Technical Report UMIACS-TR-2004-41, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2004.

Compact Procedural Implementation in DSP Software Synthesis through Recursive Graph Decomposition

Ming-Yung Ko[1], Praveen K. Murthy[2], and Shuvra S. Bhattacharyya[1]

[1] Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland at College Park, USA
{myko,ssb}@eng.umd.edu
[2] Fujitsu Research Laboratories
San Jose, California, USA
pmurthy@fla.fujitsu.com

Abstract. Synthesis of digital signal processing (DSP) software from dataflow-based formal models is an effective approach for tackling the complexity of modern DSP applications. In this paper, an efficient method is proposed for applying subroutine call instantiation of module functionality when synthesizing embedded software from a dataflow specification. The technique is based on a novel recursive decomposition of subgraphs in a cluster hierarchy that is optimized for low buffer size. Applying this technique, one can achieve significantly lower buffer sizes than what is available for minimum code size inlined schedules, which have been the emphasis of prior software synthesis work. Furthermore, it is guaranteed that the number of procedure calls in the synthesized program is polynomially bounded in the size of the input dataflow graph, even though the number of module invocations may increase exponentially. This recursive decomposition approach provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis. The experimental results demonstrate a significant improvement in buffer cost, especially for more irregular multi-rate DSP applications, with moderate code and execution time overhead.

1 Introduction and Related Work

Due to the growing complexity of DSP applications, the use of dataflow-based, block diagram programming environments is becoming increasingly popular for DSP system design. The advantages of such environments include intuitive appeal; promotion of useful software engineering practices such as modularity and code reuse; and improved quality of synthesized code through automatic code generation. Examples of commercial DSP design tools that incorporate dataflow semantics include System Canvas from Angeles Design Systems [12], SPW from Cadence Design Systems, ADS from Agilent, Cocentric System Studio from Synopsys [2], GEDAE from Lockheed, and the Autocoding Toolset from Management, Communications, and Control, Inc. [15]. Research-oriented tools and languages related to dataflow-based DSP design include Ptolemy from U. C. Berkeley [3], GRAPE from K. U. Leuven [8], Compaan from Leiden University [16], and StreamIt from MIT [5].

A significant body of theory and algorithms has been developed for synthesis of software from dataflow-based block diagram representations. Many of these techniques pertain to the *synchronous dataflow (SDF)* model [9], which can be viewed as an important common denominator across a wide variety of DSP design tools. The major advantage of SDF is the potential for static analysis and optimization. In [1], algorithms are developed to optimize buffer space while obeying the constraint of minimal code space. A multiple objective optimization is proposed in [18] to compute the full range of Pareto-optimal solutions in trading off code size, data space, and execution time. Vectorization can be incorporated into SDF graphs to reduce the rate of context switching and enhance execution performance [7, 14].

In this paper, an efficient method is proposed for applying subroutine call instantiation of module functionality to minimize buffering requirements when synthesizing embedded software from SDF specifications. The technique is based on a novel recursive decomposition of subgraphs in a cluster hierarchy that is optimized for low buffer size. Applying this technique, one can achieve significantly lower buffer sizes than what is available for minimum code size inlined schedules, which have been the emphasis of prior software synthesis work. Furthermore, it is guaranteed that the number of procedure calls in the synthesized program is polynomially bounded in the size of the input dataflow graph, thereby bounding the code size overhead. Having such a bound is particularly important because the number of module invocations may increase exponentially in an SDF graph. Our recursive decomposition approach provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis.

In [17], an alternative buffer minimization technique through transforming schedules is investigated. The transformation performs once division and modulo operations over loop counts within a selected sub-schedule. In contrast to managing loop counts in schedules, we target token exchange rates which is suitable in the context of graph theory. Since schedules can be related to SDF actor clusters, our approach, in a sense, is a more general work in that all clusters are traversed and division/modulo are recursively applied as far as applicable. Moreover, our graph decomposition strategy extends a two-actor theory which prooves a minimal buffer bound [1]. The recursion generates results better, or at least as good as, than that of [17] due to the fact that each division/modulo induces less buffer requirement. This paper also distinguishes from [17] where multiple coding styles, including procedures, are traded off in the software synthesis.

Buffer minimization and use of subroutine calls during code synthesis have also been explored in the *phased scheduling* technique [4]. This work is part of the StreamIt language [5] for developing streaming applications. Phased scheduling applies to a restricted subset of SDF graphs, in particular each basic computation unit (called a *filter* in StreamIt) allows only a single input and output. In contrast, the recursive graph decomposition approach applies to all SDF graphs that have *single appearance schedules* (this class includes all properly-constructed, acyclic SDF graphs), and furthermore, can be applied outside the tightly interdependent components of SDF graphs that do not have single appearance schedules. Tightly interdependent components are unique, maximal subgraphs that exhibit a certain form of data dependency [1]. Through extensive experiments with single appearance scheduling, it has been observed that tightly interdependent components arise only very infrequently in practice [1]. Integrating phased scheduling concepts with the decomposition approach presented in this paper is an interesting direction for further work.

Panda surveys data memory optimization techniques for compiling *high level languages (HLLs)*, such as C, including techniques such as code transformation, register allocation, and address generation [13]. Due to the instruction-level parallelism capability found in many DSP processors, the study of independent register transfers is also a useful subject. The work of [10] investigates an integer programming approach for code compaction that obeys exact timing constraints and saves code space as well. Since code for individual actors is often specified by HLLs, several such techniques are complementary to the techniques developed in this paper. In particular HLL compilation techniques can be used for performing intra-actor optimization in conjunction with the inter-actor, SDF-based optimizations developed in this paper.

2 Background and Notation

An SDF program specification is a directed graph where vertices represent functional blocks (*actors*) and edges represent data dependencies. Actors are activated when sufficient inputs are available, and FIFO queues (or *buffers*) are usually allocated to buffer data transferred between actors. In addition, for each edge e, the numbers of data values produced $prd(e)$ and consumed $cns(e)$ are fixed at compile time for each invocation of the source actor $src(e)$ and sink actor $snk(e)$, respectively.

A *schedule* is a sequence of actor executions (or *firings*). We compile an SDF graph by first constructing a *valid schedule*, a finite schedule that fires each actor at least once, and does not lead to unbounded buffer accumulation (if the schedule is repeated indefinitely) nor buffer underflow on any edge. To avoid buffer overflow and underflow problems, the total amount of data produced and consumed is required to be matched on all edges. In [9], efficient algorithms are presented to determine whether or not a valid schedule exists for an SDF graph, and to determine the minimum number of firings of each actor in a valid schedule. We denote the *repetitions* of an actor as this minimum number of firings and collect the repetitions for all actors in the *repetitions vector*. Therefore, given an edge e and repetitions vector q, the *balance equation* for e is written as $q(src(e))\,prd(e) = q(snk(e))\,cns(e)$.

To save code space, actor firings can be incorporated within loop constructs to form *looped schedules*. Looped schedules group sequential firings into schedule loops; each such loop is composed of a loop iteration count and one or more iterands. In addition to being firings, iterands also can be schedules, and therefore, it is possible to form nested looped schedules. The notation we use for a schedule loop L is $L = (nI_1I_2\ldots I_m)$, where n denotes the iteration count and $I_1, I_2, \ldots, I_m$ denote the iterands of L. *Single appearance schedules (SAS)* refer to schedules where each actor appears only once. In inlined code implementation, an SAS contains a single copy of code for every actor and results in minimal code space requirements. For an acyclic SDF graph, an SAS can easily be derived from a topological sorting of the actors. However, such an SAS often requires relatively high buffer cost. A more memory-efficient method of SAS construction is to perform a certain form of dynamic programming optimization (called *DPPO* for *dynamic programming post optimization*) over a topological sort to generate a buffer-efficient, nested looped schedule [1]. In this paper, we employ the *acyclic pairwise grouping for adjacent nodes (APGAN)* algorithm [1] for the generation of topological sorts and the DPPO method described above for the optimization of these topological sorts into more buffer-efficient form.

3 Recursive Decomposition of a Two-Actor SDF Graph

Given a two-actor SDF graph as shown on the left in Figure 1, we can recursively generate a schedule that has a buffer memory requirement of the least amount possible. The scheduling technique works in the following way: given the edge AB, and $prd(AB) = n > cns(AB) = m$, we derive the new graph shown on the right in Figure 1 where the actor set is $\{A_1, B_1\}$ and $prd(A_1B_1) = n \bmod m$. The actor A_1 is a hierarchical actor that represents the schedule $A\ (\lfloor n/m \rfloor B)$, and B_1 just represents B. Consider a minimum buffer schedule for the reduced graph, where we replace occurrences of A_1 a by $A\ (\lfloor n/m \rfloor B)$, and occurrences of B_1 are replaced by B. For example, suppose that $n = 3$ and $m = 2$. Then 3 mod 2 = 1, the minimum buffer schedule for the reduced graph would be $A_1A_1B_1$, and this would result in the schedule $ABABB$ after the replacement. As can be verified, this later schedule is a valid schedule for the original graph, and is also a minimum buffer schedule for it, having a buffer memory requirement of $n + m - 1$ as expected.

However, the advantage of the reduced graph is depicted in Figure 2: the schedule for the reduced graph can be implemented using procedure calls in a way that is more parsimonious than simply replacing each occurrence of A and B in $ABABB$ by procedure calls. This latter approach would require 5 procedure calls, whereas the hierarchical implementation depicted in Figure 2 requires only 3 procedure calls. The topmost procedure implements the SAS $A_1A_1B_1 = (2A_1)B_1$, where A_1 is really a procedure call; this procedure call implements the SAS AB, which in turn call the actors A and

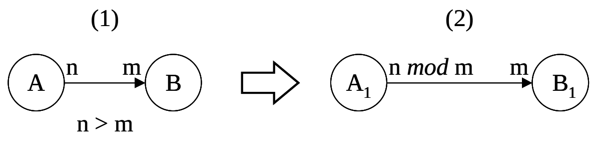


Figure 1 A two-actor SDF graph and its reduced version.

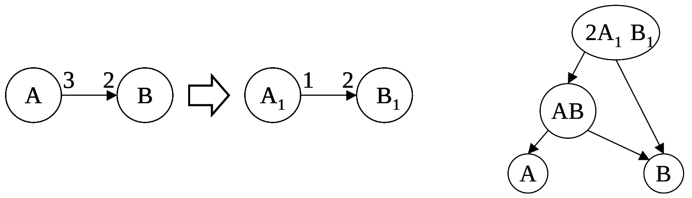


Figure 2 A hierarchical procedural implementation of a minimum buffer schedule for the SDF graph on the left.

B. Of-course, we could implement the schedule $ABABB$ more efficiently than simply using five procedure calls; for example, we could generate inline code for the schedule $(2AB)B$; this would have 3 blocks of code: two for B, and one for A. We would have to do a trade-off analysis to see whether the overhead of the 3 procedure calls would be less than the code-size increase of using 3 appearances (instead of 2).

We first state an important theorem from [1]:

Theorem 1: For the two-actor SDF graph depicted on the left in Figure 1, the minimum buffer requirement over all schedules is given by $n + m - gcd(n, m)$.

Proof: See [1].

We denote $n + m - gcd(n, m)$ for a two-actor SDF graph depicted on the left in Figure 1 as the *VBMLB (the buffer memory lower bound over all valid schedules)*. The definition of VBMLB also applies to an SDF edge. Similarly, for arbitrary SDF graphs, the VBMLB for a graph can be defined as the sum of VBMLBs over all edges.

Theorem 2 shows that the preservation of the minimum buffer schedule in the reduced graph in the above example is not a coincidence.

Theorem 2: The minimum buffer schedule for the reduced graph on the right in Figure 1 yields a minimum buffer schedule for the graph on the left when the appropriate substitutions of the actors are made.

Proof: Let $gcd(n, m) = g$. The equation $gcd(n \bmod m, m) = g$ must hold since a fundamental property of the gcd is that $gcd(n, m) = gcd(n \bmod m, m)$. So the minimum buffer requirement for the reduced graph is given by $n \bmod m + m - g$ from Theorem 1. Now, when A_1 is replaced by $A\ (\lfloor n/m \rfloor B)$ to get a schedule for the original graph, we see that the maximum number of tokens is going to be reached after a firing of A since firings of B consume tokens. Since the maximum number of tokens reached in the reduced graph on edge $A_1 B_1$ is $n \bmod m + m - g$, the maximum number reached on AB when we replace A_1 by $A\ (\lfloor n/m \rfloor B)$ will be

$$n \bmod m + m - g + \left\lfloor \frac{n}{m} \right\rfloor m = n + m - g .$$

Hence, the theorem is proved. **QED**.

Theorem 3: An SAS for a two-actor graph satisfies the VBMLB if and only if either $n \mid m$ (n is dividable by m) or $m \mid n$. A 2-actor SDF graph where either $n \mid m$ or $m \mid n$ is called a *perfect SDF graph (PSG)* in this paper.

Proof: (Forward direction) Assume WLOG that $n > m$. Then the SAS is going to be $(m/(gcd(n, m)))A)((n/(gcd(n, m)))B)$. The buffering requirement of this schedule is $mn/gcd(n, m)$. Since this satisfies the VBMLB, we have

$$\frac{m}{gcd(n, m)} n = m + n - gcd(n, m) . \qquad (1)$$

Since $n > m$, we have to show that (1) implies $m \mid n$. The contrapositive is that if $m \mid n$ does not hold then Equation 1 does not hold. Indeed, if $m \mid n$ does not hold, then $gcd(n, m) < m$, and $m/(gcd(n, m)) \geq 2$. In the R.H.S. of (1), we have

$m - gcd(n, m) < m < n$, meaning that the R.H.S. is $< 2n$. This shows that (1) cannot hold.

The reverse direction follows easily since if $m \mid n$, then the L.H.S. is n, and the R.H.S. is $m + n - m = n$. **QED**.

Theorem 4: A minimum buffer schedule for a two-actor SDF graph can be generated in the recursive hierarchical manner by reducing the graph until either $n \mid m$ or $m \mid n$.

Proof: This follows by Theorems 2 and 3 since reduction until $n \mid m$ or $m \mid n$ is necessary for the terminal schedule to be an SAS by Theorem 3, and the back substitution process preserves the VBMLB by Theorem 2.

Theorem 5: The number of reductions needed to reduce a two-actor SDF graph to a PSG is polynomial in the size of the SDF graph and is bounded by $O(\log n + \log m)$.

Proof: This follows by Lame's theorem for showing that the Euclidean GCD algorithm runs in polynomial time. We repeat the proof here for completeness; it is taken from [6]. Suppose that $n > m > 0$, and there are $k \geq 1$ reductions to get the PSG. Then we show that $n > F_{k+2}$ and $m > F_{k+1}$, where F_k is the k^{th} Fibonacci number ($F_k = F_{k-1} + F_{k-2}, \forall k > 1, F_0 = 0, F_1 = 1$). This will imply that if $m \leq F_{k+1}$, then there are fewer than k reductions to get the PSG. Since

$$F_k \approx \phi^k = \left(\frac{(1+\sqrt{5})}{2}\right)^k ,$$

the number of reductions is $O(\log\ m)$.

The proof is by induction on k. For the basis, let $k = 1$. Then $m > 1 = F_2$ since needing one reduction implies that m cannot be 1. Since $n > m$, we must have $n > 2 = F_3$. Now assume that it is true that if $k - 1$ reductions are required then $n > F_{k+1}$ and $m > F_k$. We will show that it holds for k reductions also. Since $k > 0$, we have $m > 1$, and the reduction process will produce a reduced graph with $n' = n \bmod m$ and $m' = m$. We will then make $k - 1$ additional reductions (the next reduction will result in a graph with $n'' = n'$, and $m'' = m' \bmod n'$ and so on). The inductive hypothesis implies that $m > F_{k+1}$ (since $m > n \bmod m$ after the first reduction), proving one part of the requirement, and that $n \bmod m > F_k$. Now, $m + n \bmod m = m + (n - \lfloor n/m \rfloor m) \leq n$. Hence,

$$n \geq m + n \bmod m > F_{k+1} + F_k = F_{k+2} ,$$

as required.

We can also show that if $m = F_{k+1}$, then there are exactly $k - 1$ reductions. Indeed, if $n = F_4 = 3$ and $m = F_3 = 2$, $k + 1 = 3$ there is $k - 1 = 1$ reduction. For $k \geq 2$, we have $F_{k+1} \bmod F_k = F_{k-1}$. Thus reducing the graph with $n = F_{k+1}, m = F_k$ results in a graph with $n' = F_{k-1}, m' = F_k$, which shows inductively that there will be $k - 1$ reductions exactly. **QED**.

Thus, we can implement the minimum buffer schedule in a recursive, hierarchical manner, where the number of subroutine calls is guaranteed to be polynomially bounded in the size of the original two-actor SDF graph.

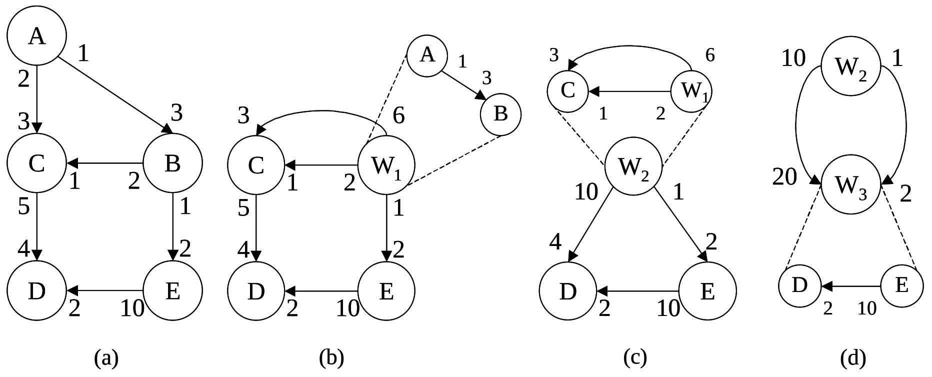
4 Extension to Arbitrary SAS

Any SAS S can be represented as an *R-schedule*,

$$S = (i_L S_L)(i_R S_R) \ ,$$

where S_L is the schedule for a "left" portion of the graph and S_R is the schedule for the corresponding "right" portion [1]. The schedules S_L, S_R can be recursively decomposed this way until we obtain schedules for two-actor graphs. In fact, the decomposition above can be represented as a clustered graph where the top level graph has two hierarchical actors and one or more edges between them. Each hierarchical actor in turn contains two-actor graphs with hierarchical actors until we reach two-actor graphs with non-hierarchical actors. Figure 3 shows an SDF graph, an SAS for it, and the resulting cluster hierarchy.

This suggests that the hierarchical implementation of the minimum buffer schedule can be applied naturally to an arbitrary SAS starting at the top-most level. In Figure 3, the graph in (d) is a PSG and has the SAS $(2W_2)W_3$. We then decompose the actors W_2 and W_3. For W_3, the graph is also a PSG, and has the schedule $E\ (5D)$. Similarly, the graph for W_2 is also a PSG with the schedule $W_1(2C)$. Finally, the graph for W_1 is also a PSG, and has the schedule $(3A)B$. Hence, in this example, no reductions are needed at any stage in the hierarchy at all, and the overall buffering requirement is $20 + 2 = 22$ for the graph in (d), 10 for W_3, 8 for W_2, and 3 for W_1, for a total requirement of 43. The VBMLB for this graph is 29. The reason that even the hierarchical decomposition does not yield the VBMLB is that the clustering process amplifies the produced/consumed parameters on edges, and inflates the VBMLB costs on those edges.



SAS: (2 ((3A) B) (2C)) (E (5D))

Figure 3 An SAS showing how an SDF graph can be decomposed into a series of two-actor subgraphs.

The extension to an arbitrary SDF graph, in other words, is to compute the VBMLB of the cluster hierarchy that underlies the given R-schedule. That is the goal the graph decomposition achieves and an algorithm overview is illustrated in Figure 4. The VBMLB of the cluster hierarchy is calculated through summation over the VBMLB of all edges at each hierarchical level (e.g., W_1, W_2, W_3, and the top-most level comprising W_2 and W_3 in Figure 3). We denote this cost as the *VBMLB for a graph cluster hierarchy* and for the example of Figure 3, the cluster hierarchy VBMLB is 43 as computed in the previous paragraph.

To obtain an R-schedule, DPPO is a useful algorithm to start with. As discussed in Section 2, DPPO is a dynamic programming approach to generating an SAS with minimal buffering cost. Because the original DPPO algorithm pertains to direct implementation in SAS form, the cost function in the dynamic programming approach is based on a buffering requirement calculation that assumes such implementation as an SAS. If, however, the SAS is to be processed using the decomposition techniques developed in Section 4, the VBMLB value for an edge e,

$$prd(e) + cns(e) - gcd(prd(e), cns(e)) \ ,$$

is a more appropriate cost criterion for the dynamic programming formulation. This modified DPPO approach will evaluate a VBMLB-optimized R-schedule, which provides a hierarchical clustering suitable for our recursive graph decomposition.

Notice that we had to deal with multiple edges between actors in the above example. It is not immediately obvious whether there exists a schedule for a two-actor graph with multiple edges between the two actors that will simultaneously yield the VBMLB on each edge individually. We prove several results below that guarantee that there does exist such a schedule, and that a schedule that yields the VBMLB on any one edge yields the VBMLB on all edges simultaneously.

Consider the consistent two-actor graph shown in Figure 5. The repetitions vector satisfies the following equations:

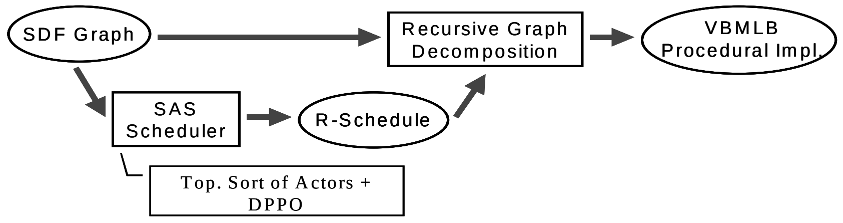


Figure 4 An algorithm overview for arbitrary SDF graphs.

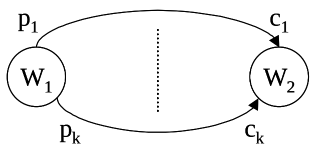


Figure 5 A two-actor SDF multi-graph.

$$q(W_1)p_i = q(W_2)c_i \qquad \forall i = 1, \ldots, k \ . \tag{2}$$

The fact that the graph is consistent means that (2) has a valid, non-zero solution.

Lemma 1: Suppose that $p_1 \le \ldots \le p_k$. Then $c_1 \le \ldots \le c_k$.

Proof: Suppose not. Suppose that for some i, j, we have $p_i \le p_j$ but $c_i > c_j$. Equation (2) implies that $c_i / p_i = c_j / p_j$. But $p_i \le p_j$ and $c_i > c_j$ implies $c_i / p_i > c_j / p_j$, contradicting (2). **QED**.

Now consider the two graphs shown in Figure 6. Let these graphs have the same repetitions vector. Thus, we have

$$q(A)p_1 = q(B)c_1 \text{ and } q(A)p_2 = q(B)c_2 \ . \tag{3}$$

Theorem 6: The two graphs in Figure 6 (I) and (II) have the same set of valid schedules.

Proof: Suppose not. Suppose there is a schedule for (I) that is not valid for (II). Let σ be the firing sequence $X_1 X_2 \ldots X_{q(A)+q(B)}$, where $X_i \in \{A, B\}$. Since σ is not valid for (II), there is some point at which a negative state would be reached in this firing sequence in graph (II). By a negative state, we mean a state in which at least one buffer has had more tokens consumed from it than the number of tokens that have been produced into it. That is, after n_A, n_B firings of A and B respectively, we have $n_A p_2 - n_B c_2 < 0$ while $n_A p_1 - n_B c_1 \ge 0$. So,

$$0 \le n_A p_1 - n_B c_1 < n_B \frac{c_2}{p_2} p_1 - n_B c_1 \ .$$

By (3), we have $c_1 / p_1 = c_2 / p_2$. Thus $n_B (c_2 / p_2) p_1 - n_B c_1 = 0$, giving a contradiction. **QED**.

Theorem 7: The schedule that yields the VBMLB for (I) also yields the same VBMLB for (II).

Proof: Let σ be the firing sequence $X_1 X_2 \ldots X_{q(A)+q(B)}$, where $X_i \in \{A, B\}$, that yields the VBMLB for (I). By Theorem 6, σ is valid for (II) also. Since σ is the VBMLB schedule for (I), at some point, after n_A^*, n_B^* firings of A and B respectively, we have

$$n_A^* p_1 - n_B^* c_1 = p_1 + c_1 - gcd(p_1, c_1) \tag{4}$$

and for all other n_A and n_B in σ,

$$n_A p_1 - n_B c_1 \le n_A^* p_1 - n_B^* c_1 \ . \tag{5}$$

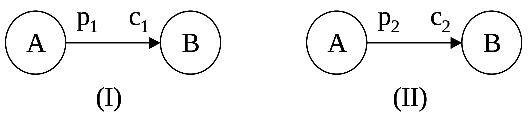


Figure 6 Two two-actor graphs with the same repetitions vector.

We have that

$$n_A^* p_2 - n_B^* c_2 = n_A^* p_2 - n_B^* \frac{p_2}{p_1} c_1$$

$$= p_2\left(n_A^* - \frac{c_1}{p_1} n_B^*\right)$$

$$= p_2\left(n_A^* + \frac{p_1 + c_1 - gcd(p_1, c_1) - n_A^* p_1}{p_1}\right)$$

$$= p_2\left(1 + \frac{c_1}{p_1} - \frac{gcd(p_1, c_1)}{p_1}\right)$$

$$= p_2 + c_2 - \frac{gcd(p_1, c_1)}{p_1} p_2$$

$$= p_2 + c_2 - gcd\left(p_1 \frac{p_2}{p_1}, \frac{c_1 p_2}{p_1}\right)$$

$$= p_2 + c_2 - gcd(p_2, c_2)$$

By (5), we have

$$n_A \le n_B \frac{c_1}{p_1} + 1 + \frac{c_1}{p_1} - \frac{gcd(p_1, c_1)}{p_1} \quad .$$

Thus,

$$n_A p_2 - n_B c_2 \le \left(n_B \frac{c_1}{p_1} + 1 + \frac{c_1}{p_1} - \frac{gcd(p_1, c_1)}{p_1}\right) p_2 - n_B c_2 = p_2 + c_2 - gcd(p_2, c_2) \quad .$$

Hence, this shows that σ yields the VBMLB for (II) also. **QED**.

Theorem 8: For the graph in Figure 5, there is a schedule that yields the VBMLB on every edge simultaneously.

Proof: Follows from the above results.

5 CD-DAT Example

Given the CD-DAT example in Figure 7, the DPPO algorithm returns the SAS $(7(7(3AB)(2C))(4D))(32E(5F))$. This schedule can be decomposed into two-actor clustered graphs as shown in Figure 8. The complete procedure call sequence is shown in Figure 9, where each vertex represents a subroutine, and the edges represent caller-

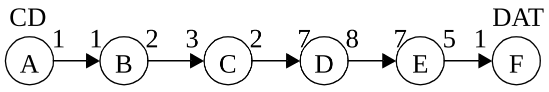


Figure 7 A CD-DAT sample rate converter example.

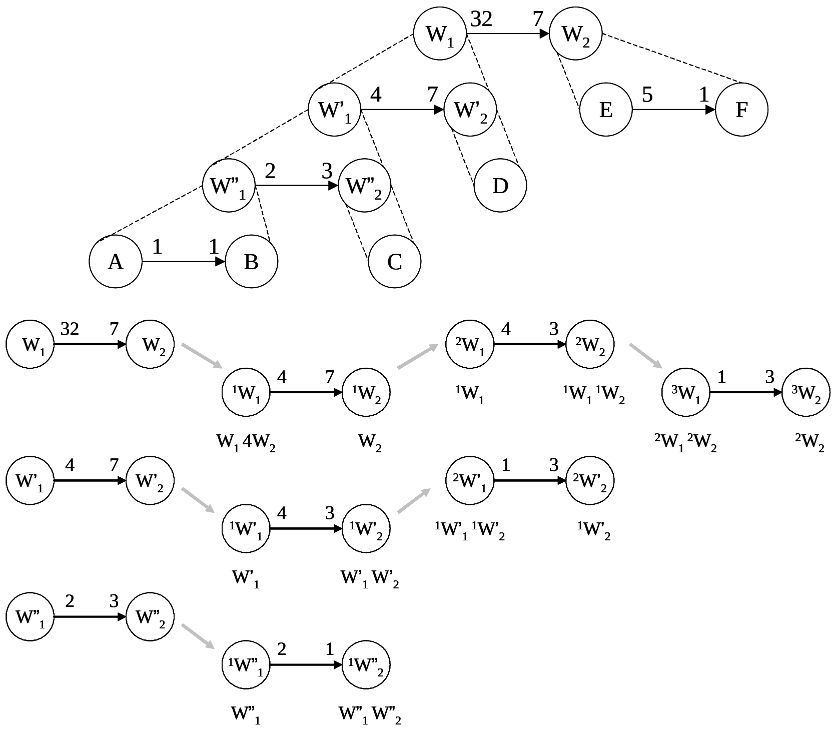


Figure 8 The recursive decomposition of the CD-DAT graph.

callee relationships. The generated C style code is shown as well in Figure 10. The total buffer memory requirement is:

$$(32 + 7 - 1) + (4 + 7 - 1) + (2 + 3 - 1) + 5 + 1 = 58$$

This is a 72% improvement over the best requirement of 205 obtained for a strictly inlined implementation of a SAS. The requirement of 205 is obtained by using a buffer merging technique [11].

6 Overall Running Time

Previously, we showed that the number of decompositions required to reach a PSG is polynomial in the size of the two-actor SDF graph. For arbitrary graphs, the clustering process expands the produced/consumed numbers; in fact, these numbers can increase multiplicatively. Is the decomposition process still polynomial in the size of the SDF graph? It is, as the following analysis shows. First off, we have that the repetitions

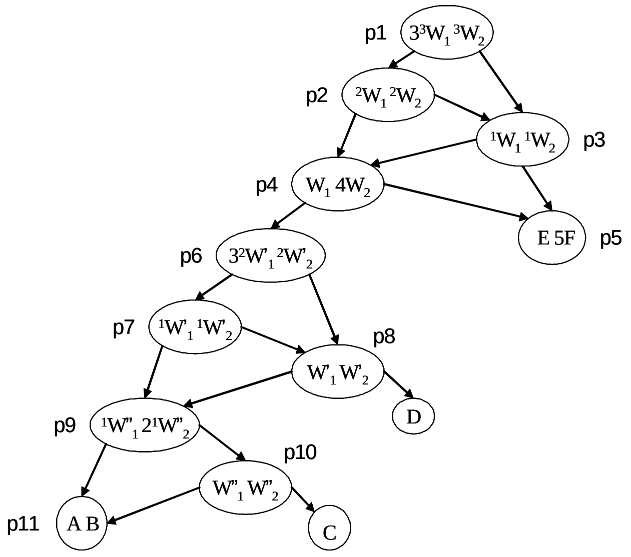


Figure 9 Procedure call sequence for the CD-DAT example.

```
p1() {
  for (int i=0; i<3; i++)
    p2();
  p3();
}

p2() {
  p4();
  p3();
}

p3() {
  p4();
  p5();
}

p4() {
  p6();
  for (int i=0; i<4; i++)
    p5();
}

p5() {
  inline of actor E;
  for (int i=0; i<5; i++)
    inline of actor F;
}
```

```
p6() {
  for (int i=0; i<3; i++)
    p7();
  p8();
}

p7() {
  p9();
  p8();
}

p8() {
  p9();
  inline of actor D;
}

p9() {
  p11();
  for (int i=0; i<2; i++)
    p10();
}

p10() {
  p11();
  inline of actor C;
}

p11() {
  inline of actor A;
  inline of actor B;
}
```

Figure 10 Generated C code for the CD-DAT example.

number for any actor v is $q(v) = O(P^{|E|})$, where $p = MAX_{e \in E}(prd(e), cns(e))$ and E is the set of edges in the SDF graph. If we cluster some set of actors $\{v_i, \ldots, v_j\}$ into a actor W, the produced parameter on each edge e_k leaving W [1] is increased by

$$\frac{q(v_k)}{gcd(q(v_i), \ldots q(v_j))} prd(e_k) \le q(v_k) prd(e_k) \,.$$

Since the number of decompositions was $O(\log(m))$, we see that if $m = q(v_k)prd(e_k)$, then

$$\log(m) = \log(q(v_k)) + \log(prd(e_k)) = |E| \log(P) + \log(prd(e_k)) ,$$

and this is a polynomial function of the SDF graph.

7 Experimental Results

Our optimization algorithm is particularly beneficial to a certain class of applications. The statement of Theorem 3 tells us that no reduction is needed for edges with production and consumption rates that are multiples of one another. We call such edges *uniform edges*. Precisely, if an edge e has production and consumption rates m and n, respectively, then e is uniform if either $m|n$ or $n|m$. Our proposed strategy can improve buffering cost for *non-uniform* edges and generate the same buffering cost as existing SAS techniques for uniform edges.

We define two metrics for measuring this form of uniformity for a given SDF graph $G = (V, E)$ and an associated R-schedule S. For this purpose, we denote E_c as the set of edges in the cluster hierarchy associated with S. Thus, $|E_c| = |E|$ since every $e \in E$ has a corresponding edge in one of the clustered two-actor subgraphs associated with S. The set E_c can be partitioned into two sets: the uniform edge set E_u, which consists of the uniform edges, and the non-uniform edge set E_{nu}, which consists of the remaining edges.

Metric 1: Uniformity based on edge count:

$$U_1(G, S) = \frac{|E_u|}{|E|} .$$

Metric 2: Uniformity based on buffer cost:

$$U_2(G, S) = \frac{\sum_{e \in E_u} b(e)}{\sum_{e \in E} b(e)} ,$$

where $b(e)$ is the buffer cost on edge e for the given graph and schedule.

Our procedural implementation technique produces no improvement in buffering cost when uniformity is 100% (note that 100% uniformity for Metric 1 is equivalent to 100% uniformity for Metric 2). This is because if uniformity is 100%, then the two-actor graphs in the cluster hierarchy do not require any decomposition to achieve their associated VBMLB values.

We examined several SDF applications that exhibit uniformity values below 100%, and the results are listed in Table 1. The first three columns give the benchmark names and graph sizes. Uniformity is measured by the proposed metrics and is listed in the fourth and fifth columns. The R-schedule in the uniformity computation is generated by the combination of APGAN and DPPO [1]. The last column is the *buffer cost ratio* of our procedural implementation over an R-schedule calculated by the combination of

Table 1: Experimental results for real applications.

	actor count	edge count	U_1 (%)	U_2 (%)	buffer cost ratio (%)
aqmf235_2d	20	22	90	88	88
aqmf235_3d	44	50	76	70	76
aqmf23_2d	20	22	90	86	93
aqmf23_3d	44	50	80	70	87
nqmf23	32	35	82	84	86
cd2dat	8	7	42	4	9
cd2dat2	6	5	40	10	21
dat2cd	5	4	50	17	14
filtBankNu	26	28	82	83	90
cdma2k_rev	143	157	96	77	90

APGAN and DPPO. A lower ratio means that our procedural implementation consumes less buffer cost. The first five *qmf* benchmarks are multirate filter bank systems with different depths and low-pass and high-pass components. Following those are three sample rate converters: *cd2dat*, *cd2dat2*, and *dat2cd*. The function of *cd2dat2* is equivalent to *cd2dat* except for an alternative breakdown into multiple stages. A two-channel non-uniform filter bank with depth of two is given in *filtBankNu*. The last benchmark *cdma2k_rev* is a CDMA example of a reverse link using HPSK modulation and demodulation under SR3.

Uniformity and buffer cost ratio are roughly in a linear relationship in Table 1. To further explore this relationship between buffer cost ratio and uniformity, we experimented with a large set of randomly-generated SDF graphs, and the results are illustrated in Figure 11. Both charts in the figure exhibit an approximately proportional relationship between uniformity and buffer cost ratio. The lower the uniformity, the lower the buffer cost ratio.

To better understand the overheads of execution time and code size for procedural over inlined implementation, we examined the cd2dat and dat2cd examples in more de-

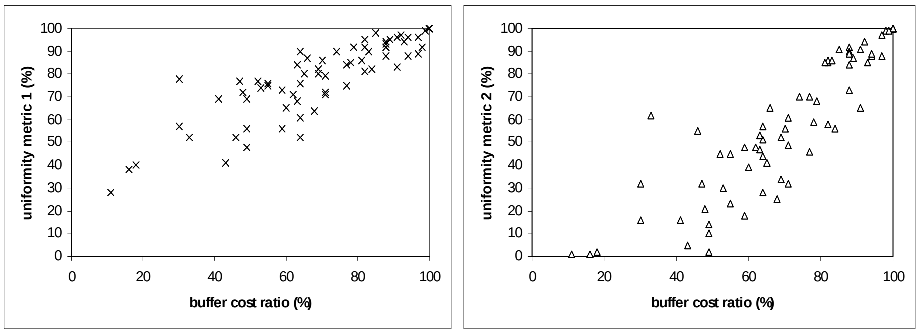


Figure 11 Relationship between uniformity and buffer cost ratio for random graphs.

tail. In the experiment for *cd2dat*, we obtained 0.75% and 10.85% execution time and code size overheads, respectively, compared to inlined implementations of the schedules returned by APGAN and GDPPO. In the experiment for *dat2cd*, the overheads observed were 1.26% and 9.45% respectively. In these experiments, we used the Code Composer Studio by Texas Instruments for the *TMS320C67x* series processors. In general the overheads depend heavily on the granularity of the actors. In the applications of Table 1, the actors are mostly of coarse granularity. However, in the presence of many fine-grained (low complexity) actors, the relative overheads are likely to increase; and for such applications, the procedural implementation approach proposed in this paper is less favorable, unless buffer memory constraints are especially severe.

8 Conclusion

In this paper, an efficient method is developed for applying subroutine call instantiation of module functionality when synthesizing embedded software from an SDF specification. This approach provides for significantly lower buffer sizes, with polynomially bounded procedure call overhead, than what is available for minimum code size, inlined schedules. This recursive decomposition approach thus provides an efficient means for integrating subroutine-based module instantiation into the design space of DSP software synthesis. We develop metrics for characterizing a certain form of uniformity in SDF schedules, and show that the benefits of the proposed techniques increase with decreasing uniformity. Directions for future work include integrating the procedural implementation approach in this paper with existing techniques for inlined implementation. For example, different subgraphs in an SDF specification may be best handled using different techniques, depending on application constraints and subgraph characteristics (e.g., based on uniformity, as defined in this paper, and actor granularity). Integration with other strategies for buffer optimization such as phased scheduling [4] and buffer merging [11] are also useful directions for further investigation.

References

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

[2] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, May 2000.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity — the Ptolemy approach. *Proceedings of the IEEE*, January 2003.

[4] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages 103-112, San Diego, California, June 2003.

[5] M. Karczmarek, W. Thies, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, April 2002.

[6] D. E. Knuth. *Seminumerical algorithms, 2nd edition, The Art of Computer Programming volume 2*, Addison-Wesley, Reading, MA, 1981.

[7] K. N. Lalgudi, M. C. Papaefthymiou, and M. Potkonjak. Optimizing computations for effective block-processing. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):604-630, July 2000.

[8] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete. Grape-II: A system-level prototyping environment for DSP applications. *IEEE Computer Magazine*, 28(2):35-43, February 1995.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of IEEE*, vol. 75, pp. 1235–1245, September 1987.

[10] R. Leupers and P. Marwedel. Time-constrained code compaction for DSP's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(1):112-122, March 1997.

[11] P. K. Murthy and S. S. Bhattacharyya. Buffer merging — a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 2004. To appear.

[12] P. K. Murthy, E. G. Cohen, and S. Rowland. System Canvas: A new design environment for embedded DSP and telecommunication systems. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, April 2001.

[13] P. R. Panda, F. Catthoor, N. D. Dutt, et. al. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation for Electronic Systems*, 6(2):149-206, April 2001.

[14] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyer. Optimum vectorization of scalable synchronous dataflow graphs. *Proceedings of the International Conference on Application Specific Array Processors*, pp. 285-296, October 1993.

[15] C. B. Robbins. *Autocoding Toolset software tools for automatic generation of parallel application software*. Technical report, Management, Communications & Control, Inc., 2002.

[16] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference*, February 2004.

[17] W. Sung and S. Ha. Memory Efficient Software Synthesis using Mixed Coding Style from Dataflow Graph. *IEEE Transactions on VLSI Systems*, Vol. 8, pp 522-526, October 2000.

[18] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, pages 83-98, February 2000.