CUSTOMIZING SOFTWARE TOOLKITS FOR EMBEDDED SYSTEMS-ON-CHIP

Ashok Halambi, Nikil Dutt, Alex Nicolau

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

Modern Embedded Systems-on-Chips (SOCs) will allow the system designer to customize Intellectual Property (IP) cores (fixed and programmable), together with custom logic and large amounts of embedded memories. As the software content in these emerging embedded SOC begins to dominate the SOC design process, there is a critical need for support of an integrated software development environment (including compilers, simulators and debuggers). Furthermore, since many characteristics of these processor core IPs (e.g., instruction-sets, memory configurations) are increasingly customizable, the entire software toolkit chain needs to be customized and generated to support both early design space exploration (for performance, power and cost constraints), as well as high-quality software generation. This paper describes our Architecture Description Language (ADL) driven approach for customizing software toolkits.

1. Introduction

The advent of System-on-Chip (SOC) technology has resulted in a paradigm shift for the design process of embedded systems employing programmable processors with custom hardware. The design processes most impacted by this emerging technology include 1) System Design Space Exploration (DSE) and 2) Software design. Traditionally, embedded systems developers performed limited exploration of the design space using standard processor and memory architectures. Further, software development was usually done using existing, stock processors (with supported integrated software development environments) or done manually using processor specific low-level languages (assembly). This was feasible because the software content in such systems was low and also because the processor architecture was fairly simple (e.g. no Instruction Level Parallelism features) and well-defined (e.g. no parameterizable components).

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35409-5_23](https://doi.org/10.1007/978-0-387-35409-5_23)

The dotted box in Figure 1. shows a contemporary hardware/software co-design methodology for the design of traditional embedded systems consisting of programmable processors, application specific integrated circuits (ASICs), memories, I/O interfaces, etc. This contemporary design flow starts from specifying an application in a system design language. The application is then partitioned into tasks that are either assigned to software (i.e., executed on the processor) or hardware (ASIC) such that design constraints (e.g., performance, power consumption, cost, etc.) are satisfied. After hardware/software partitioning, tasks assigned to software are translated into programs (either in high-level languages such as C/C++ or in assembly), and then compiled into object code (which resides in memory). Tasks assigned to hardware are translated into HDL descriptions and then synthesized into ASICs. In traditional co-design systems, the target architecture template is pre-defined. Specifically, the processor is fixed or can be selected from a library of pre-designed processors, but customization of the processor architecture is not allowed. Even in co-design systems allowing customization of the processor, the fundamental architecture can rarely be changed.

Figure 1. ADL based co-design flow for SOC

In the SOC domain, system-level design libraries increasingly consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and super-scalar RISC, to VLIWs and hybrid ASIPs. These processor cores typically allow customization through

parameterization of features (such as number of functional units, operation latencies etc.). Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers, and partitioned register files). Together, these features allow exploration of a wide range of processor-memory organizations in order to customize the design for a specific embedded application.

The contemporary co-design flow (which does not permit processor-memory customization) limits the ability of the system designer to fully utilize emerging IP libraries and restricts the exploration of alternative (often superior) SOC architectures. Consequently there is tremendous interest in a language-based design methodology for embedded SOC optimization and exploration; Architectural Description Languages (ADLs) are used to drive DSE and automatic compiler/simulator toolkit generation.

As with an HDL-based ASIC design flow, several benefits accrue from a language-based design methodology for embedded SOC design, including the ability to perform (formal) verification and consistency checking, to modify easily the target architecture and memory organization for DSE, and to automatically generate the software toolkit from a single specification. Figure 1. illustrates the ADL-based SOC co-design flow, wherein the architecture template of the SOC (possibly using IP blocks) is specified in an ADL. This template is then verified or validated using formal methods. After verification, the software toolkit is automatically generated to be used for software compilation and co-simulation of the hardware and software.

Another important and noticeable trend in the embedded SOC domain is the increasing migration of system functionality from hardware to software, resulting in a high degree of software content for newer SOC designs. This trend, combined with shrinking time-to-market cycles, has resulted in intense pressure to migrate the software development to a high-level language (such as C, C++, Java) based environment in order to reduce time spent in system design.

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 [14] 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 ADL) as input to the compiler along with the program corresponding to the application.

The compilation process can be broadly broken into two steps: analysis and synthesis[1]. During analysis, the program (in HLL) is converted into an intermediate representation (IR) that contains all the desired information such as control and data dependences. During synthesis, the IR is transformed and optimized in order to generate efficient target specific code. The synthesis step is more complex and typically includes the following phases: Instruction Selection, Scheduling, Resource Allocation, Code Optimizations/Transformations, and Code Generation [15]. The effectiveness of each phase depends on the target architecture and the application. A further problem during the synthesis step is that the optimal ordering

between these phases (and other optimizations) is highly dependent on the target architecture and the application program. (For example, the ordering between the memory assignment optimizations and instruction scheduling passes is very critical for memory-intensive applications.) As a result, traditionally, compilers have been painstakingly hand-tuned to a particular architecture (or architecture class) and application domain(s). However, stringent time-to-market constraints for SOC designs no longer make it feasible to manually generate compilers tuned to particular architectures.

The ADL-based retargetable compiler approach allows fast generation of a target specific compiler from a high-level description of the processor architecture. However, in order to be able to generate high-quality code, there is a need for techniques that allow customization of the compiler based on the application (specified in a HLL) and the architecture (specified in an ADL). 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. The EXPRESSION ADL is used to specify the processor-memory subsystem and achieve retargetability of EXPRESS. In Section 2 we present related work in the areas of compiler retargetability and customization. In Section 3 we present our EXPRESS compiler framework, briefly describing EXPRESSION, EXPRESS and Transmutations. In Section 4 we present some preliminary results motivating the need for compiler customization. Finally, we conclude with a summary in Section 5.

2. Related Work

The problem of generating efficient software toolkits for embedded systems has been the focus of several recent research activities. [9] 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 [13] project demonstrated a compiler for a limited embedded processor class with irregular architectures. CHESS [12] is a retargetable code generation environment for fixed-point DSP processors. CHESS uses the nML [4] ADL to achieve retargetability. The AVIV [10] compiler, using the ISDL [7] ADL, produces machine code optimized for size. The MSSQ and RECORD compilers use the MIMOLA [2] 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 [22], 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 [17], [3]. 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 CHES compiler uses data routing as a technique to simultaneously solve the problems of code selection and register allocation. Mutation Scheduling (MS) [18] 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 3.3, 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. The detailed architectural information need by EXPRESS to efficiently retarget itself is derived from a high-level description of the processor-memory subsystem in EXPRESSION. In the following, we first present a brief overview of

the EXPRESSION ADL, the EXPRESS compiler and then describe the Transmutations framework in detail.

3.1 EXPRESSION ADL

We use EXPRESSION, an ADL designed to support design space exploration (DSE) of a wide class of processor architectures ranging from RISC, DSP, ASIP, and VLIW, coupled with a variety of memory system organization and hierarchies [8], in order to retarget EXPRESS, an optimizing memory-aware, ILP compiler. EXPRESSION contains an integrated specification of both structure and behavior of the processor-memory system. The structure is a net-list of components (i.e., units, storages, ports, and connections), as well as the valid unit-to-storage or storage-to-unit data transfers. The pipeline architecture is described as the ordering of units which comprise the pipeline stages, plus the timing of the multi-cycled units. The behavior describes the IS in a hierarchical manner. Each operation is defined in terms of its opcode, operands, and format. Each instruction is viewed as a list of slots to be filled with operations. In EXPRESSION, resource conflicts between instructions are not explicitly described, but reservation tables (RTs) specifying the conflicts are automatically generated and passed to the ILP compiler [6]. Since manual description of RTs for deeply pipelined or VLIW processors is cumbersome and error-prone, this ability to automatically generate RTs facilitates rapid DSE of SOC architectures. Another key feature of EXPRESSION is the ability to specify novel memory systems including memory hierarchies, on-chip DRAM, frame buffers, partitioned memory address spaces, etc. EXPRESSION is also able to automatically generate the timing behavior of each operation based on the timing behavior of architectural components [5]. The architecture specific resource constraint and timing information is then used by the EXPRESS compiler to retarget its optimizations and produce code tailored for that particular architecture.

3.2 EXPRESS

EXPRESS is an optimizing, memory-aware, Instruction Level Parallelizing (ILP) compiler. EXPRESS uses the EXPRESSION ADL [8] to retarget itself to a wide class of processor architectures and memory systems. Figure 2. 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 [19] - a loop pipelining technique, TiPS : Trailblazing Percolation Scheduling [16] - 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 [11], a cycle accurate, structural simulator to analyze the performance of generated code.

Figure 2. EXPRESS Framework

3.3 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 that 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 [18] 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 2. 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 that 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 [21] suite, and also from scientific computation benchmarks.

Table 1: Phase Ordering for Partial Predication

Bench	Partial Pred.					
	Pred.		Spec.		Pred. & Spec.	
	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 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.

Table 2: Phase Ordering for Full Predication

Bench	Partial Pred.					
	Pred.		Spec.		Pred. & Spec.	
	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 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] A. V. Aho, R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*, Addition-Wesley.
- [2] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer, (1994). *The MIMOLA Language Version 4.1*. University of Dortmund.
- [3] D. Berson, R. Gupta, and M. L. Soffa (1994). Resource spackling: A framework for integrating register allocation in local and global schedulers. In *Working Conf. on Par. Arch. and Comp. Techniques*.
- [4] M. Freericks, (1991). The nML machine description formalism. Technical Report 1991/15, Fachbereich Informatik, TU Berlin, 1991.
- [5] P. Grun, N. Dutt, and A. Nicolau, (2000). Memory aware compilation through accurate timing extraction. In *Proc. of Design Automation Conference*.
- [6] P. Grun, A. Halambi, N. Dutt, and A. Nicolau, (1999). RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. To appear at *12th Int'l Symp. System Synthesis*, 1999.
- [7] G. Hadjiyiannis, S. Hanono, and S. Devadas, (1997). ISDL: An instruction set description language for retargetability. In *Proc. of 34th Design Automation Conf.*, 299-302.
- [8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau (1999). EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of Design Automation and Test in Europe*.
- [9] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau, (1999). Automatic software toolkit generation for embedded systems-on-chip. In *Proc. of Intn'l Conf. on VLSI and CAD*.
- [10] S. Hanono and S. Devadas, (1998). Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proc. of 35th DAC*, 510-515.
- [11] A. Khare, N. Savoie, A. Halambi, P. Grun, N. Dutt, and A. Nicolau (1999). V-SAT: A visual specification and analysis for system-on-chip exploration. In *Proc. EUROMICRO-99*.

- [12] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, (1995). *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.
- [13] C. Liem, P. Paulin, M. Cornero, and A. Jerraya (1995). Industrial experience using rule-driven retargetable code generation for multimedia applications. In *Proc. of 8th Int'l Symp. System Synthesis*, 60-65.
- [14] P. Marwedel and G. Goossens, editors (1995). *Code Generation for Embedded Processors*. Kluwer Academic Publishers.
- [15] S. S. Muchnick (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- [16] A. Nicolau and S. Novack (1993). Trailblazing: A hierarchical approach to percolation scheduling. In *Proc. of Int'n'l Conf. on Parallel Processsing*.
- [17] A. Nicolau, R. Potasman, and H. Wang (1991). *Register allocation, renaming and their impact on parallelism*. Lang. and Compilers for Par. Comp.
- [18] S. Novack and A. Nicolau (1994). Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, 1994.
- [19] S. Novack and A. Nicolau (1997). Resource directed loop pipelining : Exposing just enough parallelism. *The Computer Journal*.
- [20] P. R. Panda, N. Dutt, and A. Nicolau (1999). *Memory Issues in Embedded Systems-on-Chip*. Kluwer Academic Publishers.
- [21] Trimaran Release: <http://www.trimaran.org>. *The MDES User Manual*, 1997.
- [22] T. Wilson, G. Grewal, B. Halley, and D. Banerji (1994). An integrated approach to retargetable code generation. In *Proc. of 7th Int. Symp. on High-Level Synthesis*.

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Inc.