Memory Efficient Software Synthesis with Mixed Coding Style from Dataflow Graphs

Wonyong Sung and Soonhoi Ha

Abstract—**This paper presents a set of techniques to reduce the code and data sizes for software synthesis from graphical digital signal-processing programs based on the synchronous dataflow model. By sharing the kernel code among multiple instances of a block with a shared function, we can further reduce the code size below the previous results based on inline coding style. A systematic approach also is devised to give up the single appearance schedule for reducing the data buffer requirement. The proposed techniques have been evaluated with two real-life examples to prove their significance.**

Index Terms—**Code sharing, memory requirement, schedule adjustment, software synthesis.**

I. Introduction

Minimizing the memory requirement is very important to synthesize code for embedded systems, specially for an on-chip design. Critical constraints on the memory size have made assembly programming by hand still a popular way of software development for embedded systems in spite of low productivity. Growing complexity of embedded systems, fast design turnaround time, limited development budget, and short lifecycle 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 sizes for software synthesis from graphical digital signal-processing (DSP) programs based on the synchronous dataflow (SDF) [1] model, which can be used as a block diagram specification model. In an SDF graph, 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.

Fig. 1(a) is an example SDF graph, and each arc is annotated with the number of tokens produced or consumed by an activation of its source or destination node. Each arc is assigned to a data buffer whose size can be constrained to the maximum number of tokens accumulated during a chosen execution of the graph. These data buffers compose the state of the SDF graph. Numerous DSP design environments, including a number of commercial tools, support SDF or closely related models ([2]–[4]) for both simulation and code generation.

A key property of the SDF model is that static schedules can be constructed at compile time, thus removing the run-time overhead of dynamic scheduling. Fig. 1(b) shows three valid schedules. Among valid schedules for an SDF graph, if every block appears exactly once in the schedule such as $\Sigma1$ and $\Sigma2$ in Fig. 1, the schedule is called a single appearance(SA) schedule. An SA-schedule that has no nested loop is called a flat SA-schedule. $\Sigma1$ of Fig. 1 is a flat SA-schedule, while $\Sigma2$ is not.

Software synthesis from an SDF graph includes determining a feasible schedule and a coding style for each dataflow node, both of

Manuscript received August 31, 1999. This work was supported by the academic fund of Ministry of Education, Republic of Korea, through the Inter-University Semiconductor Research Center, Seoul National University, under ISRC-98-E-2103.

W. Sung is with the CAP Laboratory in Seoul National University, Korea (e-mail: yong@iris.snu.ac.kr).

S. Ha is with the Department of Computer Engineering, Seoul National University, Korea (e-mail: sha@iris.snu.ac.kr).

Publisher Item Identifier S 1063-8210(00)09510-X.

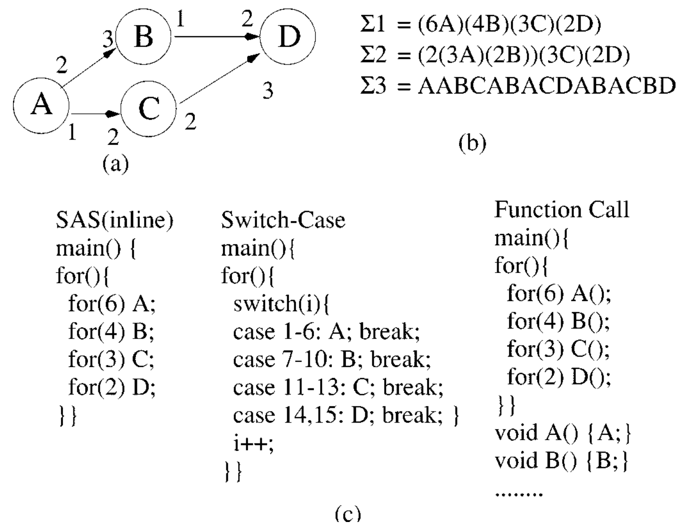

Fig. 1. (a) A sample SDF graph, (b) three possible schedules, and (c) three coding styles

which affect the memory requirements of the generated software for code and data. One of the main scheduling objectives for software synthesis is to minimize the total (sum of code and data) 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 function. The former generates an inline code for each node at the scheduled position, while the latter calls a function that contains the kernel. After a schedule is performed, the code for each dataflow node is generated depending on the coding style of the node. Fig. 1(c) shows three programs based on the same schedule $\Sigma1$ with various coding styles: inline, switch, and function. Among multiple valid schedules for an SDF graph, there exist tradeoffs between code memory size and data buffer memory requirement, depending on the coding style. Since a single appearance schedule guarantees the minimum code size for inline code generation, a group of researchers focused on finding a single appearance schedule that minimizes the buffer memory requirement [5]. Bhattacharyya *et al.*developed an algorithm to find out a looped SA-schedule with minimum buffer requirement among multiple looped SA-schedules, ignoring buffer sharing possibilities. Ritz *et al.* used an integer linear programming formulation to minimize the data memory [4]. Both works ([5], [4]) stick to single appearance schedules and do not exploit code-sharing optimization, in which multiple nodes share the same kernel in the generated code. There is a recent work on mixing coding styles for memory-efficient code synthesis [6]. However, they also did not consider the possibility of code sharing.

Most previous approaches first assume a coding style for all dataflow nodes and search for 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 that we will demonstrate. In this paper, we propose a pair of optimization techniques to overcome their limitations by mixing the coding style of different dataflow nodes. 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

1063–8210/00$10.00 © 2000 IEEE

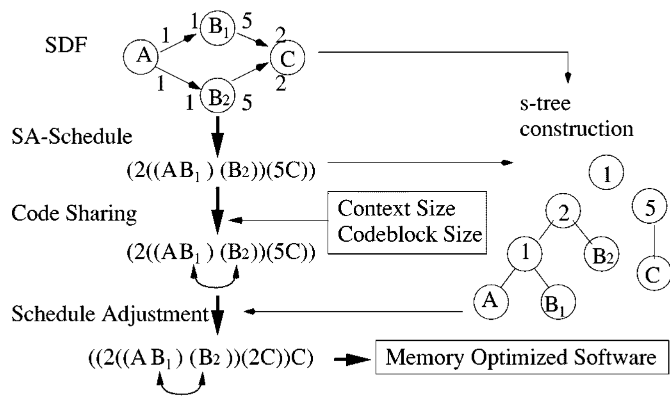

Fig. 2. Optimization steps in software synthesis.

these two techniques, we could reduce the memory requirements of two important examples by 13.6% and 8.1% over the best results from SA-schedule [5] at little expense of increased run-time overhead.

II. Proposed Strategy

Our strategy starts with a single appearance schedule obtained by the method described in [5]. Then, we apply our optimization techniques, code-sharing optimization and schedule adjustment, in sequence, as illustrated with an example in Fig. 2. In code-sharing optimization, we examine the graph to find multiple instances of a same block. Suppose that B1 and B2 represent the same block. Multiple instances are treated as different nodes in a single appearance schedule and appeared separately in the generated inlined code. The code-sharing technique described in the next section marks the two nodes B1 and B2 as shared. In the next phase, we give up SA-schedule to further reduce the buffer size if the gain is greater than the overhead. To manipulate the SA-schedule more efficiently, we express the schedule with a schedule tree, shortly, s-tree, which will be explained in Section III. From the s-tree, we can identify the possible locations of schedule adjustment and obtain the adjusted schedule. The resulting schedule is $(2(AB_1B_22C))C$ and node C is implemented as a function to avoid code duplication.

III. Code-Sharing Optimization

Fig. 3(a) shows the structure of the CD2DAT example, which is a sample rate converter from compact disc to digital audio tape. Since there are four instances of a finite impulse response (FIR) filter, it becomes a candidate of code sharing. Each FIR filter has its own state values such as tap values. Also, each input or output port of an instance is bound to its own buffer. In case the kernel code is shared among multiple instances, we should maintain, for each instance, separate state variables and buffers, which define the "context" of each instance. An example of the context of a FIR filter is depicted in Fig. 3(b). To decide whether a code block had better be shared or not, we compute the overhead and the gain of sharing. When $\Delta(\alpha)$ is an overhead, Ω is a code block size, and α is the number of instances of a block, code sharing is accepted if the inequality $1 > (\Delta(\alpha)/(\alpha - 1)\Omega)$ is satisfied.

The overhead incurred by code sharing comes from additional "context" data structure. We compute the sharing overhead $\Delta(\alpha)$ by dividing it into three parts: a context size overhead in the data block, the reference overhead, and the function call overhead in the code memory $(\Delta(\alpha) = \alpha(\Delta_{\text{context}} + \Delta_{\text{call}}) + \Delta_{\text{reference}})$.

In an implementation point of view, a context includes state variables as well as pointers to input and output token buffers. Since the state variables are also needed for each instance in the inlined code, the context size overhead comes from only pointer variables for input and output ports, which we call per-port overhead. For multirate computation, a port is usually implemented with a buffer array or a circular queue. Thus, at most three integer variables and a pointer variable are needed per port. To point the next read or write location in the array, an offset is needed. Since the offset is required also in the inline style, the offset is not counted as overhead. Two more integers are needed to delimit the end of the array and to describe the offset increment after each activation of the node, which are compiled as constants in an inline code. Therefore, the per-port overhead λ and the total context size overhead of a block are computed using $\Delta_{\text{context}} = \alpha \times \lambda \times (\#\text{ of ports})$, where $\lambda = 2 \times$ size_of(int) + size_of(pointer).

A reference overhead is an overhead resulting from accessing a port or a state through the context structure. When we access a variable through the context structure, we usually need additional codes. 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 on whether it is a port or a state. We consider three variable types: scalar type (such as integer or double), array, and constant. We define the overhead cost δ as a function of the reference type. In a SPARC/Solaris environment, the values of δ are 36, 32, 60, and 128, respectively, for scalar variable, constant, array of state, and array of port. By counting the references in a code block, we compute the reference overhead $\Delta_{\text{reference}}$, where $\varepsilon(t)$ is the reference count of type t in the kernel $(\Delta_{\text{reference}} = \Sigma_{t \in S,\, C,\, AS,\, AP}\, \varepsilon(t) \times \delta(t))$.

The constants we use in equations are highly machine dependent. The sizes of types and the addressing modes of a processor are two determinants. Since we can obtain the constant values easily from manuals or simple test programs, our technique is applicable to other than the SPARC/Solaris environment. For ARM7 processor, $\delta(t)$ is 16, 16, 24, and 40. The function call overhead is constant for a given processor: e.g., 12 for SPARC and 8 for ARM7. In summary, we compute the code-sharing overhead of a block using the port count and the reference counts, which can be obtained from the kernel of the block.

IV. Adjusting Single Appearance Schedule

Consider an example of Fig. 4. $\Sigma 1$ is the optimal SA-schedule in terms of buffer memory requirement. However, we can further reduce the buffer memory requirement by altering the schedule to $\Sigma 2$. $\Sigma 2$ is not an SA-schedule any more, for node C appears twice. Though the data buffer memory requirement is decreased, the code size is increased due to the duplication of node C. The major concern of this section is how to measure the gain and the cost of such schedule adjustment quantitatively. This section formalizes the schedule adjustment by introducing a binary tree representation of an SDF schedule (called schedule tree or s-tree).

Definition IV.1: An *s-tree representation* is recursively defined as a looped schedule $\gamma(S_l,\, S_r)$, where S_l and S_r are s-tree representations. A single dataflow node firing *N* is considered as the basic s-tree representation, 1(N, NULL), where NULL is an empty s-tree representation.

Any SDF schedule can be translated into an s-tree representation by inserting parenthesis at the proper places. For example, an s-tree representation of A(2B)C is ((A(2B))C) or (A((2B)C)). Scanning the list from left to right, we create a node when we meet a loop count or a node and create child nodes when we meet a left parenthesis. We move up to the parent node when we meet a right parenthesis. In the constructed binary tree, each leaf node corresponds to an SDF block, and an intermediate node represents a cluster of node invocations that are included in the s-tree subschedule with loop count γ. Since we start with an SA-schedule, there is only one invocation of any SDF block in the tree. We assign the buffer requirement information to each s-tree node as a tuple $[\mathbf{I},\, \mathbf{W},\, \mathbf{O}]$. It indicates the set of input buffers, the buffers between two child clusters, and the set of output buffers for the corresponding cluster to be scheduled. The $\mathbf{I}$ and $\mathbf{O}$ of a leaf node

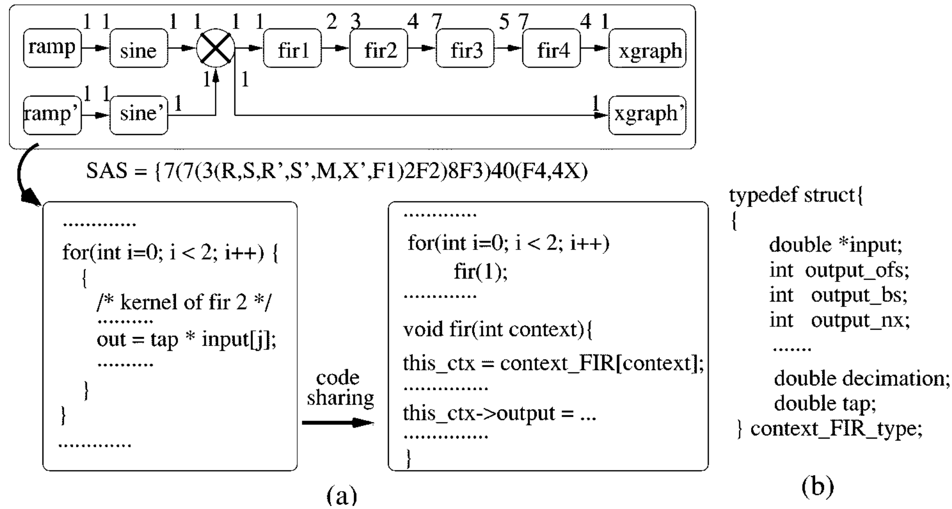

Fig. 3. CD2DAT example: from inline single appearance schedule to code-shared function code.

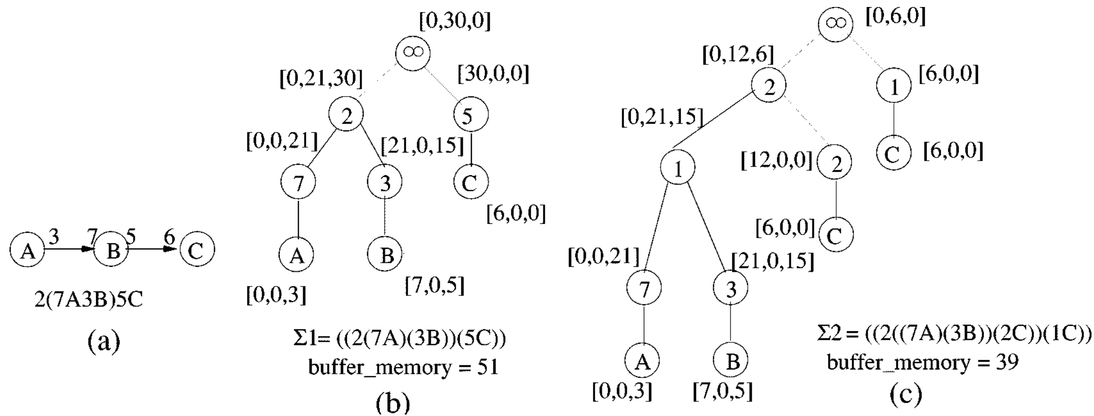

Fig. 4. (a) An example SDF graph, (b) schedule tree (s-tree) data structure for the graph, and (c) s-tree after schedule adjustment.

become the set of input and output buffers of the corresponding SDF block, and the $\mathbf{W}$ becomes null since there is no child node. In Fig. 4, we specify the maximum number of tokens in each set of tuple notation for brevity.

Suppose the clusters associated with an intermediate node and its two child nodes are $\mathbf{X}$, $\mathbf{L}$, and $\mathbf{R}$, as shown in Fig. 5. Some output ports of the left cluster $\mathbf{L}$ are connected to some inputs of the right cluster $\mathbf{R}$ inside the parent cluster $\mathbf{X}$. Those connections between two child clusters define the $\mathbf{W}$ set of the intermediate node; $\boldsymbol{W_X} = \boldsymbol{O_L} \cap \boldsymbol{I_R}$. As obviously shown in Fig. 5, each invocation of the cluster $\boldsymbol{X}$ requires $\boldsymbol{I_L} \cup \boldsymbol{I_R} - \boldsymbol{W_X}$ input buffers and $\boldsymbol{O_L} \cup \boldsymbol{O_R} - \boldsymbol{W_X}$ output buffers. In addition, the sizes of input buffers and output buffers should be multiplied by the loop count γ for the intermediate node.

We can compute the tuple information of all nodes in a bottom-up fashion starting from the leaf nodes.

Theorem IV.1: Summing up the size of the $\mathbf{W}$ set of all nodes produces the total buffer requirement needed by the schedule that the s-tree represents.

Proof: 1) An intermediate node of s-tree represents a cluster of SDF graph as shown in Fig. 5(a) and (b). In addition, the size of the $\mathbf{W}$ set of an intermediate node represents the buffer size for intercluster arcs between two child clusters. 2) Since all the arcs inside the cluster are encapsulated and invisible from outside, an arc belongs to only one cluster. 3) If an arc does not belong to any cluster, it lies between two separate clusters. Since the root represents the single top-level cluster, no arc is omitted for buffer computation. From 1), 2), and 3), the proof completes. ■

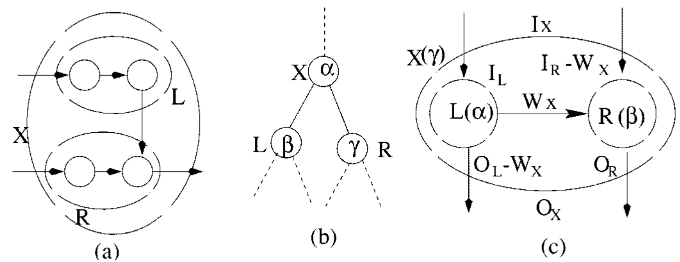

Fig. 5. (a) Clusters in an SDF graph, (b) s-tree representation, and (c) port and buffer information of three clusters.

For example, when we make an s-tree from the schedule ((2(7A3B))(5C)), the size of the $\mathbf{W}$ set of the root node is the total buffer size of arcs between two clusters (2(7A3B)) and (5C). For an intermediate node $\mathbf{G}$, we define $|\mathbf{G}|$ as the sum of the sizes of the $\mathbf{W}$ set of $\mathbf{G}$ and all nodes below $\mathbf{G}$, which is equal to the total buffer size of all arcs inside the cluster. If $\mathbf{G}$ is the root node, $|\mathbf{G}|$ is also called $|s - tree|$.

Fig. 4(b) shows the s-tree and its schedule for the SDF graph in Fig. 4(a), and its |s-tree| is 51. At the first step of schedule adjustment, an intermediate node is selected as an adjustment point by comparing the gain and the cost. Then, the schedule is adjusted by manipulating

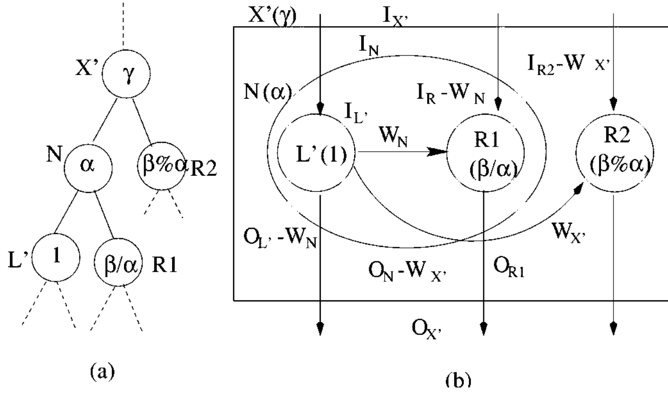

Fig. 6. The cluster structure after schedule adjustment: (a) part of new s-tree and (b) definition of new [I, W, O] tuples for clusters.

the s-tree. These steps are repeated until there is no intermediate node where the gain is larger than the cost. In the s-tree of Fig. 4(b), the buffer requirement between node B and C is 30. If we give up SA-schedule and construct a new schedule as (2(7A3B2C))(C), the buffer requirement between B and C is reduced to 18. Only when the buffer reduction is larger than the code size increment is the schedule adjustment regarded as worthwhile.

Fact IV.1: When a schedule Σ has two clusters L and R, that is, $\Sigma = \alpha(L)\beta(R)$, where α and β are loop counts of clusters L and R, the schedule can be adjusted as follows:

$$\begin{aligned} \Sigma_{\text{new}} &= \alpha(L(\beta \div \alpha)R)(\beta\%\alpha)R, \quad \text{if } \alpha < \beta \\ \Sigma_{\text{new}} &= (\alpha\%\beta)L\beta((\alpha \div \beta)LR), \quad \text{otherwise.} \end{aligned} \tag{1}$$

If we select an adjustment point, there are two subclusters L and R. Assume that the root node is chosen as an adjustment point in Fig. 4(b). The left loop cluster (L) is {2, 7, 3, A, B} with loop count 2 ($\gamma = 2$), and the right one (R) is {5, C} with loop count 5 ($\gamma = 5$). To reduce the buffer requirement between L and R, we merge R into L. The merged portion of R has a new loop count ($\beta \div \alpha$). The remainder of R is located outside the merged L with a loop count ($\beta\%\alpha$). For a cluster structure of Fig. 5(c) before schedule adjustment, Fig. 6 shows the cluster structure after schedule adjustment (cloning and merging). We can perform the schedule adjustment procedure with the s-tree data structure by showing that all the tuples of nodes after adjustment can be derived from those before adjustment. [7]

To compute the gain of schedule adjustment, we compute the change of buffer requirement. We denote $W(X)$ as the size of W set for node X. The buffer requirement within cluster X before the schedule adjustment is $(W(X)+|L|+|R|)$. After the schedule adjustment, the buffer requirement within X becomes $(W(N)+W(X')+|L'|+|R|)$. Since $R1$ and $R2$ are actually the same cluster, we should count the buffer size inside the cluster only once. Thus, $|R|$ is used instead of $|R1| + |R2|$. For $|L'|$ equal to $|L|$, we obtain the buffer size reduction, which is the gain of schedule adjustment, by $W(X) - (W(N) + W(X'))$. The gain becomes $W(X)(1 - (((\beta \div \alpha) + (\beta\%\alpha)) \div \beta))$, which is summarized as the following theorem.

Theorem IV.2: The gain of schedule adjustment is defined as the difference between the old and new $|s - tree|$. For each intermediate node (X), the gain of schedule adjustment at the node is computed as follows, assuming that R is cloned and merged into L:

$$\begin{aligned} \text{Gain } &= |s - tree|_{\text{old}} - |s - tree|_{\text{new}} \\ &= |W(X)|(1 - (((\beta \div \alpha) + (\beta\%\alpha))/\beta)). \end{aligned} \tag{2}$$

Since we generate the code in a hybrid style, which is a mixture of inlines, functions, and shared functions, we define three sets of

TABLE I
MEMORY REQUIREMENTS FOR CD2DAT EXAMPLE. (UNIT : BYTE)

Step	Code	Data	0-Init	Total
SAS	7560	1436	0(6904)	15900
Sharing	6428	1532	6904	14864
Adjust	6488	1544	5704	13736

TABLE II
MEMORY REQUIREMENTS FOR FILTER BANK EXAMPLE. (UNIT : BYTE)

Step	Code	Data	0-Init	Total
SAS	22204	1244	0(47008)	70456
Sharing	16380	1560	47008	64948
Adjust	16440	1552	46768	64760

blocks during optimization procedure: IN (inline), FN (function), and FS (function shared). The key difference between a function and a shared function is the way of accessing variables (port or state). A shared function accesses them only through the context described in Section III, while a function accesses them directly. The FS set is defined during the code-sharing optimization phase and not changed during the schedule adjustment phase. During the schedule adjustment phase, nodes in IN can be moved into FN. The following algorithm shows the detailed procedure:

```
for (each node n ∈ ClonedCluster) {
  if (n is an intermediate node &
  loop_count > 1 )
    Cost += LoopOverhead;
  else if(n ∈ IN {
    if (Cost2FN(n) ≤ instanceCount *
  BlockSize(n)) {
      Cost += Cost2FN(n);
      Move2FN(n);
    } else Cost += BlockSize(n);
  } else Cost + = Cost4Call(n); /* n ∈ FN,
  FS */
}
```

When a cluster N is cloned, we investigate all nodes inside. If a leaf node in N is a member of IN and the moving cost of the node from IN to FN, "Cost2FN(N)," is smaller than its kernel code size "BlockSize(N)," multiplied by the number of instances "instanceCount," the node will be moved into FN. The moving cost from IN to FN includes function body overhead, function call overhead, and variable migration overhead from local variables to global variables. If a node is already in FN or FS, the additional cost is only one more function call "Cost4Call(N)." Since the loop structure of a cluster is also cloned regardless of the coding style, "LoopOverhead" should be added to each cloned intermediate node.

When the number of leaf nodes 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. The final schedule obtained by the proposed optimization procedure becomes $7(7(3(R, S, R', S', M, X', F1)2F2), 8F3, 5(F4, X1))5(F4, X1)$.

V. EXPERIMENTAL RESULTS

Two real-life examples are chosen to show effectiveness of our approach: they are eight-channel filter bank and compact disc to digital audio tape converter, shown in Fig. 3, both of which are borrowed from

the Ptolemy distribution [2]. Full-fledged discussion of experimental results can be found in [7]. We summarize the final memory requirements at each optimization step.

Tables I and II show stepwise optimization of the memory requirement when it is executed on ARM7 processor. In both tables, SAS does not require any uninitialized (0-init) data. However, since variables are managed as automatic varaibles in stack segment in SAS method, the same amount of buffer is required in run-time memory. The CD2DAT example shows significant code-size reduction from 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. Compared with full inline implementation, the run-time overhead of the generated software is below 3%.

VI. Conclusion

In this paper, we have presented a pair of optimization techniques to jointly minimize the code and data memory requirement. 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 in an important class of applications.

Beyond what we achieved in this paper, there are more works to be studied in the future. At first our techniques should be extended to deal with the case where no SA-schedule exists. Considering the possibility of buffer sharing is another topic. Moreover, it may be better to devise a new scheduling policy to minimize the code and data memory considering the proposed optimization possiblities.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proce. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Sim.*, vol. 4, pp. 155–182, 1994.

[3] R. Lauwereins, M. Engels, J. A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren, "Grape: A case tool for digital signal parallel processing," *IEEE Acoust., Speech, Signal Processing Mag.* , vol. 7, no. 2, pp. 32–43, 1990.

[4] S. Ritz, M. Willems, and H. Meyr, "Scheduling for optimum data memory compaction in block diagram oriented software synthesis," in *Proc. Int. Conf. Acoustics, Speech, and Signal Processing*, 1995, pp. 2651–2653.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software implementations," *Design Automat. Embedded Syst.*, vol. 2, no. 1, pp. 33–60, Jan. 1990.

[6] J. Teich, E. Zitzler, and S. S. Bhattacharyya, "3d exploration of software schedules for dsp algorithms," in *Proc. CODES'99*, 1999.

[7] W. Sung, "Hardware software cosimulation using backplane approach from synchronous dataflowrepresentation," Ph.D. dissertation, Dept. of Computer Engineering, Seoul National Univ., Feb. 2000.

A Compositional Model for the Functional Verification of High-Level Synthesis Results

Dominique Borrione, Julia Dushina, and Laurence Pierre

***Abstract*—High-level synthesis systems, such as Amical, translate a behavioral description to an abstract automaton in which the states are decision and synchronization points, and operations are executed on the state transitions. After the scheduling and allocation of the functional units, the system is modeled as the interconnection of an operative and a control part. To formally verify this synthesis mechanism, we combine a detailed state encoding of the control part with an abstract view of the data part. We only compute the set of reachable states of the control part, and compose functional expressions in the data part. We show that, for each two corresponding state transitions in the abstract automaton and in the synthesized control part, the expressions computed in the data registers and outputs are equal.**

***Index Terms*—Formal verification, specification, state transition graphs.**

I. Introduction

In the context of the design of complex integrated circuits, the current challenge is the design-error free generation of large systems from behavioral specifications, allowing one to reuse previously designed parts. High-level synthesis (HLS) tools are now available, but their constant evolution and the concurrent change of the design libraries are so rapid that neither the programs nor the libraries can be considered provably correct. As a consequence, the results of HLS must undergo extensive verification before being fed to the logic design step [1]. Yet, due to the complexity of the circuits, and to their abstract specification, their exhaustive simulation is out of reach, and the current technology of automatic formal verification tools is no longer applicable.

Let us briefly recall the principles of today's verification tools. At the bit and word level, circuits have a fixed word length, fixed-width datapath, and a finite and known number of memory elements. Binary decision diagrams [2] and their enhancements [3], [4] efficiently represent the set of states reachable from the initial state by repetitively applying the "next state" transition relation. The functional correctness of the circuit needs only be established on the reachable states. "Model checkers" compute the truth of properties, expressed as temporal logic formulas, on the reachable states of a design description. These tools in reality perform either an explicit or a symbolic enumeration of the reachable states. As a consequence, their applicability is limited to circuits with fixed structure, and with a number of states that in practice cannot exceed 2^{100} (i.e., 100 bits of memory). These tools can no longer be applied *automatically* on designs with datapaths and on abstract specifications: the designer must proceed to manual abstractions (such as datapath width reduction) [5] and verification decomposition [6], the validity of which are not supported by automatic software. Other approaches, also requiring expert manual guidance, gave interesting results combining symbolic simulation and theorem proving [7], [8], but their wide use is hindered by the lack of automatic translation from conventional design languages to the input format of theorem provers.

Manuscript received August 4, 1999.

D. Borrione and J. Dushina are with TIMA/UJF, Grenoble Cedex 38031 France (e-mail: Dominique.Borrione@imag.fr).

L. Pierre is with LIM, Université de Provence, Marseille Cedex 13, 13453 France (e-mail: Laurence.Pierre@cmi.univ-mrs.fr).

Publisher Item Identifier S 1063-8210(00)09509-3.

1063–8210/00$10.00 © 2000 IEEE