Presented at *the Workshop on Compiler and Architecture Support for Embedded Systems*, Washington, D. C., October, 1999.

Optimization Trade-offs in the Synthesis of Software for Embedded DSP

Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park
ssb@eng.umd.edu

Abstract

A broad variety of implementation considerations become important when implementing software for embedded signal processing applications. These include complex trade-offs such as those involving code size, data buffering requirements, performance, and power consumption. This presentation motivates the use of high-level, dataflow-based programming models for automated synthesis of DSP software; presents an overview of optimization issues and trade-offs that arise in uniprocessor software synthesis from dataflow specifications; describes useful techniques that have been developed to address some of these trade-offs; and motivates important problems in this area for further investigation.

1. Introduction

The implementation of software for embedded digital signal processing (DSP) applications is an extremely complex process. The complexity arises from escalating functionality in the applications; intense time-to-market pressures; and stringent cost, power and speed constraints. To help manage such complexity, DSP system designers have increasingly been employing high-level, graphical design environments in which system specification is based on hierarchical dataflow graphs. Consequently, a significant industry has emerged for the development of dataflow-based DSP design environments. Leading products in this industry include SPW (Cadence), COSSAP (Synopsys), DSP Station (Mentor Graphics), and the Advanced Design System (Hewlett Packard).

In this proposed talk, I will discuss the state-of-the-art in software synthesis technology for high-level, dataflow-based DSP design environments, and motivate the wide range of open challenges that remain in this area. Dataflow-based computational models provide block-diagram semantics that are both intuitive to DSP system designers, and efficient from the point of view of embedded software synthesis. In dataflow, a computational specification is represented as a directed graph in which vertices (*actors*) specify computational functions, and edges specify communication between functions. Actors can be of arbitrary complexity. Typically, they range in complexity from elementary operations such as addition or multiplication to DSP subsystems such as FFT units or adaptive filters.

A dataflow edge represents a FIFO (first-in-first-out) queue that buffers data as it passes from the output of one actor to the input of another. When dataflow graphs are used to represent signal processing applications, a dataflow edge e has a non-negative integer delay $delay(e)$ associated with it. The delay of an edge gives the number of initial data values that are queued on the edge.

Synchronous dataflow (SDF) [9] is the simplest and most popular form of dataflow for DSP. The SDF model imposes the restriction that the number of data values produced by an actor onto each output edge is constant, and similarly the number of data values consumed by an actor from each input edge is

constant. A broad class of important signal processing applications conform to the SDF model [3]. All of the commercial design tools listed above employ SDF or very closely related semantics.

A simple example of an SDF graph is shown in Figure 1. Each edge is annotated with the number of tokens produced by the source actor and the number consumed by the sink actor. The "5D" above edge (X, Y) denotes a delay of magnitude 5.

A *periodic schedule* for an SDF graph is a schedule that invokes each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. Typically, an SDF graph is compiled by encapsulating the code for a period schedule within an infinite loop. Given an SDF graph $G = (V, E)$, the number of times $x(A)$ to invoke each actor $A \in V$ in a periodic schedule can be determined by solving the so-called *balance equations* for G:

$$x(src(e))p(e) = x(snk(e))c(e), \text{ for all } e \in E, \quad (1)$$

where $src(e)$ and $snk(e)$ denote the source and sink actors of e; and $p(e)$ and $c(e)$ denote the numbers of data values produced by $src(e)$ and consumed by $snk(e)$, respectively. A minimal solution to (1) is called the *repetitions vector*, and is denoted by the symbol q. For example, the repetitions vector for Figure 1 is given by $q(X) = 1, q(X) = q(Y) = 10$.

2. Synthesis trade-offs

Software synthesis tools generate application programs by piecing together code modules from a predefined library of software building blocks. These code modules are defined as specifications in the target language of the synthesis tool. Most SDF-based design systems use a model of synthesis called *threading*. Given an SDF representation of a block-diagram program specification, a threaded synthesis tool begins by constructing a periodic schedule. The synthesis tool then steps through the schedule and for each actor instance A that it encounters, it inserts the associated code module M_A from the given library (*inline threading*), or inserts a call to a subroutine that invokes M_A (*subroutine threading*). Threaded tools may employ purely inline threading, purely subroutine threading, or a mixture of inlined and subprogram-based instantiation of actor functionality (*hybrid threading*). The sequence of code modules and subroutine calls that is generated from a dataflow graph is processed by a buffer management phase that inserts the necessary target program statements to route data appropriately between actors.

The scheduling phase and the selection of the *threading mode* (inline, subroutine, or hybrid) for each actor invocation have a large impact on key implementation metrics. Major metrics involved in this trade-off include the code size cost, the buffering cost (the amount of memory allocated to accommodate inter-actor data transfers), and a number of overhead components that affect execution time and power consumption. These overhead components include loop initiation and loop iteration overhead, subroutine overhead, and overhead due to inter-actor context switching. Inter-actor context switching overhead can be estimated by the number of *actor activations* per schedule period (an activation occurs whenever two distinct actors are invoked in succession) [11].

Even for a simple SDF graph, the underlying range of trade-offs may be very complex and exhibit complex dependency on target processor and actor library characteristics. For example, consider the SDF graph in Figure 1. Several threading mode/schedule combinations for this graph are given in Table 1, along with expressions for each of the implementation metrics described above. The last column gives the num-

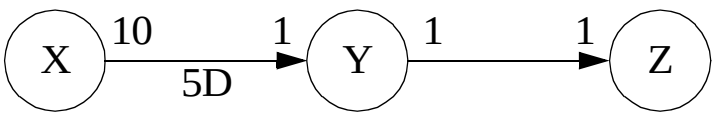

Figure 1. A simple SDF graph.

Schedule	Threading Mode	Code Size	Buffering Cost	Loop Overhead	Subroutine Overhead	Actor Activations
$YZYZYZYZYZX$ $YZYZYZYZYZ$	I	$\kappa(X) + 10\kappa(Y) + 10\kappa(Z)$	11	0	0	21
$YZYZYZYZYZX$ $YZYZYZYZYZ$	S	$\kappa(X) + \kappa(Y) + \kappa(Z)$ $+21\sigma_c$	11	0	$21\sigma_t$	21
$(5YZ)X(5YZ)$	I	$\kappa(X) + 2\kappa(Y) + 2\kappa(Z)$ $+2L_c$	11	$2L_s + 10L_i$	0	21
$(5YZ)X(5YZ)$	S	$\kappa(X) + \kappa(Y) + \kappa(Z)$ $+2L_c + 5\sigma_c$	11	$2L_s + 10L_i$	$21\sigma_t$	21
$X(10Y)(10Z)$	I	$\kappa(X) + \kappa(Y) + \kappa(Z)$ $+2L_c$	25	$2L_s + 20L_i$	0	3
$X(10YZ)$	I	$\kappa(X) + \kappa(Y) + \kappa(Z) + L_c$	16	$L_s + 10L_i$	0	21

Table 1. An illustration of software synthesis trade-offs for Figure 1.

ber of actor activations. Here, L_s and L_i denote the run-time loop startup and loop iteration overhead of the target processor; L_c denotes the code size overhead of a loop; σ_t (σ_c) denotes the run-time (code size) overhead of a subroutine call; and $\kappa(A)$ denotes the program memory cost of the library code module for actor A. Depending on the target processor, actor library, and threading implementation, some or all of these parameters may not be known precisely; in such cases, reasonable estimates can often be derived.

Parenthesized terms in the schedules (column 1 of Table 1) indicate repetitive invocation patterns that are to be translated into loops in the target code. More precisely, a parenthesized term (*schedule loop*) of the form $(nT_1T_2\ldots T_n)$ specifies the successive repetition n times of the subschedule $T_1T_2\ldots T_n$. Schedules that contain only one appearance of each actor, such as those listed in the bottom two rows of Table 1 are referred to as *single appearance schedules*. Because of their code size optimality, and because they have been shown to satisfy a number of useful formal properties, single appearance schedules have been the focus of a significant component of work in DSP software synthesis.

The schedules compared in Table 1 show a limited subset of the design space for pure inline or pure subroutine threading modes. Depending on the parameters of the target processor and actor library, and depending on the relevant implementation objectives, *any one* of the schedules listed in Table 1 can be a Pareto point of the overall design space. Hybrid threading introduces numerous additional points in the design space. Even for this simple example, the overall design space is large and complex. The most desirable implementation strategy varies strongly with the exact requirements of the application.

3. Optimization techniques

To date, software synthesis from SDF graphs has centered largely around the problem of generating memory-efficient implementations. Minimizing program and data memory requirements is critical in many embedded DSP applications. On-chip memory capacities are limited, and the speed, power, and financial cost penalties of employing off-chip memory may be prohibitive or highly undesirable. Three general avenues have been investigated for minimizing memory requirements — minimization of the buffering cost, which usually forms a large component of the over all data space cost; minimization of code size; and joint exploration of the trade-off involving code size and buffering cost.

It has been shown that the problem of constructing a schedule that minimizes the buffering cost over all periodic schedules is NP-complete [3]. Thus, for practical, scalable algorithms, we must resort to heuristics. Ade, Lauwereins and Peperstraete [1] have developed techniques for computing tight lower bounds on the buffering cost for a number of restricted subclasses of delayless, acyclic graphs, including arbitrary-length chain-structured graphs. Some of these bounds have been generalized to handle delays in [3]. Cubric and Panangaden have presented an algorithm that achieves optimum buffering costs for acyclic SDF graphs that may have one or more independent, undirected cycles [6]. An effective heuristic for general graphs, which is employed in the Gabriel [8] and Ptolemy [5] design environments is given in [3]. Govindarajan, Gao, and Desai have developed an SDF buffer minimization algorithm for multiprocessor implementation [7]. This algorithm minimizes the buffering cost over all multiprocessor schedules that have optimal throughput.

To date, research on the second avenue of SDF memory minimization — code size minimization — has focused largely on single appearance schedules. A periodic single appearance schedule exists for any properly-constructed, acyclic SDF graph. Furthermore, a periodic single appearance schedule can be derived easily from any topological sort of an acyclic graph G: if $A_1, A_2, \ldots, A_m$ is a topological sort of G, then it is easily seen that the single appearance schedule $(q(A_1)A_1)(q(A_2)A_2)\ldots(q(A_m)A_m)$ is periodic. For a cyclic graph, a single appearance schedule may or may not exist depending on the location and magnitude of delays in the graph. An efficient strategy, called the *loose interdependence algorithm framework (LIAF)*, has been developed that constructs a single appearance schedule whenever one exists [3]. Furthermore, for general graphs, this approach guarantees that all actors that are not contained in a certain type of subgraph, called *tightly interdependent subgraphs*, will have only one appearance in the generated schedule. In practice, tightly interdependent subgraphs arise only very rarely, and thus, the LIAF technique guarantees full code size optimality for most applications.

The LIAF constructs a single appearance schedule by decomposing the input graph into a hierarchy of acyclic subgraphs, which correspond to an outer-level hierarchy of nested loops in the generated schedule. The acyclic subgraphs in the hierarchy can be scheduled with any existing algorithm that constructs single appearance schedules for acyclic graphs. The particular algorithm that is used in a given implementation of the LIAF is called the *acyclic scheduling algorithm*. For example, the topological-sort-based approach described above could be used as the acyclic scheduling algorithm. However, this simple approach has been shown to lead to relatively large buffering costs. This motivates a key problem in the joint minimization of code and data for SDF specifications. This is the problem of constructing a single appearance schedule for an acyclic SDF graph that minimizes the buffering cost over all periodic single appearance schedules. Since any topological sort leads to a distinct schedule for an acyclic graph, and the number of topological sorts is not polynomially bounded in the graph size, exhaustive evaluation of single appearance schedules is not tractable. Thus, as with the (arbitrary appearance) buffer minimization problem, heuristics have been explored. Two complementary, low-complexity heuristics, called APGAN and RPMC [3], have proven to be effective on practical applications when both are applied, and the best resulting schedule is selected. Furthermore, it has been shown that APGAN gives optimal results for a broad class of SDF systems [4].

Although APGAN and RPMC provide good performance on many applications, these heuristics can sometimes produce results that are far from optimal [14]. Furthermore, DSP software tools are often allowed to spend more time for optimization of code than what is required by low-complexity, deterministic algorithms such as APGAN and RPMC. Motivated by these observations, Zitzler, Teich, and Bhattacharyya have developed an effective stochastic optimization methodology, called GASAS, for constructing minimum buffer single appearance schedules [14][15]. The GASAS approach is based on a genetic algorithm formulation in which periodic schedules are encoded as “chromosomes,” which randomly “mutate” and “recombine” to explore the search space. To exploit the valuable optimality property of APGAN whenever it applies, the solution generated by APGAN is included in the initial population, and an *elitist* evolution policy [2] is enforced to ensure that the fittest individual always survives to the next generation.

At the Aachen University of Technology, as part of the COSSAP design environment (now devel-

oped by Synopsys) project, Ritz, Pankert, and Meyr have investigated the minimization of the context-switch overhead, or the average rate at which actor activations occur [11]. Activation overhead includes saving the contents of registers that are used by the next actor to invoke, if necessary, and loading state variables and buffer pointers into registers. The concept of grouping multiple invocations of the same actor together to reduce context-switch overhead is referred to as *vectorization*. The *scalable SDF (SSDF)* programming/code generation model [10] allows the benefits of vectorization to extend beyond the actor interface level (inter-actor context switching). For example, context switching between successive sub-functions of a complex actor can be amortized over N_v invocations of the sub-functions, where N_v is the given vectorization parameter.

A general, optimal algorithm for vectorization of SSDF graphs is given in [11]. Joint minimization of vectorization and buffering cost is developed in [12]. Adaptation of the retiming transformation to improve vectorization for multirate SDF graphs is addressed in [17].

The techniques discussed above assume a fixed threading mode. In particular, they do not attempt to exploit the flexibility offered by hybrid threading. Sung, Kim, and Ha have developed an approach that employs hybrid threading to share code among different actors that have similar functionality [13]. This hybrid threading approach incorporates a method for systematically expanding a schedule to trade-off increases in code size for decreased buffering cost. Also, the GASAS framework has been significantly extended to consider multiple appearance schedules, and apply hybrid threading to reduce the code size cost of highly irregular schedules, which cannot be accommodated by compact loop structures [16]. Such irregularity often arises when exploring the space of schedules whose buffering cost is significantly lower than what is achievable by a single appearance schedule [3].

We will present an overview of the optimization techniques mentioned above, and conclude with a discussion of open issues in the area of optimized software synthesis for DSP. Relationships with ongoing efforts on DSP code generation from procedural programming languages, and with conventional general-purpose compiler techniques will also be discussed.

4. References

[1] M. Ade, R. Lauwereins, and J. A. Peperstraete. “Buffer memory requirements in DSP applications.” In *Proceedings of the IEEE Workshop on Rapid System Prototyping*, June 1994, pp. 198–123.

[2] T. Back, U. Hammel, and H. Schwefel. “Evolutionary computation: Comments on the history and current state.” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, 1997.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. “APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations.” *Journal of Design Automation for Embedded Systems*, January 1997.

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. “Ptolemy: A framework for simulating and prototyping heterogeneous systems.” *International Journal of Computer Simulation*, January 1994.

[6] M. Cubric and P. Panangaden. “Minimal memory schedules for dataflow networks.” In *CONCUR ’93*, August 1993.

[7] R. Govindarajan, G. R. Gao, and P. Desai. “Minimizing memory requirements in rate-optimal schedules.” In *Proceedings of the International Conference on Application Specific Array Processors*, August 1994.

[8] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya. “Gabriel: A design environment for DSP.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, November 1989.

[9] E. A. Lee and D. G. Messerschmitt. “Synchronous dataflow.” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[10] S. Ritz, M. Pankert, and H. Meyr. “High level software synthesis for signal processing systems.” In *Proceedings of the International Conference on Application Specific Array Processors*, August 1992.

[11] S. Ritz, M. Pankert, and H. Meyr. “Optimum vectorization of scalable synchronous dataflow graphs.” In *Proceedings of the International Conference on Application Specific Array Processors*, October 1993.

[12] S. Ritz, M. Willems, and H. Meyr. “Scheduling for optimum data memory compaction in block diagram oriented software synthesis.” In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, May 1995.

[13] W. Sung, J. Kim, and S. Ha. “Memory efficient synthesis from dataflow graphs.” In *Proceedings of the International Symposium on Systems Synthesis*, 1998.

[14] J. Teich, E. Zitzler, and S. S. Bhattacharyya. “Optimized software synthesis for digital signal processing algorithms — an evolutionary approach.” Tech. Rep., Computer Engineering and Communication Networks Laboratory, Swiss Federal Institute of Technology, Zurich, January 1998.

[15] E. Zitzler, J. Teich, and S. S. Bhattacharyya. “Evolutionary algorithms for the synthesis of embedded software.” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. Accepted for publication; to appear.

[16] E. Zitzler, J. Teich, and S. S. Bhattacharyya. “Multidimensional exploration of software implementations for DSP algorithms.” *Journal of VLSI Signal Processing Systems*. Accepted for publication; to appear.

[17] V. Zivojnovic, S. Ritz, and H. Meyr. “Retiming of DSP programs for optimum vectorization.” In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1994.