A Customizable Compiler Framework for Embedded Systems [*]

Ashok Halambi ahalambi@ics.uci.edu
Aviral Shrivastava aviral@ics.uci.edu
Nikil Dutt dutt@ics.uci.edu
Alex Nicolau nicolau@ics.uci.edu

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

Extended Abstract submitted to SCOPES 2001

[*]This work was partially supported by grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348).

1 Introduction

Embedded Systems containing programmable components (i.e. processor-memory based systems) are used to provide for the dual needs of short design times, and higher design flexibility. Designers may reuse preverified processor, memory Intellectual Property (IP) blocks to shorten the design times. In addition, the programmability of such systems allows for modifications late in the design stage and also allows for easier upgrading as compared to non-programmable ASIC solutions. Previously, software for such programmable embedded systems was developed using assembly-level languages. However, the high degree of software content for newer designs and shrinking time-to-market cycles has resulted in migration to a high-level language (such as C, C++, Java) based software development environment.

To effectively explore the processor-memory design space and develop software in a high-level language, the designer requires a high quality software toolkit (primarily a highly optimizing compiler and cycle-accurate simulator). Compilers for embedded systems have been the focus of several research efforts [11] recently. A promising approach to automatic compiler generation is the "retargetable compiler" approach. A compiler is classified as retargetable if it can be adapted to generate code for different target processors with significant reuse of the compiler source code. Retargetability is typically achieved by providing target machine information (in an Architecture Description Language – ADL) as input to the compiler along with the program corresponding to the application.

The complexity in retargeting the compiler depends on the range and features of target processors it supports and also on its optimizing capability. Embedded processors typically have irregular pipeline, datapath, and memory structures. Conventional compiler techniques do not work well under these constraints. Therefore, recent research has concentrated on techniques that target embedded processor characteristics. Examples of such techniques include optimized memory address computation, SIMD (Single Instruction, Multiple Data) optimization, data placement etc. However, the problem of integrating these techniques with the traditional compiler phases (including machine-independent optimizations, code selection, instruction scheduling, etc.) still remains. The "optimal" ordering of all these techniques is very much dependent on the processor architecture and application characteristics. Furthermore, the ordering is also influenced by the optimization criteria (such as power, cost, performance, etc.). In this paper we present **Transmutations**, a compiler customization framework that integrates the various embedded system compilation techniques, and allows for dynamic ordering between the compiler phases. The Transmutations framework is a part of the **EXPRESS** retargetable compiler for embedded systems.

2 Related Work

The problem of generating efficient software toolkits for embedded systems has been the focus of several recent research activities. [6] contains a survey of some of the recent efforts in automatically generating the software toolkit from a specification of the system in an Architecture Description Language (ADL). In this paper we focus on the problem of increasing the efficiency of the compiler by customizing it for the given application and architecture (and also the design goals).

Previous research in the area of compilers for embedded systems has resulted in the development of various techniques to increase the efficiency of the compiler for performance, code-size and power goals. Also, there have been some projects that aim to incorporate these individual techniques in a complete compiler flow for a (narrow) range of processors. The CodeSyn [10] project demonstrated a compiler for a limited embedded processor class with irregular architectures. CHESS [9] is a retargetable code generation environment for fixed-point DSP processors. CHESS uses the nML [3] ADL to achieve retargetability. The AVIV [7] compiler, using the ISDL [4] ADL, produces machine code optimized for size. The MSSQ and RECORD compilers use the MIMOLA [1] ADL to achieve retargetability. MMSQ is able to produce microcode for a large range of datapath architectures, but suffers from low code quality. The RECORD compiler, however, targets mainly DSP architectures.

The quality of generated code is heavily influenced by the ordering of the compiler optimizations (also known as phases). Most compilers rely on a predetermined ordering of the phases. However, as these phases are mutually dependent and may adversely affect each other, this approach is sub-optimal when retargeting the compiler for a wide range of processors and applications. Simultaneous execution of all the phases in order to avoid restricting the solution space is not practical because of the large number of optimizations. An example of this is the Integer Linear Programming based approach proposed in [17], which suffers from extremely high runtime requirements.

In recent years, techniques that integrate some optimizations in order to mitigate the phase ordering problem have been reported. For example, instruction scheduling and register allocation have been integrated in [13], [2]. However, most such techniques have only considered RISC like architectures with homogeneous register files.

The AVIV compiler attempts to solve the phase ordering problem by performing a heuristic branch-and-bound step that executes resource allocation/assignment, operation grouping, and scheduling concurrently. The CHESS compiler uses data routing as a technique to simultaneously solve the problems of code selection and register allocation.

Mutation Scheduling (MS) [14] integrates code selection and register allocation into instruction scheduling by "adapting"the computation of values to conform to varying resource constraints and availability. As the problems are NP-hard, MS depends on heuristic guidance to limit the search space. However, MS only integrated the traditional compiler phases and mainly considered homogeneous architectures. Transmutations incorporates MS and further, as explained in Section 3.2, provides for changing the ordering of other embedded systems optimizations (such as memory optimizations, SIMD, etc.).

3 EXPRESS Retargetable Compiler

In order to effectively compile for modern embedded processor architectures, the compiler needs to incorporate a large set of optimizations. These optimizations may target different aspects of the architecture (e.g. conditional instructions) or the application (e.g. SIMD). The EXPRESS retargetable compiler adopts a "toolbox" approach to incorporating both traditional and embedded systems specific compiler optimizations. However, in such an approach, the phase ordering between the optimizations has a huge impact on the quality of generated code. The problem of determining the 'optimal' phase ordering is further complicated by the fact that most applications have regions with different characteristics (e.g. loop regions, if-block regions, etc) which require different optimization orderings. Statically determined phase orderings may not be able to satisfy the stringent constraints of performance, power, code size, etc. The compiler requires the ability to dynamically determine, based on the region(s) of interest, the best ordering of optimizations. The EXPRESS compiler incorporates Transmutations, an approach that attempts to provide for dynamic ordering of the phases based on the program characteristics and available resources. In the following, we first present a brief overview of the EXPRESS compiler and then describe the Transmutations framework in detail.

3.1 EXPRESS

EXPRESS is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. EXPRESS uses the EXPRESSION ADL [5] to retarget itself to a wide class of processor architectures and memory systems. Figure 1 shows the EXPRESS compiler along with the Transmutations framework. The inputs to EXPRESS are the application specified in C, and the processor architecture specified in EXPRESSION. The front-end is GCC based and performs some of conventional optimizations. The core transformations in EXPRESS include **RDLP** [15] – a loop pipelining technique, **TiPS** : Trailblazing Percolation Scheduling [12] – a speculative code motion technique, Instruction Selection, Register Allocation and If-Conversion – a technique for architectures with predicated Instruction Sets. The back-end generates assembly code for the processor ISA. We use SIMPRESS [8], a cycle accurate, structural simulator to analyze the performance of generated code.

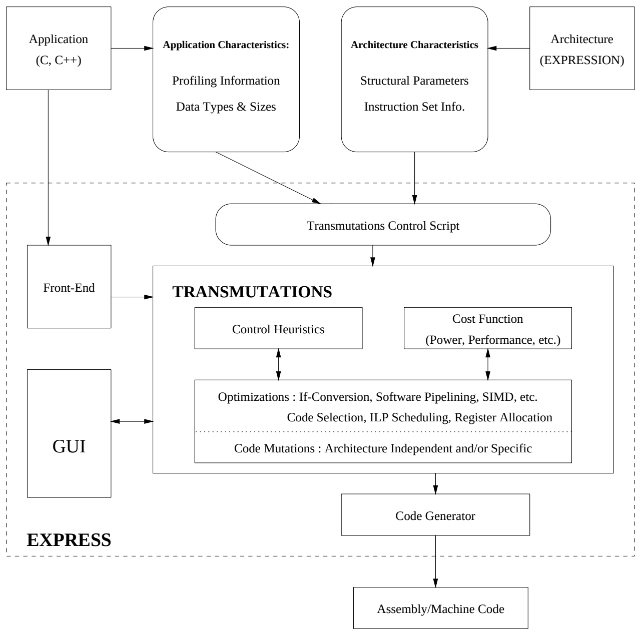
Application
(C, C++)
Application Characteristics:
Profiling Information
Data Types & Sizes
Architecture Characteristics
Structural Parameters
Instruction Set Info.
Architecture
(EXPRESSION)
Transmutations Control Script
Front-End
TRANSMUTATIONS
Control Heuristics
Cost Function
(Power, Performance, etc.)
Optimizations : If-Conversion, Software Pipelining, SIMD, etc.
Code Selection, ILP Scheduling, Register Allocation
Code Mutations : Architecture Independent and/or Specific
GUI
Code Generator
EXPRESS
Assembly/Machine Code

Figure 1. **EXPRESS** Framework

3.2 Transmutations Framework

Mutation Scheduling (MS) attempts to couple the phases of Instruction Selection and Register Allocation into the ILP Scheduler by providing semantically equivalent computations of program values which have different resource usage patterns. MS adopts a "local" view of the search space by only providing for mutations of values through algebraic transformations. Transmutations incorporates MS and also provides a framework for phase-ordering between all transformations, including the traditional compiler optimizations and memory optimizations. Furthermore, Transmutations attempts to customize the compiler for a wide variety of architecture styles including RISC, VLIW and Superscalar.

Through the Transmutations framework, the EXPRESS compiler is able to dynamically "adapt" both the program code and the order of transformations based on the resource availability and program region characteristics. Examples of code mutations [14] possible in EXPRESS include architecture-independent mutations such as Tree Height Reduction (THR), and architecture-specific mutations such as Strength Reduction, Synonyms etc. Each code mutation has a cost function that determines its impact on performance, code size, memory access etc. The heuristics in the Transmutations framework use this information in order to assign priorities for the mutations based on the resource availability. The heuristics also determine the ordering of the compiler phases. Transmutations also allows for user-guidance through the Transmutations Control Script as shown in Figure 1. In the script, the user can specify new mutation transformations, strategies for phase orderings, and also specify the heuristics and cost functions. This allows the user to customize the compiler based on the application and processor domain.

4 Experiments

We conducted some experiments to demonstrate the importance of customizing the optimization flow based on the architecture, the application and the design goals. While EXPRESS supports various phase orderings of all the optimizations, in this paper we focus on two very important transformations : If-Conversion and Speculative code motion. We performed experiments with ordering these two transformations along with other conventional optimizations such as Dead Code Removal.

If-Conversion is a technique for converting control dependent operations into conditionally executed operations. This technique is very useful for predicated architectures which allow for conditional execution of operations based on the value of a Boolean source operand, referred to as the guarding predicate. If-conversion eliminates the branch instruction and converts control dependencies to data dependencies. As a result, the true and the false branch basic-blocks of a if-statement get merged into a larger basic-block with greater parallelism. However, If-Conversion may increase the number of instructions that get executed dynamically because instructions from both paths of the branch get executed. Furthermore, depending on the architecture, If-Conversion may also increase the code size. We consider two architectural choices in supporting predication : restricted (also known as Partial Predication) and aggressive (also known as Full Predication). In the restricted version only a limited set of predicated instructions are available in the ISA. In our experiments, the Partially Predicated architecture only supports conditional moves, while the Fully Predicated architecture supports the guarded execution of all operations.

The speculative code motion technique in EXPRESS is based on the TiPS (Trailblazing Percolation Scheduling) technique developed at UC, Irvine. TiPS is a beyond basic block scheduling technique that attempts to extract the maximum ILP available in the application. TiPS has been proven to extract good performance while limiting the code size explosion associated with most speculative code motion techniques.

The simulation architecture platform is a MIPS variant with 2 ALUs, a Float Unit, a Branch and a Load/Store unit. It accepts the MIPS ISA, and also supports both Partial and Full Predication. We assume that the latency of each operation is 1 cycle and the branch mis-prediction penalty is 4 cycles. We chose the MIPS as our experimental platform because of the wide variety of architecture styles with the same ISA. The MIPS R4000 is RISC, while the MIPS R10000 is superscalar with ILP and the R12000 supports Partial Predication with conditional moves. The demonstrator benchmarks are control-intensive kernels with nested-if structures. These benchmarks have been chosen from the Trimaran [16] suite, and also from scientific computation benchmarks.

	Partial Pred.					
	Pred.		Spec.		Pred. & Spec.	
Bench	Spdup	Size	Spdup	Size	Spdup	Size
dag	1.00	1.00	1.4	0.94	1.26	1.00
ifthen	1.00	1.00	1.22	0.95	1.28	1.00
hyper	1.00	1.00	0.98	1.00	1.17	1.00
minloc	1.00	1.00	1.12	1.10	1.32	1.00

Table 1. Phase Ordering for Partial Predication

Table 1 presents the speedup and code size obtained on the Partial Predication model. The second and third columns present the normalized speedup and code size (respectively) after If-Conversion alone. The next two columns present the speedup and code size for

Speculation alone as compared to If-Conversion. The last two columns present the speedup and code size obtained by performing Speculation after If-Conversion. As can be seen from the table, the optimal ordering of these transformations is dependent on the application and also the compilation goals. For example, for the *ifthen* benchmark, Speculation alone performs comparably to Predication followed by Speculation and at the same time has lower code size. This is because, in the partial predication model, a lot of conditional moves are inserted during If-Conversion. This contributes both to the code size and to reduced parallelism. However, for the *minloc* benchmark, Speculation suffers from code size explosion and lower performance as compared to Predication and Speculation. There is no difference in code size with Predication alone as compared to Predication and Speculation because Predication converts Ifs to straight line code and thus prevents code explosion during the Speculation phase.

	Full Pred.					
	Pred.		Spec.		Pred. & Spec.	
Bench	Spdup	Size	Spdup	Size	Spdup	Size
dag	1.00	1.00	1.26	1.09	1.15	1.00
ifthen	1.00	1.00	1.11	1.08	1.32	1.00
hyper	1.00	1.00	0.98	1.06	1.18	1.00
minloc	1.00	1.00	1.15	1.08	1.23	1.05

Table 2. Phase Ordering for Full Predication

Table 2 presents the speedup and code size obtained on the Full Predication model. We discern some slight variations to the speedup and code size numbers as compared to the Partial Predication model. In particular, Speculation alone always results in a code increase of 6 – 10% as compared to Predication alone. This is because If-Conversion does not insert any conditional moves and instead chooses to convert the conditional operations into their predicated counterparts. Once again, however, the optimal ordering of these phases is very much dependent on the application and the design goal. The EXPRESS customizable compiler, which allows for dynamic ordering of the phases, is very useful and can provide significant advantages over predetermined static phase orderings.

5 Summary

Software generation for embedded systems is very complex because of the wide variety of architectural styles, diverse application domains and design goals. In this paper we present a customizable retargetable compiler framework that determines the phase-ordering between transformations dynamically based on the resource availability and the program region characteristics. We present some experiments with ordering If-Conversion – a predicated execution technique, and Speculative code motion. The results indicate that flexibility in the ordering of the transformations is important while compiling for embedded systems. Our future work includes performing experiments exploring the various phase orderings, and also incorporating more transformations into the EXPRESS compiler.

References

[1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. *The MIMOLA Language Version 4.1.* University of Dortmund, 1994.

[2] D. Berson, R. Gupta, and M. L. Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. In *Working Conf. on Par. Arch. and Comp. Techniques*, 1994.

[3] M. Freericks. The nML machine description formalism. Technical Report 1991/15, Fachbereich Informatik, TU Berlin, 1991.

[4] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proc. of 34th Design Automation Conf.*, pages 299–302, 1997.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of Design Automation and Test in Europe*, 1999.

[6] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic software toolkit generation for embedded systems-on-chip. In *Proc. of Intn'l Conf. on VLSI and CAD*, 1999.

[7] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proc. of 35th Design Automation Conf.*, pages 510–515, 1998.

[8] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis for system-on-chip exploration. In *Proc. EUROMICRO-99*, 1999.

[9] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. *CHESS: Retargetable Code Generation for Embedded DSP Processors,* In *Code Generation for Embedded Processors* (P. Marwedel and G. Goossens, ed.), pages 85–102. Kluwer Academic Publishers, 1995.

[10] C. Liem, P. Paulin, M. Cornero, and A. Jerraya. Industrial experience using rule-driven retargetable code generation for multimedia applications. In *Proc. of 8th Int'l Symp. System Synthesis*, pages 60–65, 1995.

[11] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[12] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *Proc. of Intn'l Conf. on Parallel Processsing*, 1993.

[13] A. Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on parallelism. *Lang. and Compilers for Par. Comp.*, 1991.

[14] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, 1994.

[15] S. Novack and A. Nicolau. Resource directed loop pipelining : Exposing just enough parallelism. *The Computer Journal*, 1997.

[16] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.

[17] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An integrated approach to retargetable code generation. In *Proc. of 7th Int. Symp. on High-Level Synthesis*, 1994.