the code model is a simple notation that formally describes and abstracts the details of a loop nest that are needed by loop optimization. The code model also has all necessary information needed to estimate the cache performance of a loop.

3.2.2. Optimization Models

To represent the impact of loop optimizations on cache performance, we model the transformation of an unoptimized loop nest to an optimized loop nest. We have optimization models for loop interchange, loop unrolling, loop tiling, loop reversal, loop fusion, and loop distribution. In all of these models, the transformation is represented by a sequence of functions that effect the various aspects of a loop's representation. For example, in loop reversal, the direction in which a loop traverses its iteration range is reversed. Our optimization models have an impact function that describes how the loop is changed. In the case of loop reversal, for a given loop code model, a new loop code model is generated in which the loop header has been changed to indicate the new traversal order. These models are unique in that they do not apply the code transformation to the actual code. Instead, they predict how the code is changed, and the predicted changes are embodied in the new code model produced by the optimization models.

3.2.3. Cache Model

With the code and optimization models, we can accurately reflect and predict how the application code is transformed by a loop optimization. To determine whether or not the loop optimization had a benefit in terms of its cache performance, we use a separate cache model. The cache model takes the loop code model and models the effect that the array references within the loop have on the cache. The model indicates how a given reference pattern effects both the misses and hits. Thus, to measure whether or not an optimization improves cache performance, we can use the cache model to determine whether the number of cache misses was increased or decreased. This estimation is done by comparing the cache performance with the code model before the loop optimization model is applied and with the code model after the loop optimization model is applied.

3.2.4. Preliminary Results

To investigate the accuracy of our framework toward predicting the impact of loop optimizations on cache misses, we implemented our models and tested them with several benchmark loops. The benchmarks came from the PERFECT suite and other researchers [11]. For this initial study, we looked at benchmarks with a

Benchmark	Interchange	Tiling	Reversal	Unrolling	Fusion	Distribution
alv	100%	100%	97.4%	100%	N/A	100%
irkernel	98.7%	100%	93.4%	100%	N/A	N/A
lgsi	100%	100%	82%	100%	N/A	N/A
smsi	100%	100%	86.8%	100%	N/A	N/A
srsi	100%	100%	86.8%	100%	N/A	N/A
tfsi	100%	97.4%	100%	100%	N/A	N/A
tomcat3	98.7%	92.1%	93.4%	100%	100%	N/A

Table 1: Model Prediction Accuracy

single loop nest. The benchmarks include *alv, irkernel, lgsi, smsi, srsi, tfsi,* and *tomcat3*.

Using these benchmarks, we validated our optimization and cache models. A tool was developed that takes a loop nest and based on our models computes the number of cache misses for both the orginal and optimized loop nests. The difference in the cache misses is used to predict the impact of a loop optimization on cache performance. To validate the predictions, we ran the original and optimized loop nests with the SimpleScalar sim-cache simulator [3] to measure the impact on cache performance. The simulator was configured with a 1 kilobyte direct-mapped data cache with 32-byte blocks. We can scale the input data set sizes relative to this small cache to simulate different ratios of working set size to cache size. With this cache configuration, we compare our predictions against the simulation results. If an optimization improves performance with the simulation results, and our model predicted that the optimization should be applied, then we consider that to be a correct prediction. If the simulation results do not match our predicted results, then we consider that to be a misprediction. We computed a prediction accuracy for our models that captures how often our model gives the correct predictions.

Table 1 shows the prediction accuracy of our models for each benchmark and loop optimization considered. The prediction accuracies in the table are averages across a range of trip counts for each benchmark. The trip count was varied from 50 to 200 for each benchmark to simulate different ratios of working set size to cache size and to determine whether our model can accurately reflect different loop configurations. The prediction accuracy of our framework in determining when to apply optimizations is 97.2% on average. The prediction accuracy for loop reversal on *lgsi* is 82%. This lower prediction accuracy is because for most trip counts, the cache miss reduction of loop reversal is so small (the reduction is just one or two misses) that our model can not predict the benefit. Instead, our model does not apply loop reversal in these cases when the miss reduction is so small. Not applying reversal in this case does no harm since the relative improvement of applying reversal is minimal and can be ignored.

4. Trade-off of Code Size versus Performance

Many embedded and mobile applications have stringent requirements in terms of performance, code size, and power consumption. As a prime example, consider the software for cellular telephones. Here quality of service considerations requires maximizing performance, cost considerations requires minimizing code size, and battery life requires minimizing power consumption. We are exploring the use of adaptive compilation as a means of meeting the cross-cutting demands of performance, code size, and power consumption in systems where changing execution profiles make traditional static techniques ineffectual.

Our preliminary work has focused on the exploitation of the ARM7TMDI architecture The ARM7TDMI architecture was designed for cost-sensitive applications that require a balance between performance, code size, and power consumption.[1] To allow trade offs between performance, code size, and power, the ARM7TDMI architecture provides two instruction sets—ARM and Thumb—within a single CPU. The ARM instruction set is a 32-bit RISC-like instruction set. It is used when high-performance and low power consumption is required. The Thumb instruction set is a subset of the ARM instruction set and each instruction is encoded in 16-bits. Thus a 32-bit word can hold two Thumb instructions. Published reports indicate that Thumb programs are typical 30% smaller than equivalent ARM programs. This reduction in code space

1. Our measurements of currently available implementations of the ARM/Thumb architecture, indicate that improving performance also reduces power consumption. For other architectures or other implementations of the ARM this relationship between performance and power consumption might not hold. Consequently, we will continue to treat performance, code size, and power consumption as distinct constraints.

comes at a price—Thumb programs execute more instructions and consequently use more power. As a result, the Thumb instruction set is used when compact code is required and performance and power are lesser considerations.

Current compilation tools for the ARM7TMDI platform provide a mechanism for mixed use of ARM and Thumb instruction sets at the module level. The instruction set to be used for a whole source file is specified when the file is compiled, and different modules compiled into different instruction sets can be linked together to build a binary. While this approach gives the software developer some control over balancing performance, code size, and power consumption, it is a coarse-grained approach. In this research, we are developing a more fine-grained approach that is applied dynamically to exploit the trade-off relationship between the two instruction sets. With a fine-grained approach, within a module performance critical sections will be compiled into ARM code, while portions of the code that are seldom executed (e.g., error checking and recovery code) will be compiled to Thumb code. While our overall goal is a dynamic approach, we are first focusing on developing static techniques that use offline execution profiles. Using the experience gained, we will extend the approach so that applications with dynamically changing execution profiles and resource requirements can be handled.

4.1. Fine-grained code generation for performance/space/power trade offs

To achieve more flexibility in making trade offs between performance, code density, and power consumption, we need to be able to switch between the ARM and Thumb instruction sets at a finer-grained level. Conceptually, we need a mechanism that automatically uses ARM code for frequently executed sections of code, and that uses Thumb code for infrequently executed code sections. If applied properly, this should lead to code that achieves performance that is nearly the same as a program that has been compiled entirely into ARM code, yet is nearly as compact as a program that has been compiled entirely into Thumb code. Indeed, this is the goal of this research—produce code that is a compact as a Thumb program yet achieves performance equivalent to an ARM program.

One of the first tasks of this research is to determine how fine-grained of an approach is feasible. One level of granularity is to use ARM code for loops with deep nest levels and high iteration counts, and Thumb code for all other parts of the program. We expect this approach to be too coarse to achieve our goal. For example, consider a loop that has two alternative paths in its body as shown in flow graph of Figure 3. Suppose blocks `C` and `D` are frequently executed while blocks `E` and `F` are not. Such a situation might arise if blocks `E` and `F` are error handling code. In most circumstances blocks `E` and `F` are never executed. In this case, the best code would be produced by compiling blocks `A`, `B`, `C`, `D`, and `G` into ARM code and blocks `E` and `F` into Thumb code.

A more fine-grained approach would be at the basic-block level. Assuming profile data was available to indicate which blocks were heavily used, this approach would address the example given in Figure 3. However, we cannot base our decision as to which instruction set to use solely on the execution frequency of the blocks. For example, consider two blocks, `A` and `B`, which are executed 1000 and 500 times, respectively. If we decide which blocks are to be compiled to ARM or Thumb code based on execution counts only, block `A` would have a higher probability than block `B` of being compiled to ARM code. However, if it happens that block `A` is a very small block (say 3 instructions), and block `B` is a large block (say fifty instructions), then it would be best to favor `B` over `A` as a candidate for being compiled to ARM code (both in terms of code size and execution time).

The previous example indicates that we must have an accurate cost-benefit model to base our code generation decisions. Another factor that must be considered is the overhead introduced by the mode switch instructions necessary to transition between machine models.[1] It is crucial to the success of the fine-grained approach that we minimize the overhead of mode switches. Such effects must be considered in the cost-benefit model employed.

Figure 2 contains a diagram that illustrates the system we are building. Since there is no one-to-one correspondence in control flow structures between the same program compiled into ARM code and Thumb code (the ARM has predicated execution, while the Thumb does not), we first compile the entire program into Thumb instructions. We enumerate a number of candidate paths for translation to ARM code, calculate the cost (in terms of code size) of translating the blocks on this path as well as the benefit (in terms of execution time) from translating those blocks, and select one path at a time that is expected to give the most savings in execution time for a small increase in code size.

For now we assume that the candidate paths are acyclic subpaths that begin at the entry node of a loop and

1. To switch instruction sets requires executing a special mode switch instruction.

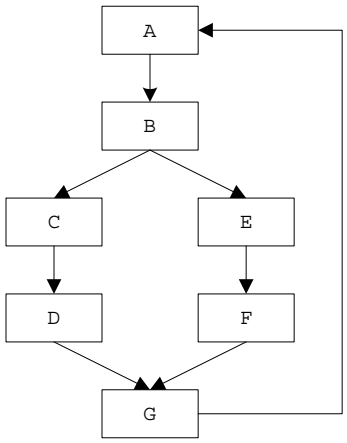

Figure 3: Flow graph of candidate loop for mixed-mode code generation.

end at the exit node of that loop. Let *V(p)* denote the set of nodes (basic blocks) on a candidate path *p*, and let *f(v)* denote the execution frequency of node *v*. When we choose one path from the set of candidate paths, we can identify the set of edges along which the processors execution mode must be switched. that is, the edges on which mode switch instructions must be inserted are the ones that connect a node belonging to the selected path and another node that is not (yet) selected for translation. Note that when either the source or the destination of two (or more) edges is the same, we need only generate one set of mode switch instructions for those two (or more) edges. Let *E*(p)* denote the set of edges along which mode switch instructions should be inserted, and let *f(e)* denote the execution frequency of edge *e*. Then the benefit of selecting a path *p* can be calculated by

$$b(p) = \sum_{v \in V(p)} (f(v)) \times (t_T(v) - t_A(v))) - o_t \times \sum_{e \in E^*(p)} f(e)$$

where $t_T(v)$ and $t_A(v)$ denote the estimated execution time of block *v* compiled into Thumb instructions and ARM instructions, respectively. The cycle time overhead of one mode switch instruction is denoted by o_t. Intuitively, the benefit function $b(p)$ gives the expected savings in the execution time by translating the blocks on path *p* into ARM instructions. On the other hand, the cost of selecting a path p can be calculated by

$$c(p) = \sum_{v \in V(p)} (s_A(v) - s_T(v)) + s_s \times |E^*(p)|$$

where $s_A(v)$ and $S_T(v)$ denote the code size of block *v* compiled into ARM and Thumb instructions, respectively, while o_s denotes the code size overhead for one set of mode switch instructions.

4.2. Preliminary results

A preliminary implementation has been developed to test both the feasibility and the effectiveness of this approach for addressing performance/size/power trade offs. The implementation is preliminary as some of the phases are not fully implemented. For example, we manually generate the necessary profiles information manually using a number of tools. Furthermore, the code quality of the ARM code for the mixed binaries could be improved. For example, the translation from Thumb to ARM code currently does not take advantage of the additional registers available when running in ARM mode.

Figure 4 compares the time/space trade offs of the approach for three benchmarks from the MediaBench. All numbers have been normalized to the ARM measurements. Recall our goal is to have the mixed binary have nearly the same code size as the Thumb binary, yet have the mixed binaries achieve nearly the same execution performance as the ARM binary.

For the *crc* benchmark, our goal is achieved—the mixed binary size is nearly the same as the Thumb binary, and the code actual runs faster than the ARM binary. The improved performance is because of better cache behavior with the compressed binary. For the *sha* benchmark, again the mixed binary is nearly the same as the Thumb binary, however the execution time is 50 percent greater than the ARM binary. We believe that with an improved register allocator, the execution time can be reduced to near that of the ARM. The results for the *dijkstra* benchmark are similar to the results for *sha*.

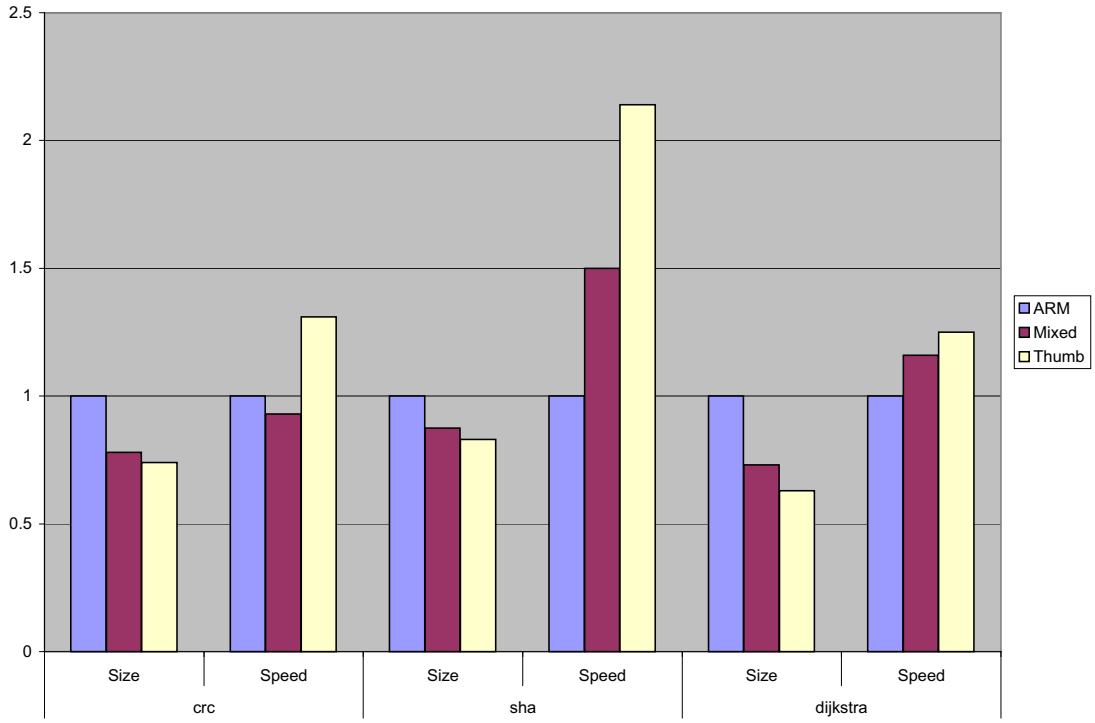

Figure 4: Size and speed comparison of pure ARM binaries, mixed binaries, and pure Thumb binaries

4.3. Dynamic fine-grained code generation for performance/space/power trade offs

The preliminary steps outlined in the previous section used offline profiles to determine which program paths should be compile to ARM code and which should be compiled to Thumb code. While a necessary first step, there are many types of applications where offline profiles are inadequate. These applications are embedded in systems where there are changing demands and resources (e.g., handheld devices, mobile sensors, real-time data acquisition, etc.). In this type of dynamic application, over the lifetime of a program's execution, certain program paths may be "hot" while at other times they may be "cold."

To handle applications with dynamically changing resource requirements requires that we adapt our compilation strategy so that it can be applied dynamically as the program is executing. Our approach is to use analysis of offline profiles to build code generation plans that are executed at runtime as the program moves through different phases. (Such transitions are sometimes called phase shifts). There are many research challenges that must be addressed to realize this type of dynamic fine-grained code generation. Some of the major challenges are:

- to develop estimation models that take into account the cost of dynamic code generation,
- to develop tools that develop code generation plans that can be applied dynamically,
- to develop low-overhead profiling techniques that are suitable for supplying information to a dynamic code generator, and
- to develop a fast, retargetable code generator for use in dynamic code generation.

We believe the techniques and algorithms that are developed for this research will contribute significantly to the development and deployment of embedded and mobile applications that have stringent requirements in terms of performance, code size, and power consumption.

5. Related Work

Dynamic compilation schemes choose portions of the code to compile/recompile and then compile the code with a set of optimizations. Early strategies relied on static strategies [4,5,9,13,19]. Later efforts chose the compilation targets dynamically, attempting to focus on hot spots [6,14]. Recent work in this area has included adaptive online feedback-directed optimization [1]. Another form of dynamic compilation is selective compilation [10,15,17] which is a staged form of compila-

tion. Dynamic optimization systems separates the task of compilation from optimization with the optimizations occurring entirely at run time and not requiring any user assistance. Dynamo is a dynamic optimization system that both interprets and directly executed native code [2]. Another proposed dynamic optimization system is the continuous program optimization architecture designed for Oberon that embodies up-to-the-minute profiling information and enables continuous optimization [12]. Other work includes Plezbert and Cytron's continuous compiler that overlaps the compilation phase with program interpretation and native execution in a Java virtual machine [12]. This work is similar in the respect that a separate compilation phase is done concurrently with program execution. However, our work is more aggressive and adaptive in that our compilation phases include the full spectrum of static compilation, dynamic application of optimizations and re-optimization with profile information and optimization plans. The interactions of code optimizations play an important role in the develop of CoCo. Early research on code interactions developed a language for specifying optimizations and identified the interactions among a set of traditional optimizations using formal techniques as well as experimental techniques [18]. Recently, research efforts have been exploring the use of search techniques to identify the interactions as well as the value of determining the best order to apply optimizations at different portions of the code [7,8].

6. Summary

In this paper, we described a new approach to aggressive and adaptive code transformation based on a continous compilation system. We presented preliminary work towards developing a continuous compilation system, including a framework for predicting the impact of optimizations on machine resources and performance. In the paper, we showed that the prediction framework has high accuracy for loop optimizations and their interaction with a processor's data cache. We also described work that showed the benefit of making trade-offs in constraints for embedded systems. The work demonstrated that good performance can be achieved in embedded codes while keeping code footprint small by using a combination of compact (Thumb) and normal instructions for the ARM processor.

7. Acknowledgements

The work on predicting the impact of optimizations is the Ph.D. research of Min Zhao who is a student at the University of Pittsburgh. The research on fine-grained code generation for performance/space/power trade offs is the PhD research of Sheayun Lee of the Seoul National University who was a visiting scholar as the University of Virginia in 2002. This work was supported in part by National Science Foundation grants ACI–0203945 and ACI–0203956.

8. References

[1] M. Arnold, S. Fink, V. Sarkar and P. Sweeney, "A comparative study of static and profile-based heuristics for inlining", *ACM Workshop on Dynamic and Adaptive Optimization*, pp. 52–64, 2000.

[2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *ACM Conf. on Programming Language Design and Implementation*, pp. 1–12, 2000.

[3] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0", University of Wisconsin Computer Science Technical Report 1342, June 1997.

[4] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar et al., "The Jalapeno Dynamic Optimizing Compiler for Java", *Proc. of Java '99*, pp. 129–141, 1999.

[5] C. Chambers and D. Ungar, "Making pure object-oriented languages practical", *6th Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 1–15, 1991.

[6] M. Cierniak, G-Y Lueh, and J.M. Stichnoth, "Practicing JUDO: Java under dynamic optimizations", *ACM Conf. on Programming Language Design and Implementation*, 2000.

[7] K. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century", *Los Alamos Computer Science Institute's 2001 Symposium*, 2001.

[8] K. Cooper, P. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms", *Workshop on Lang., Compilers, and Tools for Embedded Systems*, 1999.

[9] P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system", *11th ACM Symp. on Principles of Programming Languages*, pp. 297–302, 1984.

[10] B. Grant, M. Philpose, M. Mock, C. Chambers, and S. Eggers, "An evaluation of staged run-time optimizations in DyC", *ACM Conf. on Programming Language Design and Implementation*, 1999.

[11] J.S. Hu, M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving cache locality by a combination of loop and data transformations", *IEEE Trans. on Computers*, Vol. 48, No. 2, February 1999.

[12] T. Kistler and M. Franz, "Continuous program optimization: Design and evaluation", *IEEE Trans. on Computers*, pp. 549–566, June 2001.

[13] A. Krall, "Efficient Java VM just-in-time compilation", *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pp.202–212, 1998.

[14] S. Meloan, "The Java HotSpot performance engine: An in-depth look", *Sun Java Developer Connection*, 1999.

[15] R. Marlet, C. Consel, and P. Boinot, “Efficient incremental run-time specialization for free”, *ACM Conf. on Porgramming Language Design and Implementation*, pp. 281–292, 1999.

[16] M. Plezbert and K. Cytron, “Does ‘just in time’ = ‘better late than never’?”, *ACM Symp. on Principles of Programming Languages*, 1997.

[17] M. Poletto, W. Hsieh, D. Engler, and M. Kasshoek, “‘C and tcc: a language and compiler for dynamic code generation”, *ACM Trans. on Programming Languages and Systems*, 1999.

[18] D. Whitfield and M. L. Soffa, “An approach to ordering optimizing transformations”, *ACM Symp. on Principles and Practice of Parallel Programming*, pp. 137–147, 1990.

[19] B-S Yang, S-M Moon, S. Park et al, “LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation”, *Int’l. Conf. on Parallel Architectures and Compilation Techniques*, 1999.

[20] M. Zhao, B. Childers, and M. L. Soffa, “FPO: A framework for predicting the impact of optimizations”, technical report number TR–02–102, Department of Computer Science, University of Pittsburgh, November 2002.