Memory Efficient Software Synthesis from Dataflow Graph*

Wonyong Sung, Junedong Kim, Soonhoi Ha
Codesign and Parallel Processing Laboratory
Seoul National University
E-mail : {yong,june8th,sha}@comp.snu.ac.kr

Abstract

Due to the limited amount of memory resources in embedded systems, minimizing the memory requirements is an important goal of software synthesis. This paper presents a set of techniques to reduce the code and data size for software synthesis from graphical DSP programs based on the synchronous dataflow (SDF) model. By sharing the kernel code among multiple instances of a block, we can further reduce the code size below the single appearance schedule. And, a systematic approach is presented to give up single appearance schedules to reduce the data buffer requirements. Experimental results from two real examples prove the significance of the proposed techniques.

1 Introduction

Minimizing the memory requirements is very important to synthesize code for embedded systems due to the limited amount of memory. Especially, in an on-chip design, extra memory requirements over the on-chip memory size incur additional memory cost, performance penalty, and drastic increase of power consumption. Critical constraints on the memory size have made assembly programming still a popular way of software synthesis for embedded systems in spite of low productivity.

Growing complexity of embedded systems, fast design turn-around time, limited development budget, and short life cycle of products, however, will make the use of high level software design methodology mandatory: high level language compiler or automatic code generation from block diagram specification.

In this paper, we aim to reduce the code and data size for software synthesis from graphical DSP programs based on the synchronous dataflow (SDF) model, one of block diagram specification models. An SDF graph is a coarse grain dataflow where each node contains a kernel (code fragment) of a host language tailored to an implementation engine while the dataflow graph itself is a coordination language among function modules. Numerous DSP design environments including a number of commercial tools support SDF or closely related models ([1],[3],[4]) for both simulation and code generation.

Software synthesis from an SDF graph includes determining a feasible schedule and a coding style, both of which affect the memory requirements of the generated software for code and data. One of main scheduling objectives for software synthesis is to minimize the memory requirements. Once the schedule is determined, codes are generated according to the scheduled sequence. Since nodes are prepared in libraries, the kernel inside a node is assumed already optimized and treated as a unit. Two popular coding styles are inlining and functions. The former generates an inline code for each node at the scheduled position while the latter calls a function that contains the kernel.

Previous approaches first assume a coding style and determine an optimal schedule afterwards. Also, they try to minimize the code size first and the data size later. Even though they produce good results for a set of applications, they could not produce good codes for some applications which we will demonstrate.

In this paper, we propose a pair of optimization techniques to overcome their limitations by mixing the coding style. The first technique is to reduce the code size by sharing the kernel among multiple instances of the same block; which requires function style code generation instead of inlining. The second technique is to give up single appearance schedule, an important schedule class for the minimum code size, for data memory minimization. By applying these two techniques, we could reduce the memory requirements of two important examples by 10% and 23% over the best results from SAS[7].

In section 2, we review the previous works. The proposed techniques will be explained in section 3 and 4. Two real life examples will be discussed in section 5. Section 6 will wrap up the paper with conclusions and future works.

*This study was supported by the academic fund of Ministry of Education, Republic of Korea,through Inter-University Semiconductor Research Center (ISRC-98-E-2103) in Seoul National University.

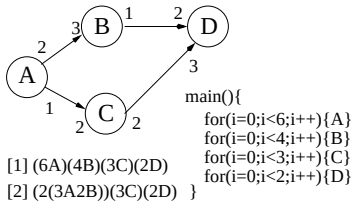

Figure 1. An SDF example

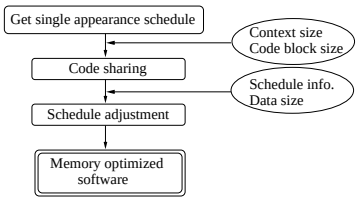

Figure 2. Optimizations in software synthesis

2 Previous Works and Our Strategy

Figure 1 is an example SDF graph, and each arc is annotated with the number of tokens produced or consumed by an invocation(activation) of its source or destination node. Each arc is assigned to a data buffer whose size is equal to the maximum number of tokens accumulated during execution of the graph. These data buffers compose the state of the SDF graph. We call an SDF graph *consistent*, when there exists a *valid cyclic schedule* which returns the graph to its original state after every repetition. A valid schedule fires each node at least once, does not deadlock, and produces no net change in the number of tokens queued on each arc. A schedule Σ is a sequence of node executions. For example, Σ_1=(6A)(4B)(3C)(2D) and Σ_2=(2(3A2B))(3C)(2D) represent two valid schedules for figure 1. Here, a parenthesized term (nS) specifies n successive firings of the subschedule S and such a term is used to be translated into a loop in the target code[5]. Among valid schedules for a consistent SDF graph, if every block appears exactly once in Σ, the schedule Σ is called **single appearance schedule(SAS** or **SA-schedule)**. Since each node has a kernel(code block) inside, if select the Σ_1, the generated C program is shown in figure1.

Since a single appearance schedule guarantees the minimum code size for inline code generation, a group of researches are focused on finding a single appearance schedule which minimize the data memory requirements. Bhattacharyya et. al. developed two heuristics: APGAN and RPMC, to find a schedule that minimizes the data memory requirements[6]. Ritz et. al. used an ILP formulation to minimize the data memory[10]. Their approach is different from Bhattacharyya's in that they allow data buffer sharing based on flat single appearance schedule. Since a flat SA-schedule usually requires more data buffer than the optimal nested SA-schedule, the advantage of sharing data buffer is not evident in general. Both works([6], [10]) stick to single appearance schedule and do not exploit code sharing optimization, which is of main interest in this paper.

Another group of researches try to minimize only data memory. Ade et. al. present an algorithm to determine the smallest possible data buffer size for arbitrary SDF applications[8]. Though their work is mainly targeted for mapping an SDF application onto Field Programmable Gate Array (FPGA) in their GRAPE environment, the efforts to compute the lower bound on buffer requirement can be applicable to software synthesis. Govindarajan et. al.[9] developed a rate optimal compile time schedule, which minimizes the buffer requirement using linear programming formulation. Both do not discuss the code size, which is likely to be more important for memory requirement. Even though the inline coding style is preferred in those previous researches, we propose the use of functions for the nodes whose kernel would be repeated several times in the inline code. As a result, the proposed method generates a code mixed with inline kernels and functions.

We start with a single appearance schedule obtained by the method described in [6]. Then, we apply our optimization techniques as shown in figure 2. In the code sharing optimization, we investigate the schedule to find multiple instances of a same block. Multiple instances are treated as different nodes in a single appearance schedule. The code sharing technique described in section 3 formulates the gain and the overhead of function code over inline code using the context size and code block size. Only when the gain is greater than the overhead, we make a function for the sharing block.

In the next phase of schedule adjustment, we give up the SA-schedule to further reduce the data size if the gain is greater than the overhead. We express the SA-schedule with the BTLC(Binary Tree with Leaf Chain) data structure. From BTLC, we could identify the possible location of schedule adjustment and obtain adjusted schedule.

3 Code Sharing Optimization

In an inline code from a single appearance schedule, multiple instances of the same block are regarded as different blocks, and the same kernel, possibly with different local states, may appear several times in the generated code. We propose a technique to share the same kernel using a function in this section. Figure 3 contains an example, which is a sample rate converter from compact disc to digital audio tape. Figure 3(b) depicts one of the library blocks: fir filter. Since there are four instances of the fir filter, it becomes a candidate of code sharing. Each fir filter has its own state values such as tab values. Also, each input or

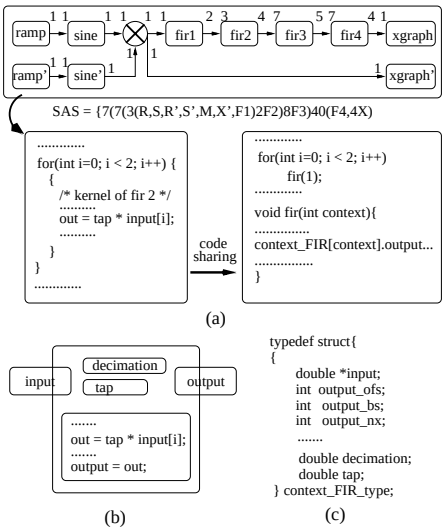

Figure 3. CD2DAT example: from inline single appearance schedule to code shared function code

output port of an instance is bound to its own buffer.

Separate state variables and buffers should be maintained for each instance, in case the code is shared among four instances. They define the **"context"** of each instance. An example of the context of the fir filter is depicted in figure 3(c). Two generated codes are shown in the figure 3(a); one is an inline style with single appearance schedule and the other is after code sharing is applied to the fir block.

To decide whether a code block had better be shared, we compute the overhead and the gain of sharing. If Δ is an overhead, Ω is a code block size, and α is the number of instances of a block, the decision function is summarized as the following inequality.

$$1 > \frac{\Delta}{(\alpha - 1)\Omega} \tag{1}$$

The overhead incurred by code sharing comes from additional data structure : **context**. We compute the sharing overhead Δ by dividing into two parts; a context size overhead in the data block and the reference code overhead in the code memory.

$$\Delta = \Delta_{context} + \Delta_{reference} \tag{2}$$

In an implementation point of view, a context includes pointers to input and output token buffers and state variables. Since the state variables are also needed for each instance in the inline code, the context size overhead includes only per-port overhead. At most three integer variables and a pointer variable are needed per port. For the multirate computation, a port is implemented with a buffer array. To point the next read or write location in the array, an offset is needed; the offset is an index of the array. Two more integers are needed to delimit the end of the array and to describe the offset increment after each activation. Therefore, the per-port overhead λ and the total context size overhead of a block are computed using the next two equations.

$$\lambda(x) = 3 * sizeof(int) + sizeof(pointer) \tag{3}$$

$$\Delta_{context} = \alpha \times \Sigma_{p_i} \lambda(p_i),\ p_i \in ports \tag{4}$$

A reference overhead is an overhead resulting from accessing a port or state through the context structure. When we access states or ports via the context structure, we need additional codes. Below shows the difference of two access methods in two simple assignment statements. The first statement is when sharing is not used, the second is when sharing is used.

```
... = value;
... = *(context_CGCRamp[context].value);
```

An assembly list is obtained through compiling with the gcc compiler in a Sparc/Solaris machine. The first assignment with direct memory access is compiled into the first assembly line, while the second assignment results in 10 assembly lines.

```
; without context
    ldd [%fp + -336],%o0
; through context
    sethi  %hi(0x20800), %o1
    ld  [ %o1 + 0x3c8 ], %o0
    mov  %o0, %o2
    sll  %o2, 2, %o1
    add  %o1, %o0, %o1
    sll  %o1, 3, %o0
    add  %fp, -424, %o1
    add  %o1, %o0, %o2
    ld  [ %o2 + 0x1c ], %o0
    ldd  [ %o0 ], %o2
```

In machine code, the sizes of the two parts are 4 bytes and 40 bytes, respectively. Thus, we can define the per-reference overhead as 36 bytes. Although the overhead may be reduced after compiler optimization, we use the worst value to be conservative.

The reference overhead is dependent on the variable type as well as whether it is a port or a state. We consider three variable types: scalar type such as integer or double, array, and constant. Although a more detailed classification may result in more accurate overhead estimation, we consider six combinations only in this paper.

Among the six combinations, the scalar and the constant types have the same overhead both for port and state. As a result, we define the overhead cost δ as a function of the

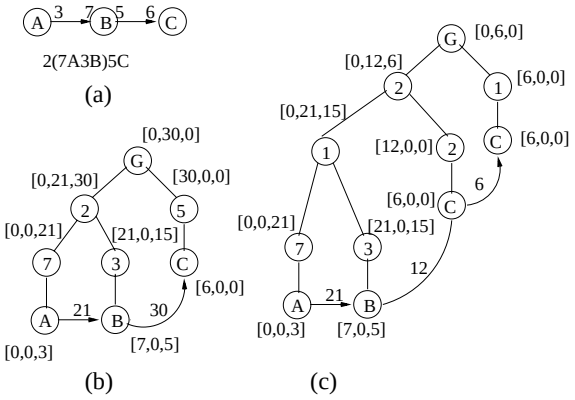

Figure 4. Schedule adjustment using BTLC

reference type.

$$\delta(t) = \begin{cases} 36 & \text{if t = scalar(S)} \\ 32 & \text{if t = constant(C)} \\ 60 & \text{if t = array,state (AS)} \\ 128 & \text{if t = array,port (AP)} \end{cases} \quad (5)$$

By counting the reference counts in a code block according to the reference type, we compute the reference overhead $\Delta_{reference}$,where $\eta(t)$ is the reference count of type t in the kernel.

$$\Delta_{reference} = \Sigma_{t\in\{S,C,AS,AP\}}(\eta(t) \times \delta(t)) \quad (6)$$

The constants we use in equations are highly machine dependent; the size of types and addressing mode of the processor will be a great concern. Since we can obtain the constant numbers easily from manuals or simple test program, our technique is applicable to other than Sparc/Solaris environment,

In summary, we can compute the code sharing overhead of a block using the port count and the reference counts, which can be easily obtained from the kernel of the node. From equation (1),(2) and (4), it is obvious that if the context overhead is greater than the kernel size like the ramp block in figure 3(a), the node may not be shared.

4 Adjusting Single Appearance Schedule

4.1 Construction of BTLC

To compute a new schedule which requires less data memory than SAS, we devise a data structure BTLC(Binary Tree with Leaf Chain) to express a schedule and its buffer requirements as depicted in figures 4 and 6. Each leaf node of a BTLC corresponds to an SDF node which contains a library block(kernel). A loop count γ [1] is represented by an

[1]Since 'γ' successive firings of the subschedule 'S' is translated into a loop in the target code, we call 'S' a loop cluster(shortly cluster) and 'γ' a loop counter.

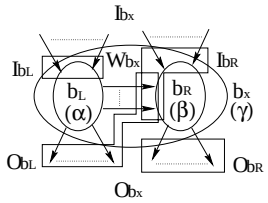

Figure 5. Two clusters and its I/O

intermediate node of a tree. In the figures 4 and 6, the loop counter γ is used as the label of each intermediate node. If we visit each node of a BTLC with depth first search(DFS) method, we obtain a single appearance schedule of the original SDF graph, such as 2(7A3B)5C in the figure 4(b).

Every node of a BTLC maintains the buffer requirement information as a tuple **[I, W, O]**: the set of input buffers, the buffer requirement between child nodes, and the set of output buffers. For compact representation, we use a tuple $[|\mathbf{I}|, |\mathbf{W}|, |\mathbf{O}|]$ rather than **[I, W, O]** in the figure 4 and 6. The **I** and **O** of a leaf node become the input and output buffers of corresponding node in SDF, and **W** becomes null. The **I** and **O** of an intermediate node are the set of input and output buffers of a cluster, and **W** is the buffer produced and consumed within the cluster. From the tuples of leaf nodes, other tuples are computed in a bottom up manner. For an intermediate node b_x, which has two child nodes,b_L and b_R as shown in figure 5, we compute the tuple $[I_{b_x}, W_{b_x}, O_{b_x}]$ with the following equations.

$$W_{b_x} = |O_{b_L} \cap I_{b_R}| \quad (7)$$

$$I_{b_x} = \gamma \times |I_{b_L} \cup I_{b_R} - W_{b_x}| \quad (8)$$

$$O_{b_x} = \gamma \times |O_{b_L} \cup O_{b_R} - W_{b_x}| \quad (9)$$

Unless the $|\mathbf{I}|$ and $|\mathbf{O}|$ of the root node are zero, the corresponding SDF graph is not consistent.

Chains along the leaf nodes represent the firing order in the schedule. The weight of a chain is the $|\mathbf{W}|$ value of the first common ancestor of two end nodes. The total sum of chain weights represents the buffer requirement needed by the schedule which the BTLC represents($|BTLC|$). Figure 4(b) shows the BTLC of an SDF graph and its schedule shown in 4(a), and its $|BTLC|$ is 51.

4.2 Schedule Adjustment using BTLC

In figure 4(a), the buffer requirement between node B and C is 30 as shown in figure 4(b). If we give up SAS and construct a schedule as 2(7A3B2C)C, the buffer requirements is reduced to 18. In this section, we show how to obtain the reduced buffer requirements systematically by splitting a chain.

A given schedule is adjusted to reduce total memory requirement. If we select a chain as an adjustment point, there

are two sub-clusters C_L and C_R connected by the chain, as shown in figure 5. Let α and β be the loop counts of them and α be smaller than β. To reduce the buffer requirements between C_L and C_R, we merge the C_R into C_L. The merged portion of C_R has a new loop count $\lfloor \beta \div \alpha \rfloor$. To be equivalent, remaining C_R is required to be located outside the merged C_L with a loop count $\beta\%\alpha$. Thus, the adjustment procedure is cloning and merging.

$$\Sigma_{new} = \begin{cases} \alpha(C_L(\beta \div \alpha)C_R)(\beta\%\alpha)C_R) & if\ \alpha < \beta \\ (\alpha\%\beta)C_L\beta((\alpha \div \beta)C_LC_R) & otherwise \end{cases} \quad (10)$$

To find a schedule adjustment point, we compute the 'Gain' and 'Cost' of an adjustment at each chain and select a chain which has the largest difference between the 'Gain' and 'Cost'. We perform schedule adjustment iteratively until we can not find a chain where the 'Gain' is larger than the 'Cost'.

For each chain, the new $|BTLC|$ after adjustment and its 'Gain' is computed as following, where $|Cluster|$ is the $|W|$ value of the common ancestor node in the old BTLC.

$$Gain = |BTLC|_{old} - |BTLC|_{new} \quad (11)$$

$$= \frac{\lfloor \beta \div \alpha \rfloor + (\beta\%\alpha)}{\beta} \times |Cluster| \quad (12)$$

The 'Cost' for adjustment is due to cloning a cluster, which is computed by following algorithm.

```
for(N ∈ ClonedCluster){
   if(N ∈ IN) {
      if(Cost2FN(N) ≤ BlockSize(N)) {
         Cost += Cost2FN(N);
         Move2FN(N);
      } else Cost += BlockSize(N);
   } else Cost += Cost4Call(N); /* N ∈ FN,FS */
   Cost += LoopOverhead;
}
```

Since we generate the code in a hybrid style, which is a mixture of inlines, functions, and shared functions, we define three sets of blocks during adjustment procedure: **IN**,**FN**,and **FS**. If a node in C_R is a member of **IN**, we compare the cost of moving that node to **FN** with its kernel code size to decide whether we maintain the node in **IN** or move the node to **FN**. The kernel size or the moving cost is added to the 'Cost' based on the decision. The moving cost from **IN** to **FN** includes the function body overhead, the function call overhead, and the variable migration overhead. The function body overhead and the function call overhead are small; they are 12 and 8 bytes in Sparc/Solaris environment. The variable migration overhead is incurred by changing the local variables into the global variables. Since a function code can not see the local variables of other

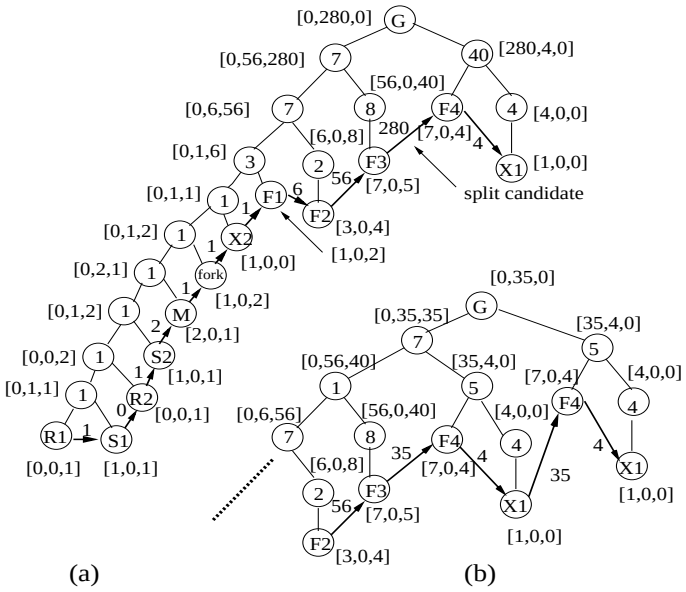

Figure 6. CD2DAT: BTLC data structure and schedule adjustment

functions, the local variables to store tokens among inline blocks should be replaced with global variables to store tokens among functions. The variable migration also increase the size of code to access global variables compared to accessing local variables. If a node is already in **FN** or **FS**, the additional cost is only one more function call. After the 'Cost' is computed for all the node in the cloned cluster, since the loop structure is also cloned, adding the loop cluster size finalize the cost computation. When the number of leaf node is N_L, the complexity to compute the 'Cost' is $O(N_L)$ and finding an adjustment point requires $O(N_L^2)$ time complexity.

Figure 6(a) is the BTLC of the cd2dat example in figure 3. The chain $\overline{(F3, F4)}$ which has 280 units of data buffer requirement is chosen. The adjusted BTLC is depicted in figure 6(b) whose buffer requirement is reduced to 143 from 351; the gain is 208.

5 Experimental Results

We have implemented the proposed scheme in our developing PeaCE environment, which is a Ptolemy extension as Codesign Environment. Two real life examples are chosen to show effectiveness of our approach; they are 8 channel filter bank and compact disk to digital audio tape converter, which are borrowed from the Ptolemy distribution.

Table 1 shows the results of the stepwise optimization. The cd2dat example shows significant code size reduction from the code sharing optimization and data size reduction

from schedule adjustment. The filter bank example containing 28 fir filters is an ideal example for code sharing optimization, which is confirmed by experiments. Since the sample rate change is not drastic, the filter bank example does not get any benefit from schedule adjustment.

Table 2 depicts the memory behavior of CD2DAT on ARM7 processor. To get the number in table 2, we use ARMulator[11] and dineroIII[12] to get a trace and the cache performance for the trace, respectively. The total fetch counts are increased, which results in longer execution time. However, the increments are not larger compared to memory buffer reduction; the increments are at most below 2.7%. Moreover, since the cache miss counts are reduced, the penalty may be reduced further. We currently analyze the memory access patterns to understand the experimental behavior rigorously.

Table 1. Change of program sizes after each optimization steps.

	CD2DAT	Filter bank
SAS	13672	28512
Code Sharing	12768	22024
Schedule Adjustment	12296	22024

Table 2. Memory behavior of CD2DAT in ARM7

	Fetches	Miss
SAS	17098177	57189
Code Sharing	17573923	52867
Schedule Adjustment	17499386	54331

6 Conclusions

In this paper, we have presented a pair of optimization techniques to jointly minimize the code and data memory requirements. As a starting point, a single appearance schedule is chosen because the schedule produces the minimum code size. Before applying the proposed optimization techniques, we carefully analyze the gains and overheads. Selective application of the optimization techniques shows significant improvements in memory requirement for both code and data.

Beyond what we achieved in this work, there are more chances of optimization to be studied in the future. As the kernel size becomes larger, the chance of code sharing optimization increases. So, we will develop a scheme to cluster fine grain nodes into a large grain before code sharing. Also, data memory requirements and cache behavior will be improved if buffer sharing is considered, which is another future work.

References

[1] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschimitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", Int. Journal of Computer Simulation, special issue on "Simulation Software Development",vol.4, pp. 155-182, April, 1994.

[2] E. A. Lee and D. G. Messerschimitt, "Synchronous Data Flow", Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, 1987

[3] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "GRAPE: A CASE Tool for Digital Signal Parallel Processing", IEEE ASSP Magazine, vol.7, (no.2):32-43, April, 1990

[4] S. Ritz, S. Pankert, H. Meyr, "High Level Software Synthesis for Signal Processing Systems", Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Albuquerque, pp.965-968, vol.2, April, 1990

[5] Shuvra S. Bhattacharyya and Edward A. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping", Journal of VLSI Signal Processing, 1992

[6] Praveen K. Murthy, Shuvra S. Bhattacharyya, and Edward A. Lee, " Joint Minimization of Code and Data for Synchronous Dataflow Programs", Journal of Formal Methods in System Design, vol. 11, no. 1, July, 1997

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC:Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations", DAES, Vol. 2, No. 1, pp. 33-60, January, 1997

[8] Marleen Ade, Rudy Lauwereins, J.A. Peperstraete, "Implementing DSP Applications on Heterogeneous Targets Using Minimal Size Data Buffers", Proceedings of RSP'96, pp. 166-172, June, 1996

[9] R. Govindarajan, G.R. Gao, and P. Desai, "Minimizing Memory Requirements in Rate-Optimal Schedules", Proceedings of the International Conference on Application Specific Array Processors, pp. 75-86, August, 1993

[10] S. Ritz, M. Willems, H. Meyr,"Scheduling for Optimum Data Memory Compaction in Block Diagram Oriented Software Synthesis",Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, p.2651-2653, 1995

[11] "ARM Software Development Toolkit Verion 2.11", available via "http://www.arm.com"

[12] "Wisconsin Architectural Research Tool Set", available via "http://www.cs.wisc.edu/~larus/warts.html"