Copyright © 1999, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

DATA FLOW AND CONTROL OPTIMIZATIONS FOR HARDWARE AND SOFTWARE CO-SYNTHESIS IN EMBEDDED SYSTEMS

by

Bassam Tabbara and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M99/31

15 June 1999

DATA FLOW AND CONTROL OPTIMIZATIONS FOR HARDWARE AND SOFTWARE CO-SYNTHESIS IN EMBEDDED SYSTEMS

by

Bassam Tabbara and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M99/31

15 June 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Data Flow and Control Optimizations for Hardware and Software Co-synthesis in Embedded Systems

Bassam Tabbara *
Alberto Sangiovanni-Vincentelli
EECS Department, U.C. Berkeley, Berkeley, CA 94720
{tbassam, alberto}@eecs.berkeley.edu

Research Report

Abstract

Current co-design methodologies of control dominated hardware software systems suffer from inefficient hardware (HW) and software (SW) synthesis of the various reactive system tasks. In order to improve synthesis quality, we propose a methodology that incorporates data flow in addition to control optimizations performed on a suitable task representation in a hardware and software co-design environment. We introduce our approach here, and report initial results of our investigation which show that performing such optimizations can lead to size and performance improvements in both the synthesized hardware and software.

1 Introduction and Overview

Embedded systems are very prevalent in today's society and promise to be even more common and found in many of the things we interact with on a daily basis. Applications vary from today's airplane or car controllers, and cellular phones and pagers to the future's autonomous kitchen appliances, and transportation vehicles.

These various applications not only require that the implementation be reliable and cost-effective, in addition they impose constraints on the hardware and the software components of the system. Invariably, the system must be *efficient* i.e. speed of execution of the software, and performance of the hardware must be adequate. The system must also be *small* in size if it is to fit seamlessly in common objects therefore both code size of the software and silicon area of the hardware must be within bounds.

1.1 Reactive System Co-synthesis

While others have developed computational models especially suited for data processing applications

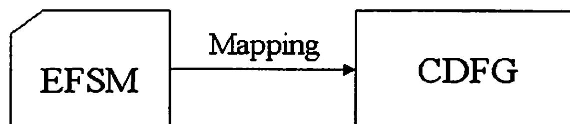

Figure 1: Reactive System Co-synthesis

(such as SDF [13] or DDF [6]), or proposed a unified model for control and data flow modeling (such as [8]), we are strong proponents of the separation of function and communication [15], in heterogeneous control-dominated embedded systems since the separation of concerns is ideal for architectural trade-offs, in addition it makes component re-use quite straightforward. This is why, in this work, we assume a model of computation and a *functional decomposition* that represents the design as a network of EFSMs as in [4], and [17].

Current software and hardware co-synthesis strategies for *control-dominated* applications are aimed at efficient (fast and compact) implementation of a reactive decision process [2]. Data flow aspects are neglected; it is generally assumed that software compilers and hardware Register Transfer Level (RTL) compilers will address these optimizations.

The typical synthesis process, as shown in Figure 1, starts with design capture using finite state machines extended with operations and data computations referred to here as Extended Finite State Machines (EFSMs). The EFSM of each system module is then mapped in this flow onto a Control Data Flow directed acyclic Graph (CDFG) which is then used to generate reactive hardware or software. A transition of the EFSM is performed by executing a path in the CDFG when the task is invoked.

While the CDFG is ideal for representing the reac-

*SRC Graduate Fellow under contract DC-324-028

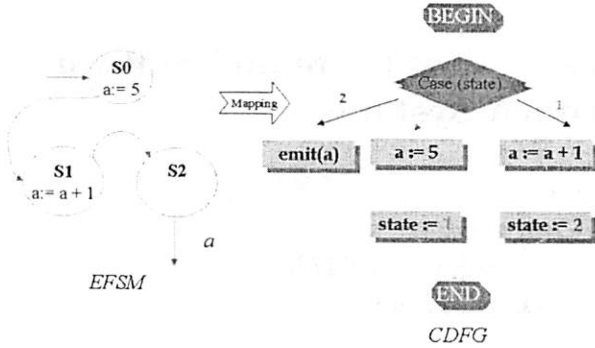

Figure 2: Data Flow Optimization and the CDFG Representation

tive tasks to be synthesized since it can be used for both early size/speed estimation as well as synthesis of the hardware and code generation of the software, this representation hides much of the control flow across invocations of the reactive module, and consequently data cannot be fully propagated. This limits data flow optimizations, as well as control optimizations that depend on this data, to just optimizing paths in the CDFG DAG without considering the optimizations across paths. We illustrate this using a simple example shown in Figure 2.

The example shows an EFSM with a constant propagation opportunity that would save a needless addition operation. The `a = 5` operation of S1 and the `a = a + 1` of S2 can be combined into one `a = 6` operation in S2. This optimization cannot be easily identified in the CDFG representation since it is distributed across two invocations of the reactive task (first for state `S1` and second for state `S2`).

1.2 Our Contribution

We introduce here a design representation for each system task that is able to capture the EFSM description, and is at the same time suitable for performing data flow and control optimizations. We show that performing data flow and control optimizations at the design representation level will directly reflect positively on the size and performance of *both* the synthesized *hardware* and *software*. We are currently evaluating this optimization for synthesis approach by incorporating data flow and control optimizations into the co-synthesis flow of a typical co-design environment that targets reactive controllers ([4])). In the sequel we describe the key ideas behind our approach.

2 Data Flow and Control Optimization Approach

Our proposed optimization approach is divided into 2 phases:

1. **Architecture Independent**: EFSMs are considered individually; data flow analysis and *intra*-EFSM optimizations are performed. The optimizations here are useful for both size and performance improvement since they involve removing redundant and useless tests and assignments, and in general decreasing the number of variables in the design.

2. **Architecture Dependent**: Optimizations in this stage rely on architectural information to perform additional optimizations tuned to the design target and typically involve some trade-off between size and performance.

   - Scheduling and Resource Allocation for Hardware: The interconnection of EFSMs is considered; issues such as resource sharing, communication overhead, scheduling and pipelining are addressed for hardware optimization.

   - Instruction Selection for Software: Instruction selection, allocation, and scheduling is performed for software using approaches similar to those being developed by the SPAM project [9].

2.1 Architecture Independent Data Flow and Control Optimizations

2.1.1 Previous Work

Previous work in control is mostly based on BDD-based optimization [3] techniques for Control Flow Graphs (CFGs) such as [5]. The limitation of these optimizations is that they neglect the data and are in fact "data value blind"; control optimizations that can result from data analysis (such as dead code elimination, and copy propagation for example) are not available to such techniques.

The two most relevant bodies of work to our research are:

- High Level Synthesis for *Silicon Compilation*

- Code Optimization Techniques for *Software Compilation*

High level synthesis for *silicon compilation* has been an active research area in the past 2 decades. The focus of such techniques however has been mostly

on approaches for scheduling, allocation, and binding of the specification (usually a Hardware Description Language (HDL)) to the hardware implementation. General optimization techniques such as common sub-expression extraction, and constant folding, are applied in a local fashion [14].

The literature is rich in data flow optimization techniques, most notably classical optimization techniques of [11], [10], and recent work by [7], [1], and [16]. Most of that work, however, has focused on ***hand-written code*** optimization. In fact the architecture independent and dependent parts are most often mixed together in a general optimizing compiler intended usually for code optimization of a specific component processor and instruction set.

2.1.2 Intermediate Design Representation: CLIF

We have developed an intermediate design representation called C-Like Intermediate Format (CLIF) for each module in the system. This representation is able to capture the EFSM semantics and behavior, and is suitable for data flow analysis.

CLIF textual intermediate format consists of a sequence of **TEST** and **ASSIGN** instructions as follows:

- **TEST** instruction
  if (condition) *goto* **label**
- **ASSIGN** instruction
  dest = op(*src1*)
  dest = *src1* op *src2*

The format has no aliasing i.e. no side effects; operations involve **ASSIGN**ing to the target a result of a computation performed by using *instructions* on one or two source operands (typically referred to in the software compilation domain as *quadruples*), or **TEST**ing a variable and performing a resulting action. The control statement is the infamous `goto` statement. The format has C syntax, and supports all the unary and binary arithmetic, boolean, and relational operators in C.

The true representation is of course the CLIF flow graph where each EFSM state is represented as a node (label in the textual representation). Edges represent control flow labeled with conditions and outputs. A simple CLIF flow graph, along with its textual representation is shown in Figure 3. The Figure shows many opportunities for data flow and control optimizations such as eliminating the `a = b + c` operation since it is useless (`a` is defined again before the

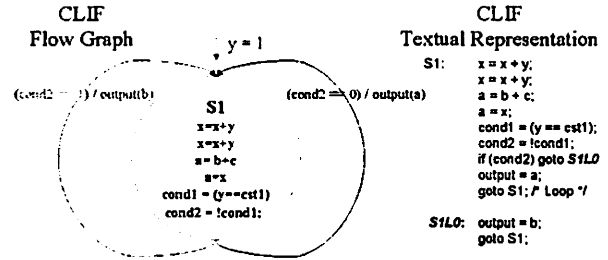

Figure 3: CLIF Flow Graph and Textual Representation

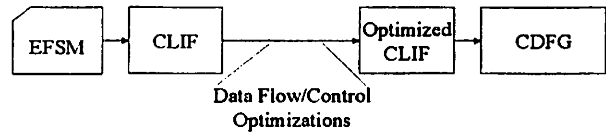

Figure 4: Proposed Optimization and Synthesis Flow

result of the said operation is used), and performing dead code elimination on the `(cond2 == 1)` branch since `y` is always equal to `1` upon entry to state `S1`, consequently `cond2` is `0`.

2.1.3 Our Optimization Flow

We propose to add data flow and control optimization at the design representation level. The purpose of the approach would be two-fold: a) raise the abstraction level, and allow optimization to be reflected in both hardware and software synthesis, b) incorporate powerful classical data flow and control optimizations that have a tremendous potential for improving the quality of the synthesized output. Our modified co-synthesis flow is shown in Figure 4. After the CDFG is generated we proceed with reactive synthesis.

2.1.4 Data Flow Analysis and Optimization

In order to implement *intra*-EFSM optimizations we need to be able to identify instructions and variables that can be eliminated in CLIF. To that end we are developing an *optimizing compiler* that examines blocks in the design flow graph in order to statically collect the data flow and control information of the task. This is referred to as data-flow analysis. Data flow information can be collected by setting up and solving a system of equations that relate data at various points in the module behavioral description. The general equation is of the form:

$$Node.pass = Node.gen + (Node.reach - Node.kill)$$

This equation means that the information passed along to other nodes in the flow graph is the information generated in this block or the information reaching this block but not killed in the block. Such an equation is called the data flow equation. The notions of "generating" and "killing" depend on the desired information. Moreover, for some problems, instead of proceeding along the flow of control, we may need to proceed in the opposite direction.

The optimizer solves a set of such **data flow equations** using the *iterative method* which has been shown to be a general (i.e. applies to arbitrary flow graphs) and optimal [11] method for the data gathering problems we solve [12]. Our goal is to optimize for speed and size.

The types of problems we solve to gather information about the design include:

- Control flow analysis
- Variable reaching definitions and uses
- Available expression computation

The code improvement techniques and transformations we perform include:

- Unreachable block elimination
- Normalization
- Copy propagation
- Constant propagation and folding
- Common sub-expression simplification and extraction
- Code motion

Control optimizations through data flow analysis include:

- Dead code/store elimination
- BDD size reduction by shrinking support due to the reduction in variable tests.

3 Preliminary Results

Initial experimental results are very encouraging and have shown a measurable improvement in the quality of the synthesized hardware and software on the order of 10-20% on control-dominated examples with greater improvement results the more abundant data computations are. Design representation level optimizations seem to assist the lower (abstraction) level optimization algorithms and heuristics as well. This is because typically the lower level algorithms deal with smaller granularity optimizations and therefore perform better on smaller inputs.

4 Future Research Opportunities

Aside from completing and expanding on the architecture independent optimizations such as addressing data flow as well as control specifications (e.g. C functions for data, SDL for control), We plan to explore opportunities in *Function Architecture Co-design*. An attributed version of the CLIF format serves as an excellent mechanism for function/architecture co-design. By analyzing the EFSM functions in the network, an automated design assistant can recommend suitable architectures for implementation (e.g. instruction selection). On the other hand, architectural constraints specified by the user can feed further optimization opportunities back to the function, and the assistant can then attempt to massage the function to meet these constraints (e.g. operator strength reduction).

Acknowledgment

Thanks goes to Abdallah Tabbara from UC Berkeley, and to Luciano Lavagno and Felice Balarin from Cadence Design Systems for their continued assistance and support. The authors also wish to thank the SRC who is funding this research under the Graduate Fellowship Program.

References

[1] Aho, A. V.; Sethi, R.; Ullman, J.D., "Compilers: Principles, Techniques, and Tools" *Addison-Wesley*, 1988.

[2] Balarin F.; Chiodo M.; Giusto P.; Hsieh H.; Jurecska A.; Lavagno L.; Passerone C.; Sangiovanni-Vincentelli A. L.; Sentovich E.; Suzuki K.; and Tabbara B., "Hardware-Software Co-Design of Embedded Systems: The POLIS Approach", *Kluwer Academic Publishers*, MA, USA, May 1997.

[3] Bryant, R. "Graph-Based Algorithms for Boolean Function Manipulation" *IEEE Transactions on Computers*, 1986.

[4] Chiodo M., Giusto P., Hsieh H., Jurecska A., Lavagno L., and Sangiovanni-Vincentelli, A., "Hardware/software Co-design of Embedded Systems" *IEEE Micro*, Vol. 14, Number 4, pp. 26-36, 1994.

[5] Chiodo, M.; Giusto, P.; Jurecska, A.; Lavagno, L.; Hsieh, H.; Suzuki, K.; Sangiovanni-Vincentelli, A.; Sentovich E., "Synthesis of Software Programs for Embedded Control Applications", *DAC*, June 1995.

[6] Choi, C.; Ha S. "Software Synthesis for Dynamic Data Flow Graph", *IEEE International Workshop on Rapid System Prototyping*, June 1997.

[7] Chow, F.C. "A Portable Machine-Independent Global Optimizer–Design and Measurement", *Ph.D. Thesis*, Stanford University, 1983.

[8] Grotker, T.; Schoenen, R.; Meyr, J., "Unified Specification of Control and Data Flow", *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1997.

[9] Hanono, S.; Devadas, S. "Instruction Selection, Resource Allocation, and Scheduling in the Avis Retargetable Code Generator".

[10] Kam, J.B., Ullman, J.D. "Global Data Flow Analysis and Iterative Algorithms" *J. ACM*, 1982, pp. 111-126.

[11] Kildall, G. "A Unified Approach to Global Program Optimization", *ACM Symposium on Principle of Programming Languages*, 1973, pp. 194-206.

[12] Lemone, K. A. "Design of Compilers: Techniques of Programming Language Translation", *CRC Press*, 1992.

[13] Murthy, P.K.; Bhattacharya, S.S.; Lee, E.A., "Joint Minimization of Code and Data for Synchronous Dataflow Programs", *Journal of Formal Methods in System Design*, July 1997.

[14] Goossens, G.; Lanneer, D.; Vahoof, J.; Rabaey, J.; Van Meerbergen, L.; De Man, H. "Optimization-based Synthesis of Multiprocessor Chips for Digital Signal Processing, with CATHEDRAL II", *International Workshop on Logic and Architecture Synthesis for Silicon Compilers*, 1988.

[15] Rowson, J.; Sangiovanni-Vincentelli, A., "Interface-Based Design", *DAC*, 1997.

[16] Tjiang, S.W.. "Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers", *Ph.D. Thesis*, Stanford University, 1993.

[17] Vahid, F., Gajski, D. "Incremental Hardware Estimation During Hardware/Software Functional Partitioning", *IEEE Transactions on VLSI Systems*, Sept. 1995.