Hardware and Software Representation, Optimization, and Co-synthesis for Embedded Systems

Bassam Tabbara
EECS Department
University of California at Berkeley
Berkeley, CA 94720
+1-510-643-5187
tbassam@eecs.berkeley.edu

Abdallah Tabbara
EECS Department
University of California at Berkeley
Berkeley, CA 94720
+1-510-643-5187
atabbara@eecs.berkeley.edu

Alberto Sangiovanni-Vincentelli
EECS Department
University of California at Berkeley
Berkeley, CA 94720
+1-510-642-4882
alberto@eecs.berkeley.edu

Abstract

Current software and hardware co-synthesis methodologies of control dominated embedded systems focus primarily on improving productivity in the complex design process. In order to improve *synthesis quality*, we propose a methodology that incorporates data flow and control optimizations performed on a novel implementation independent design task representation. The approach is applicable to any co-synthesis tool; we use a public domain co-design environment to report some results of our investigation. The data collected shows that performing such optimizations on an adequate representation can lead to considerable size and performance improvements in both the synthesized software and hardware.

Keywords

Design representation, control and data flow analysis, optimization, software and hardware co-design, co-synthesis.

1 Introduction

Embedded systems are very prevalent in today's society and promise to be even more commonly found in many of the things we interact with on a daily basis. Applications vary from today's airplane or car controllers, and the abundant communication devices to the future's autonomous transportation vehicles and consumer electronics and appliances.

Hardware/Software Co-design (HSC) is a recent field that emerged out of the pressures of the always shrinking time-to-market and the ever-increasing demand for more "intelligence" on board the embedded device. Design productivity, however, should not be improved at the expense of quality of synthesis; the various applications not only require that the product design cycle be fast and correct-in-the-first-time, the final implementation must be reliable, cost-effective, and of good merit. These requirements impose constraints on the hardware (HW) and software (SW) 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 product 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. Other design metrics include power, as well as application specific considerations such as fuel preservation and emission control in an embedded vehicle controller.

1.1 Assumptions

While fellow researchers have developed representations especially suited for data processing applications (such as SDF [16] or DDF [9]), in this work we target heterogeneous control-dominated embedded system applications, so we assume a *functional decomposition* that captures the design as a network of Finite State Machine modules extended with data computation (EFSMs) as described in [7], and [21]. Each module behavior is conveyed using graphical entry or an FSM-based reactive language (for example Esterel [6]) front-end. We focus on the representation, optimization, and synthesis of each individual task in the network, so we do not assume a specific model of computation governing the composition of these tasks in the system.

1.2 Reactive System Co-synthesis

Current software and hardware co-synthesis strategies for *control-dominated* applications are aimed at the efficient (fast and compact) implementation of a reactive decision process [3]. Data flow aspects are neglected; it is generally assumed that software compilers and hardware Register Transfer Level (RTL) compilers will address these optimizations.

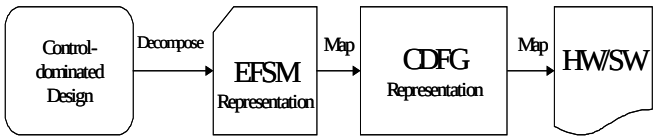

Figure 1. Reactive System Co-synthesis

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

While the CDFG is ideal for representing the reactive 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 without considering the optimizations across paths. We illustrate this using a simple example shown in Figure 2.

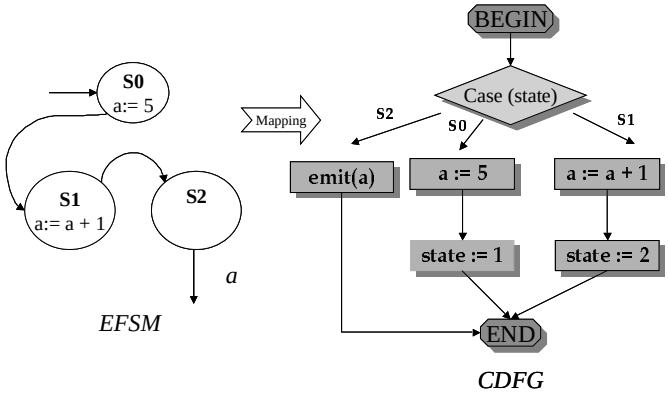

Figure 2. Data Flow Optimization and the CDFG Representation

The example shows an EFSM with a constant propagation opportunity that would save a runtime addition operation. The `a = 5` operation of `S0` and the `a = a + 1` of `S1` can be combined into one `a = 6` operation in `S1`. This optimization cannot be easily identified in the CDFG representation since it is distributed across two invocations of the reactive task (first for state `S0` and second for state `S1`).

The *key point* to emphasize here is that it cannot be expected of software and hardware compilers (as ‘‘conventional wisdom” leads one to believe) to *statically* discover such an optimization that involves a *run-time* decision in the task CDFG representation. The very simple example of Figure 2 shows that the CDFG representation has a shortcoming in identifying potential optimizations that lie across task invocations, and our recourse, as will be described shortly, is to use an equivalent task representation that is better able to identify and exploit such optimizations.

1.3 Our Contribution

We introduce here a design representation for each system task that is able to capture the EFSM description, and at the same time is 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 *software* and *hardware*. We are currently evaluating this optimization for synthesis approach by incorporating data flow and control optimizations into the co-synthesis flow of a representative public domain co-design environment that targets reactive controllers [7]. 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* Phase: 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 EFSM task.
2. *Architecture Dependent* Phase: Optimizations in this stage rely on architectural information to perform additional optimizations tuned to the design target and typically involve some estimation-based trade-off between size and performance. These include:
   - Scheduling and Resource Allocation of the *macro*-architecture: The interconnection of EFSMs is considered; issues such as resource sharing, communication overhead, scheduling and pipelining are addressed for optimization.
   - Instruction Selection for Software: Instruction selection, allocation, and scheduling are performed for the *micro*-architecture [13].

The process can of course be iterated for different functional decompositions.

3 Architecture Independent Data Flow and Control Optimizations

3.1 Related Work

Previous work in control optimization is mostly based on BDD-based techniques for Control Flow Graphs (CFGs) such as [8]. 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 operation 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 (HLS) for *silicon compilation*, and
- Code optimization techniques for *software compilation*.

High-level synthesis for silicon compilation has been an active research area in the past two 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 such as [17]. General optimization techniques, for example common sub-expression extraction, and constant folding, are applied in a local fashion. Bergamaschi recently proposed a design representation, Behavioral Network Graph (BNG), for unifying the domains of HLS and Logic Synthesis [5]. His work recognized the need for an internal design representation on which to perform data path and control optimization before logic synthesis of hardware. The BNG, however, is at an even lower abstraction level than the design CDF DAG and is therefore not suitable for our need of a unifying design representation for software and hardware at the high abstraction level in the embedded co-design domain.

The literature has a wealth of data flow optimization techniques, most notably classical optimization techniques of Kildall [15], Kam et.al. [14], and more recent work by Aho et.al. [1], and Tjiang [20]. However, the focus has been on *hand-written code* optimization. In fact the architecture independent and dependent parts are most often tightly coupled together in a general optimizing compiler intended usually for code

optimization of a specific component processor and instruction set.

In our work we specialize the aforementioned silicon and software compilation techniques to the embedded systems domain since these techniques can be applied on any internal design representation, no matter what the abstraction level, and need not be restricted to the final stages of software assembly code generation, or hardware synthesis. Early optimization, as we will show, benefits all the subsequent steps in the control-dominated co-design flow.

3.2 Intermediate Design Representation

We have developed a novel implementation-independent (initially unscheduled) task representation referred to as *Function Flow Graph (FFG)* equivalent to the EFSM representation, and quite similar to the classical CFG from the software domain. The representation is able to capture the EFSM semantics and behavior, and is well suited for control and data flow analysis.

3.2.1 Function Flow Graph (FFG)

FFG is the task representation used for design analysis and optimization. Each EFSM state is represented as a collection of nodes, and edges represent control flow. This flow graph is the data structure on which the task control flow analysis is performed, and data flow information is gathered.

Definition 1: A Function Flow Graph (FFG) is a triple $G = (V, E, N_0)$ where

i. V is a finite set of nodes
ii. $E = \{(x,y)\}$ is a subset of VxV, (x,y) is an edge from x to y where $x \in Pred(y)$, the set of predecessor nodes of y.
iii. $N_0 \in V$ is the start node corresponding to the EFSM initial state.
iv. A set of operations is *associated* with each node N.
v. Operations consist of **TEST**s performed on the EFSM inputs and internal variables, and **ASSIGN**s on the EFSM outputs and internal variables.

3.2.2 C-Like Interchange Format (CLIF)

The textual interchange format of the FFG is called the C-Like Intermediate Format (CLIF). The format consists of an unordered set of **TEST** and **ASSIGN** operations:

- **TEST** instruction
  [*if* (condition)] *goto* **label**
- **ASSIGN** instruction
  dest = `unop`(*src1*)
  dest = *src1* `binop` *src2*
  dest = `func`(*arg1*, *arg2*, ...)

The format does not have any aliasing that is there are no side effects (no pointers); operations involve **ASSIGN**ing to the target a result of a computation performed on one or two source operands (typically referred to in the software compilation domain as *quadruples*) or an assignment from a *stateless* arithmetic or Boolean function, or **TEST**ing a variable and performing a resulting action. The control transfer statement is the *goto* statement. The format has C syntax, and supports all the unary and binary arithmetic, Boolean, and relational operators in C.

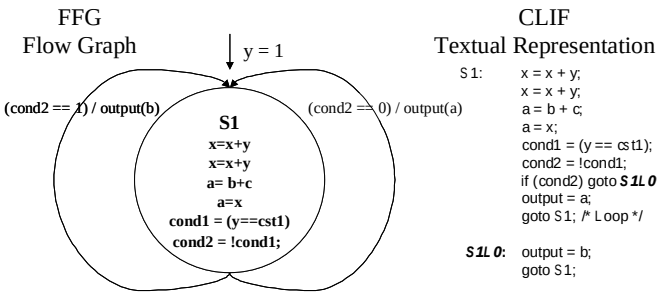

Figure 3. Simple FFG and Its CLIF Representation

A simple FFG graph, along with its equivalent CLIF textual representation is shown in Figure 3. The Figure exhibits typical opportunities for data flow and control optimizations such as eliminating the `a = b + c` operation since it is useless (`a` is re-defined before the result of the said operation is used), and performing dead operation elimination on the `(cond2 == 1)` branch since `y` is always equal to `1` upon entry to state `S1`, consequently `cond2` is `0`. While the Figure shows ''local" optimizations for simplicity, the goal is to address global versions of these optimizations.

3.3 Our Optimization Flow

We perform data flow and control optimization at the design representation level. The purpose of the approach is two-fold:

a) Raise the abstraction level, and allow optimization to be reflected in both software and hardware synthesis, and
b) Incorporate powerful classical data flow and control optimizations that have a considerable potential for improving the quality of the synthesized output.

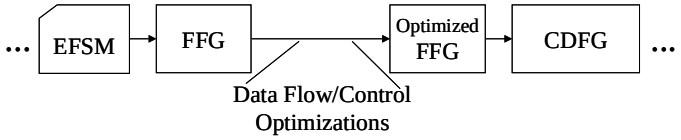

Figure 4. Our Optimization and Synthesis Flow

Our modified co-synthesis flow is shown in Figure 4. After the FFG is mapped onto an *optimized* CDFG we proceed with reactive synthesis.

3.3.1 Control and Data Flow Analysis

In order to implement task optimizations we need to identify operations and variables that can be eliminated using a static conservative procedure. To that end we have developed an *optimizer* that examines the FFG in order to statically collect data flow and control information of the task under analysis using an underlying data flow analysis framework.

Definition 2: A monotone data flow analysis framework is a triple $D = (L, \wedge, F)$ where:

i. L is a bounded semilattice with meet $\wedge$, and
ii. F is a monotone function space associated with L.

The framework can be used to manipulate the data flow information by interpreting the *node labels* on N in V of the control flow graph G as elements of an algebraic structure L.

Definition 3 (adapted from [14]): A particular (problem) instance of a monotone data flow analysis framework is a pair $I = (G, M)$ where $M: N \rightarrow F$ is a function that maps each node N in V

of G to a function in F on the node label semilattice L of the framework D.

While control flow can be inferred from the structure of the FFG flow graph, data flow information can be collected by setting up and solving a system of equations that relate data at various points in the FFG module behavioral description. In the context of the FFG and the problems we solve, M is an algorithm for extracting information from the quadruple sequence associated with each node N, and grouping it into a *set* which becomes the node label for the corresponding node in the data flow problem instance graph G; F is a pair of equations describing information flow while the meet operator for forward flow problems in this case (i.e. when we proceed along the flow of control in the FFG) is set intersection $\cap$. The general form of the system of equations to be solved is as follows:

$$\forall N \in V, \text{ where } G = (V, E, N_0) \text{ is the task FFG}$$

$$In(N) = \bigcap_{P \in \text{Pred}(N)} Out(P)$$

$$Out(N) = (In(N) - Kill(N)) \bigcup Gen(N)$$

The equation pair for every node N means that the information passed along to other nodes in the FFG (*Out(N)*) is the information reaching the node (the *meet along all paths* leading into node: *In(N)*) and not invalidated in the node (*Kill(N)*) added to the information generated in the node itself (*Gen(N)*) [14]. In some cases we may need to proceed in reverse to the flow of control (typically when "use" information is required as opposed to "definitions"); the equations differ slightly in that case. The types of problems we solve to gather information about the design are all monotone and include such classical problems as variable reaching definitions and uses, and available expression computation.

Our optimizer solves these *data flow equations* using the *iterative method*, which has been shown to be a general (i.e. applies to arbitrary flow graphs) and optimal method for the data gathering problems we solve [15]. While the average running time can be improved by *depth-first* ordering of the FFG, the method's worst-case time complexity is as follows:

$$O(n^2) \text{ where } n = \text{number of FFG nodes}$$

$$\text{and } n = \sum_{i=1}^{N} K_i$$

$$\text{where } N = \text{number of EFSM states}$$

$$K_i = 1 + C_i$$

$$C_i = \text{number of transitions in state } i$$

3.3.2 Control and Data Flow Optimizations

Once the analysis and information gathering about the task at hand is complete, optimization of the FFG begins. The objective is to optimize the FFG task representation for speed of execution and size (area of hardware, code size of software). The design representation improvement techniques and transformations through data flow analysis that are performed include:

- Live variable preservation, and dead variable removal through reaching definition and use analysis
- Identification of same computations through normalization,
- Copy propagation, and constant folding,
- Common sub-expression simplification, and
- Movement of operations between states in order to speed up the critical path guided by frequency of execution estimates.

Control optimizations include:

- Unreachable FFG node elimination,
- Dead operation elimination, and
- Reduction of test variables and normalization of **TEST** operations, which leads to tremendous size reductions in the CDFG, used for synthesis [3].

Our design representation optimization process *specializes* the classical techniques and transformations listed above to the control-dominated embedded domain where *reactive semantics* are imposed (see Section 5 for an example of one possible mapping policy) and input/output traces must be preserved.

4 Architecture Dependent Optimizations

Up till now, we have described optimizations that involve reducing useless and redundant operations, and variable tests and assignments. Such optimizations are reflected in both size (area), and speed (performance) improvements of the hardware or software. Scheduling, allocation, and binding decisions change this picture; these will usually involve a trade-off between size and speed depending on the target architecture. For example, if the architectural constraints (e.g. clock speed) are such that data path computation can be performed at every task invocation (e.g. clock cycle), then data path components can be "shared" across invocations (smallest code size/area extreme) since all their inputs and outputs would be registered for the given schedule. The data we present in Section 6 shows the other (fastest) extreme where *estimated speed* is optimized i.e. computations are all "replicated".

5 Synthesis

In order to perform synthesis, CLIF is *mapped* into the Software Hardware Intermediate FormaT (SHIFT) representation of the POLIS co-design toolset. SHIFT is a representation format for describing a network of EFSMs. It is a hierarchical netlist [3] of:

- Co-design Finite State Machines (CFSMs): finite state machines with reactive behavior
- Functions: state-less arithmetic, Boolean, or user-defined operations.

A CFSM execution consists of four phases:

1. Idle awaiting trigger inputs
2. Sample inputs when invoked
3. Compute chain of operations
4. Emit outputs, return to Idle mode

A CFSM in SHIFT is therefore composed of input, output, state or feedback signals with initial values, as well as a transition relation (TREL) that describes the reactive behavior. Functions are used in the TREL to **ASSIGN** computation results to valued outputs.

A function can be thought of as a combinational circuit in hardware or a function (with no side effects) in software.

We therefore decompose the CLIF representation of each task into a single reactive control part, and a set of data path functions consistent with the current default SHIFT macro-architecture. We then use the POLIS engine to build the CDFG for software and hardware co-synthesis.

6 Results

6.1 Experimental Procedure

Embedded system target architectures are quite numerous and varied, so for space considerations we restrict our presentation here to software synthesis on a single processor. To make our results accessible and useful to a wide audience, we choose to report size improvement results obtained from the CDFG node count directly, before the final mapping onto a software implementation. The CDFG is built after performing the FFG task representation architecture independent optimizations, as well as the architecture dependent arithmetic function replication described in Section 4. Collecting speed results involve profiling on the target architecture and so are not as simple to report. To that end, we will be reporting estimates collected on 3 representative platforms:

1. Introl compiler for the 68HC11: representing the 8-bit/16-bit data (4 MHz) widely used micro-controllers. Speed estimates were gathered using the macro-modeling technique by Suzuki et.al. [19].
2. ARM Software Development Kit (SDKv2.50) profiler for the ARM9 (160 MHz ARM 920T) family of Harvard architecture RISC processors (ARMulator) [2].
3. GNU's *gcc* compiler (with the highest level of optimization) for an Alpha 21164 64-bit (400 MHz) processor: representing the high end 32-bit and 64-bit RISC processors such as those cited in the EEMBC suite [11] since *gcc* has been ported to most of these embedded processors in the form of the Cygnus GnuPro compiler [10]. Speed estimates were obtained from the `pixie` instrumentation of the ATOM analysis Tool [18].

In order to give an idea of the proportion factor between the reported CDFG node count and the final output byte size, we also show the bytes obtained after synthesis on the *gcc* platform.

6.2 Benchmarks

We present in this section results on two benchmarks: Quick Sort and Insertion Sort fragment examples used by Aho et. al.[1]. We have adapted the examples to be reactive by periodically reading from, and writing to, an external memory block. These examples have been used quite extensively in the software optimization domain, and we hope they will serve here to show two cases where it is obvious that current optimization for synthesis techniques are outperformed by our proposed high-level design representation optimization followed by CDFG building, and synthesis approach. We also intend for these examples to demonstrate the benefits of optimization on control-dominated designs that have "intelligence" embodied in a significant data computation portion. Synthesis results are displayed in Table 1. It can be seen from the data that the representation and consequent optimizations are able to improve considerably final code quality in all cases. We can also see a case (*Insertx* task) where the high level optimizations prevented *design explosion.* Our experience has shown that this is a quite common occurrence in hardware synthesis.

SYNTHESIS & MAPPING METHOD	EFSM Task (GCC BYTES)	CDFG (NODES)	GCC (CYCLES)	ARM (CYCLES)	INTROL (CYCLES)
EFSM→CDFG	Quick (24952)	327	389	274	1165
EFSM→FFG→ CDFG	*Quick (15320)*	*246*	*111*	*186*	*494*
% Improvement	**38.6**	**25**	**71.5**	**32.1**	**58**
EFSM→CDFG	Insertx (-)	> 4000	Design exploded (very large SHIFT file)		
EFSM→CLIF→ CDFG	*Insertx (22040)*	*522*	*115*	*108*	*607*
% Improvement	-	**> 87**	-	-	**-**

Table 1. Benchmark Results

6.3 Application Design Example

The optimization technique described in this paper has been applied to the synthesis of an industrial case study from the communication networks domain: an ATM server suitable for implementing Virtual Private Networks (VPN) in ATM nodes [12]. Results are shown for 3 design tasks in Table 2. The automatically generated Real-Time Operating System RTOS [4] size was reduced by about 6% in the optimized design.

SYNTHESIS & MAPPING METHOD	EFSM Task (GCC BYTES)	CDFG (NODES)	GCC (CYCLES)	ARM (CYCLES)	INTROL (CYCLES)
EFSM→CDFG	quid (7960)	14	119	496	229.5
EFSM→FFG→ CDFG	*quid (6912)*	*11*	*110*	*490*	*221*
% Improvement	**13.2**	**21.4**	**7.6**	**1.2**	**3.7**
EFSM→CDFG	first cell (13072)	137	248	350	734
EFSM→FFG→ CDFG	*first cell (12152)*	*125*	*212*	*246*	*631*
% Improvement	**7.1**	**8.8**	**14.5**	**29.7**	**14**
EFSM→CDFG	extract (8032)	100	104	166	313.5
EFSM→FFG→ CDFG	*extract (7488)*	*88*	*103*	*154*	*298*
% Improvement	**6.8**	**12**	**0.96**	**7.2**	**4.9**

Table 2. ATM Server Design Results

Results here also show a measurable improvement in code quality reflected in both size and performance in all cases. The effect of the instruction selection and allocation performed by the target compiler at the low level are *more apparent* here because of the relatively small returns on the high level optimizations since the design is mostly control. Our experience shows that in such designs guiding low level optimizations, such as register allocation for example, with proper "hints" at the high level has a tremendous impact on size and performance.

Overall experimental results are therefore very encouraging and have shown a considerable advancement in the quality of the

synthesized software and hardware on the order of about 10-20% on control-dominated examples with greater improvement results the more abundant data computations are.

7 Conclusions

Boosting design productivity cannot be the major focus of system level co-design tools; attention should be paid as well to enhancing the quality of the final synthesized output that will run on the target architecture. We have shown that design representation level data flow and control optimizations have a considerable positive improvement effect on synthesized software. Our experience indicates that hardware synthesis benefits greatly too. The optimizations at the high level assist the lower (abstraction) level optimization algorithms and heuristics in the co-design flow as well. This is because typically the lower level algorithms deal with smaller granularity optimizations and therefore perform better on smaller inputs. It should be noted, as well, that the computation cost of the data flow and control analysis and optimization to improve synthesis quality should not be viewed as an overhead since it is most often recovered by the significant speed up of the low level synthesis techniques (e.g. building the CDFG, an $O(n^2)$ process in the number of FFG nodes, is up to 80% faster in the Quick Sort benchmark).

8 Future Work

While we have focused here on optimizing the function of an EFSM task in a system composed of such interconnected elements, we are working towards exploring opportunities in *Function Architecture Co-design*. An Attributed version of the FFG (AFFG) 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, at the *macro*-architectural level (e.g. partitioning control by decomposing the single reactive controller into several smaller control fragments) and then the *micro*-architectural level, (e.g. through the use of instruction selection techniques) for implementation. On the other hand, architectural constraints specified by the user can feed further optimization opportunities back to the EFSM function during the architecture dependent optimization and mapping step, and the assistant can then attempt to massage the function to meet these constraints (e.g. operator strength reduction can be performed on costly operations in the selected target architecture).

Acknowledgements

Our thanks go to Richard Newton and Robert Brayton from the University of California at 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 Semiconductor Research Corporation (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] ARM Limited, http://www.arm.com, 1999.

[3] 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*, May 1997.

[4] Balarin F., Chiodo M., Lavagno L., Jurecska A., Tabbara B., A. SangiovanniVincentelli, "Automatic Generation of a Real-Time Operating System for Embedded Systems", CODES/CASHE '97, Braunschweig, Germany, March 1997.

[5] Bergamaschi, R. A., "Behavioral Network Graph: Unifying the Domains of High-Level and Logic Synthesis", *DAC*, 1999.

[6] Berry, G., Couronné, P., Gothier, G. "The Synchronous Approach to Reactive and Real-Time Systems", *IEEE Proceedings*, 79, September 1991.

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

[8] 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.

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

[10] Cygnus GNU, http://www.cygnus.com, 1999.

[11] The EDN Embedded Microprocessor Benchmark Consortium (EEMBC), http://www.eembc.org, 1999.

[12] Filippi E., Lavagno L., Licciardi L., Montanaro A., Paolini M., Passerone R., Sgroi M., Sangiovanni-Vincentelli, A. "Intellectual Property Re-use in Embedded System Co-design: an Industrial Case Study", *ISSS*, Dec. '98.

[13] Hanono, S.; Devadas, S., "Instruction Selection, Resource Allocation, and Scheduling in the Avis Retargetable Code Generator", *DAC*, 1998, pp. 510-515.

[14] Kam, J.B., Ullman, J.D. "Monotone Data Flow Analysis Frameworks", *Acta Informatica*, 1977, pp. 305-307.

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

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

[17] 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.

[18] Srivastava, Amitabh and Alan Eustace, "ATOM: A System for Building Customized Program Analysis Tools" *SIGPLAN*, 1994.

[19] Suzuki, K. and Sangiovanni-Vincentelli, A. "Efficient Software Performance Estimation Methods for Hardware/Software Co-design" *DAC*, pp. 605-610, June 1996.

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

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