Beyond Single-Appearance Schedules: Efficient DSP Software Synthesis Using Nested Procedure Calls

MING-YUNG KO
University of Maryland
PRAVEEN K. MURTHY
Fujitsu Labs of America
and
SHUVRA S. BHATTACHARYYA
University of Maryland

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 work on software synthesis. 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 multirate DSP applications, with moderate code and execution time overhead.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Signal Processing Systems; D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Dataflow languages*; D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*

This research was supported in part by the Semiconductor Research Corporation (2001-HJ-905).
Authors' addresses: Ming-Yung Ko, Department of Electrical and Computer Engineering, University of Maryland, College Park, Maryland 20742; email: myko@eng.umd.edu; Praveen K. Murthy, Fujitsu Laboratories of America, 1240 East Arques Avenue, M/S 345, Sunnyvale, California 94085; email: praveen.murthy@us.fujitsu.com; Shuvra S. Bhattacharyya, Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, Maryland 20742; email: ssb@eng.umd.edu.

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Synchronous dataflow, hierarchical graph decomposition, procedural implementation, memory optimization, design methodology, embedded systems, block diagram compiler

ACM Reference Format:

Ko, M.-Y., Murthy, P. K., and Bhattacharyya, S. S. 2007. Beyond single-appearance schedules: Efficient DSP software synthesis using nested procedure calls. ACM Trans. Embedd. Comput. Syst. 6, 2, Article 14 (May 2007), 23 pages. DOI = 10.1145/1234675.1234681 http://doi.acm.org/10.1145/1234675.1234681

1. INTRODUCTION AND RELATED WORK

As a result of 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 [Murthy et al. 2001], SPW from Cadence Design Systems, ADS from Agilent, Cocentric System Studio from Synopsys [Buck and Vaidyanathan 2000], GEDAE from Lockheed, and the Autocoding Toolset from Management, Communications, and Control, Inc. [Robbins 2002]. Research-oriented tools and languages related to dataflow-based DSP design include Ptolemy from U. C. Berkeley [Eker et al. 2003], GRAPE from K. U. Leuven [Lauwereins et al. 1995], Compaan from Leiden University [Stefanov et al. 2004], and StreamIt from MIT [Thies et al. 2002].

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 [Lee and Messerschmitt 1987], 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 Bhattacharyya et al. [1996], algorithms are developed to optimize buffer space while obeying the constraint of minimal code space. A multiple objective optimization is proposed in Zitzler et al. [2000] 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 [Lalgudi et al. 2000; Ritz et al. 1993].

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 Sung and Ha [2000], an alternative buffer minimization technique through transforming looped schedules is investigated. The transformation works on a certain schedule tree data structure, and the computational complexity of the transformation is shown to be polynomial in the number of leaf nodes in this schedule tree. However, since leaf nodes in the schedule tree correspond to actor appearances in the schedule, there is, in general, no polynomial bound in terms of the size of the SDF graph on the number of these leaf nodes. Therefore, no polynomial bound emerges on the complexity of the transformation technique in terms of SDF graph size. In contrast, our graph decomposition strategy extends and hierarchically applies a two-actor SDF graph scheduling theory that guarantees achieving minimal buffer requirements [Bhattacharyya et al. 1996] and a number of theorems are developed in this paper to ensure that the complexity of our approach is polynomially bounded in the size of the SDF graph.

Buffer minimization and use of subroutine calls during code synthesis have also been explored in the phased-scheduling technique [Karczmarek et al. 2003]. This work is part of the StreamIt language [Thies et al. 2002] 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 [Bhattacharyya et al. 1996]. Through extensive experiments with single-appearance scheduling, it has been observed that tightly interdependent components arise only very infrequently, in practice [Bhattacharyya et al. 1996]. 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 [Panda et al. 2001]. Because of the instruction-level parallelism capability found in many DSP processors, the study of independent register transfers is also a useful subject. The work of Leupers and Marwedel [1997] 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 intraactor

optimization in conjunction with the interactor, 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 over- and underflow problems, the total amount of data produced and consumed is required to be matched on all edges. In Lee and Messerschmitt [1987], 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, which 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 $L = (nI_1I_2 \cdots I_m)$, 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 postoptimization*) over a topological sort to generate a buffer-efficient, nested-looped schedule [Bhattacharyya et al. 1996]. In this paper, we employ the *acyclic pairwise grouping for adjacent nodes* (*APGAN*) algorithm [Bhattacharyya et al. 1996] 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

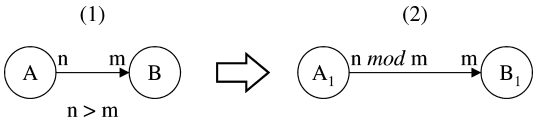


Fig. 1. A two-actor SDF graph and its reduced version.

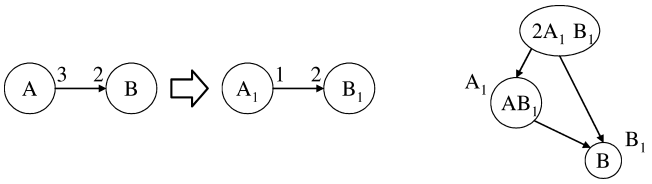


Fig. 2. A hierarchical procedural implementation of a minimum buffer schedule for the SDF graph on the left.

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 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 $3 \bmod 2 = 1$, 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 valid 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 five procedure calls, whereas the hierarchical implementation depicted in Figure 2 requires only three procedure calls. The top-most 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 B. A procedure B_1 for actor B is also needed, because it is called more than once: A_1 and the topmost procedure. 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 three 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 three procedure calls would be less than the code-size increase of using three appearances (instead of two). We would also like to clarify that *procedure calls* in this paper refer to procedure instantiation in software synthesis, rather than runtime procedure invocation.

We first state an important theorem from Bhattacharyya et al. [1996]:

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 Bhattacharyya et al. [1996]).

We denote $n + m - ged(n, m)$ 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 $gcd(n, m) = gcd(n \bmod m, m)$. Thus, the minimum buffer requirement for the reduced graph is given by $n \bmod m + m - g$, from Theorem 1. 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_1B_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. □

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* $m \mid n$.

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

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

Since $n > m$, we have to show that Eq. (1) implies $m \mid n$. The contrapositive is that if $m \mid n$ does not hold, then Eq. (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 Eq. (1), we have $m - gcd(n, m) < m < n$, meaning that the R.H.S. is $<2n$. This shows that Eq. (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 $m + n - m = n$. □

A two-actor SDF graph, where either $n \mid m$ or $m \mid n$ is called a *perfect SDF graph* (*PSG*) in this paper.

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* $m \mid n$.

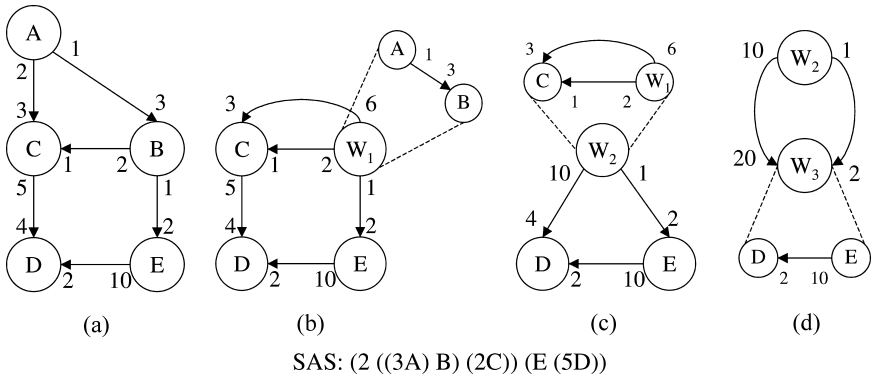


Fig. 3. An SAS showing how an SDF graph can be decomposed into a series of two-actor subgraphs.

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 that the Euclidean GCD algorithm runs in polynomial time. A complete proof is given in Ko et al. [2004]. □

Thus, we can implement the minimum buffer schedule in a nested, 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 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 [Bhattacharyya et al. 1996]. 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 nonhierarchical actors. Figure 3 shows an SDF graph, with an SAS, 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

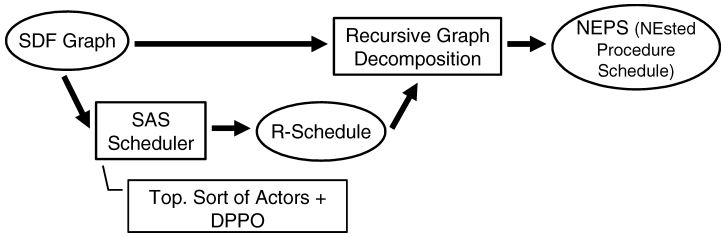


Fig. 4. An algorithm overview for arbitrary SDF graphs.

$(3A)B$. Hence, in this example, no reductions are needed at any stage in the hierarchy, 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.

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; an algorithm overview is illustrated in Figure 4. In Figure 4, our approach is termed as *NEPS* (*NEsted Procedure Schedule*). 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 this section, 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.

Although we have shown that the number of decompositions required to reach a PSG is polynomial for a two-actor SDF graph, it is not obvious from this that the complexity of our recursive decomposition approach for arbitrary graphs is also polynomial. For arbitrary graphs, the clustering process expands the produced/consumed numbers; in fact, these numbers can increase multiplicatively. Because of the nature of the logarithm operator, however, the

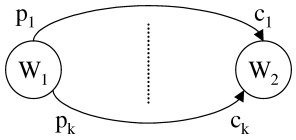


Fig. 5. A two-actor SDF multigraph.

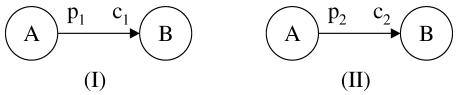


Fig. 6. Two two-actor graphs with the same repetitions vector.

multiplicatively increased rates still keep the decomposition process of polynomial complexity in the size of the input SDF graph. Full details on this complexity result can be found in Ko et al. [2004].

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:

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

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

LEMMA 1. *Suppose that* $p_1 \leq \cdots \leq p_k$. *Then* $c_1 \leq \cdots \leq c_k$.

PROOF. Suppose this is not true. Suppose that for some i and j, we have $p_i \leq p_j$, but $c_i > c_j$. Equation (2) implies that $c_i/p_i = c_j/p_j$. However, $p_i \leq p_j$ and $c_i > c_j$ implies $c_i/p_i > c_j/p_j$, contradicting Eq. (2). □

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

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

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

PROOF. Suppose that this is not the case. Suppose there is a schedule for (I) that is not valid for (II). Let σ be the firing sequence $X_1X_2 \cdots X_{q(A)+q(B)}$, 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 $n_A p_1 - n_B c_1 \geq 0$. Thus,

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

By Eq. (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. □

THEOREM 7. *The schedule that yields the VBMLB for* (I) *also yields the same VBMLB for* (II).

PROOF. Let σ be the firing sequence $X_1 X_2 \cdots X_{q(A)+q(B)}$, 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

$$n_A^* p_1 - n_B^* c_1 = p_1 + c_1 - gcd(p_1, c_1)$$

and for all other n_A and n_B in σ,

$$n_A p_1 - n_B c_1 \leq n_A^* p_1 - n_B^* c_1 \tag{4}$$

We have that

$$\begin{aligned}
n_A^* p_2 - n_B^* c_2 &= n_A^* p_2 - n_B^* \frac{p_2}{p_1} c_1 \\
&= p_2 \left(n_A^* - \frac{c_1}{p_1} n_B^* \right) \\
&= p_2 \left(n_A^* + \frac{p_1 + c_1 - gcd(p_1, c_1) - n_A^* p_1}{p_1} \right) \\
&= p_2 \left(1 + \frac{c_1}{p_1} - \frac{gcd(p_1, c_1)}{p_1} \right) \\
&= p_2 + c_2 - \frac{gcd(p_1, c_1)}{p_1} p_2 \\
&= p_2 + c_2 - gcd\left(p_1 \frac{p_2}{p_1}, \frac{c_1 p_2}{p_1} \right) \\
&= p_2 + c_2 - gcd(p_2, c_2)
\end{aligned}$$

By Eq. (4), we have

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

Thus,

$$\begin{aligned}
n_A p_2 - n_B c_2 &\leq \left(n_B \frac{c_1}{p_1} + 1 + \frac{c_1}{p_1} - \frac{gcd(p_1, c_1)}{p_1} \right) p_2 - n_B c_2 \\
&= p_2 + c_2 - gcd(p_2, c_2)
\end{aligned}$$

Hence, this shows that yields the VBMLB for (II) also. □

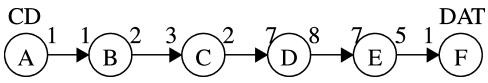


Fig. 7. A CD-DAT sample rate converter example.

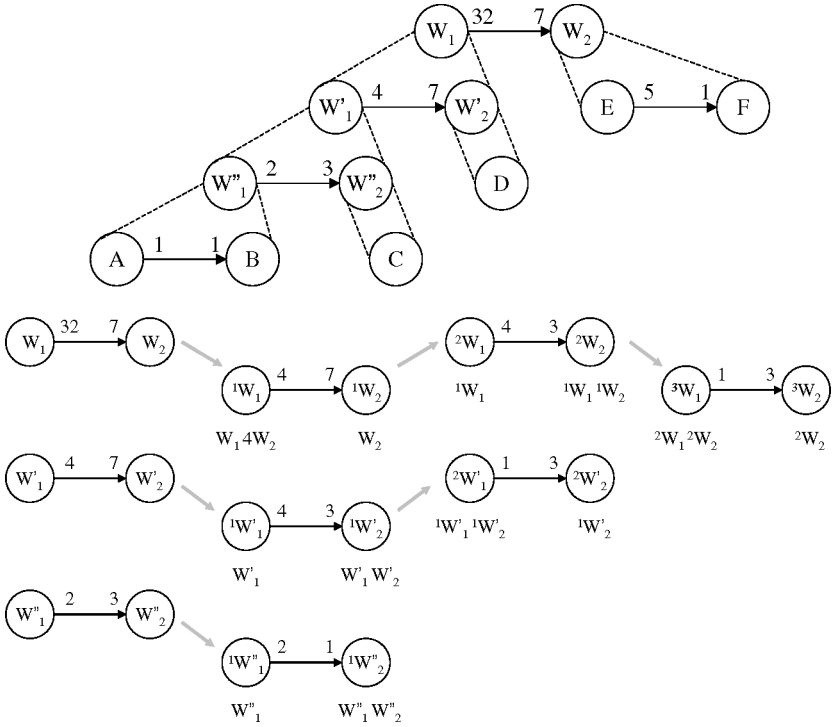


Fig. 8. The recursive decomposition of the CD-DAT graph.

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–callee relationships. The generated C style code is shown in Figures 10 with 11 procedure calls. The number of required procedure calls is the *code cost* of the NEPS, while the *buffer cost* of the NEPS is the amount of buffer space required.

Definition 1. The number of procedure calls required for an NEPS implementation is the number of nodes in the procedure call dependence graph, as in Figure 9.

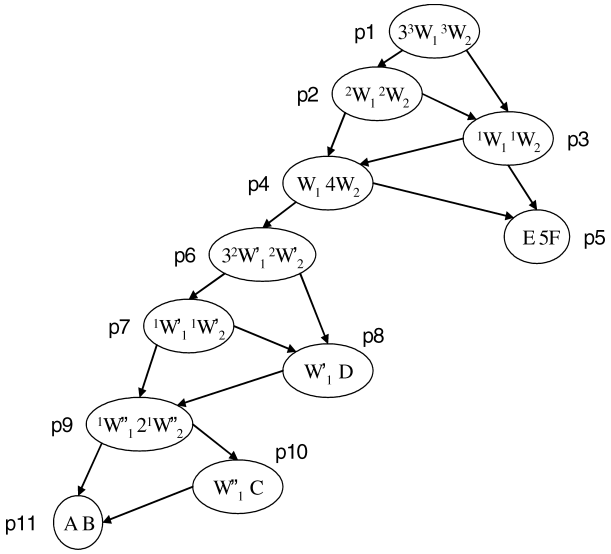


Fig. 9. Procedure call sequence for the CD-DAT example.

```
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;
}
```

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

The total buffer memory requirement of the CD-DAT 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 [Murthy and Bhattacharyya 2004].

6. EXTENSION TO GRAPHS WITH CYCLES

To deal with cycles, *the loose interdependence algorithm framework* (*LIAF*) [Bhattacharyya et al. 1996] is a systematic approach to collaborate with NEPS. LIAF is basically a recursive strategy in deriving and scheduling (generating

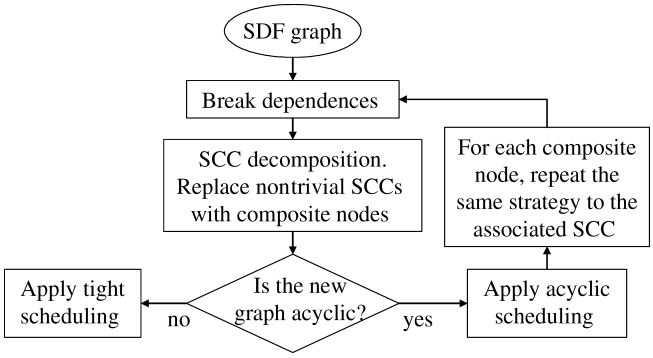


Fig. 11. Flow chart of LIAF approach.

SAS) new graph topologies where nodes may be the original nodes (called *atomic* nodes) or subgraphs of the previous graph (called *composite* nodes). Composite nodes are solved in the same fashion until atomic nodes are reached or such a recursion is infeasible. In LIAF, derivation of new graph topologies begins with breaking data dependences of certain edges as if they were removed, particularly those on cycles and with sufficient delays. Such delay sufficiency is determined by the requirement to make a cycle deadlock free and, therefore, schedulable. When all the dependences bearing enough delays are broken, *strongly connected component* (*SCC*) decomposition is then applied to the graph. A new acyclic graph can be formed from newly created composite nodes in place of the SCCs and acyclic schedules can be, hence, efficiently computed. *Loose interdependence* refers to the situation that an acyclic structure can be transformed from a cycle in this way. It is *tight interdependence*, otherwise. We call a graph a *tightly interdependent component* if tight interdependence exists in that graph. A summary of LIAF is sketched in Figure 11.

An example of LIAF scheduling is given in Figure 12. The SDF graph G in (a) is strongly connected and has three simple cycles. According to LIAF, the criterion to break a data dependence is

$$delay(e) \geq prd_G(e) \cdot q_G(src(e)) \tag{5}$$

where G is the associated graph containing e. The rationale behind the delay sufficiency is that $snk(e)$ can always obtain enough tokens to consume in an SAS scheduling and the associated data dependence does not impose any restriction on the firing sequence of $src(e)$ and $snk(e)$ anymore. Therefore, the dependence of B from C in the original graph G, C from E in SCC_Y can be broken because of

$$12 = delay(CB) \geq prd_G(CB) \cdot q_G(C) = 1.12$$

After dependence breaking, SCC decomposition is performed and nontrivial SCCs (SCC_X and SCC_Y) can be replaced with newly created composite nodes (X and Y). The new graph G in (c) is then acyclic and the SAS is $(2X)(3Y)$. To complete the synthesis, each SCC is treated as an independent graph so that data dependence breaking, SCC decomposition, and acyclic scheduling can be applied recursively until stop condition is reached. In (e), the dependence of C

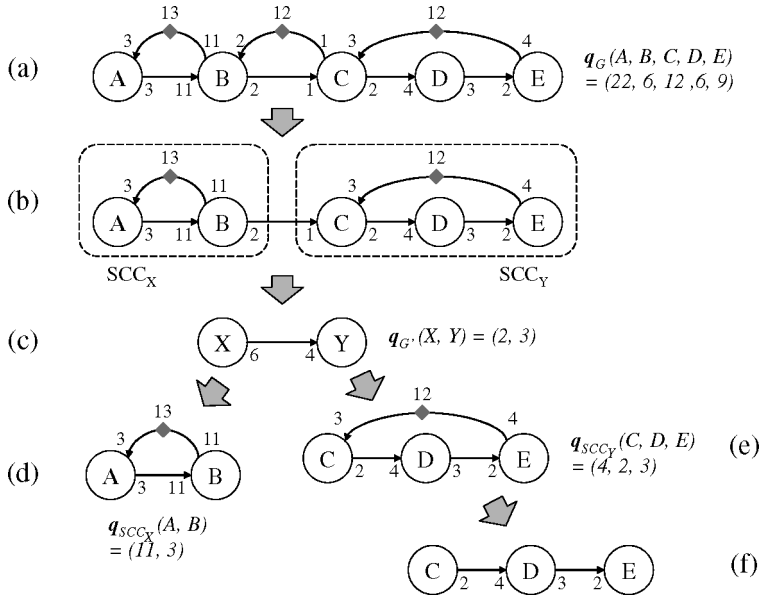


Fig. 12. An example demonstrating LIAF. Repetitions vectors are also given next tot the associated graphs. The graph of (d) has tight interdependence.

from E can be broken because of

$$12 = delay(EC) \geq prd_{SCC_Y}(CB) \cdot q_{SCC_Y}(C) = 4.3$$

Once again, acyclic structure is encountered in (f) and we can obtain the SAS $(2(2C)D)3E)$ through the combination of APGAN and DPPO, which is efficient in deriving minimum buffer cost SAS. Unfortunately, the graph in (d) (also SCC_X) does not satisfy Eq. (5),

$$13 = delay(BA) < (prd_{SCC_X}(B) \cdot q_{SCC_X}(B)) = 11.3 = 33$$

and is, thus, a tightly interdependent component. Since there is no efficient tight interdependence SAS scheduling thus, the LIAF will give us $(2X)(3(2(2C)D)3E))$ with X unresolved.

To handle tight interdependence, strategies other than SAS scheduling are needed. One effective approach is the buffer cost minimization heuristic of Bhattacharyya et al. [1996], which gives us a *multiple appearance schedule* (*MAS*) ($AAAABAAAABAAAB$) for SCC_X. The MAS can be compressed by CDDPO (discussions in Section 7.1) and we obtain $(2(4A)B)((3A)B)$, which is still a MAS for this case. Inlined implementation for MAS is costly because of multiple copies of code and procedural implementation is favored. Two ways of synthesizing SCC_X in procedures are presented in Figure 13: (a) is the implementation of $(2(4A)B)((3A)B)$ and (b) is our NEPS approach. Compared to a total of five procedures required in (a), our NEPS needs only four procedures while achieving the same buffer cost minimum of 13.

The example shown in Figure 12d suggests less delays required by NEPS to break the dependence of A from B. To break the dependence, a minimum

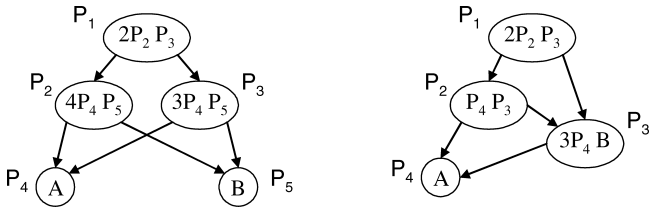


Fig. 13. Two procedural implementations of SCC_X (a) minimum buffer cost heuristic plus CDDPO compression; (b) our NEPS approach.

of 33 delays are required by LIAF, according to Eq. (5). In contrast, our NEPS demands a much lower minimum of only 13 delays. Further more, it is observed that $13 = 11 + 3 - gcd(11, 3)$. In other words, the NEPS delays requirement is equivalent to the VBMLB in this example. However, the truth of the lower delays requirement, in general cases, needs careful verification and is, therefore, left as future work.

In summary, NEPS fits into the LIAF framework neatly, and enables superior scheduling to be done for arbitrary SDF graphs. While tightly interdependent components have been problematic for previous techniques, NEPS provides a way of scheduling these more efficiently as well.

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 *nonuniform* 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 nonuniform 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.

Table I. Experimental Results for Real Applications

	Actors Count	Edges 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

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%; the results are listed in Table I. 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 [Bhattacharyya et al. 1996]. The last column is the buffer cost ratio of our procedural implementation over an R-schedule calculated by the combination of 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- 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 nonuniform 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 I. To further explore this relationship between *buffer cost ratio* and uniformity, we experimented with a large set of randomly generated SDF graphs; the results are illustrated in Figure 14. 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 detail. 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

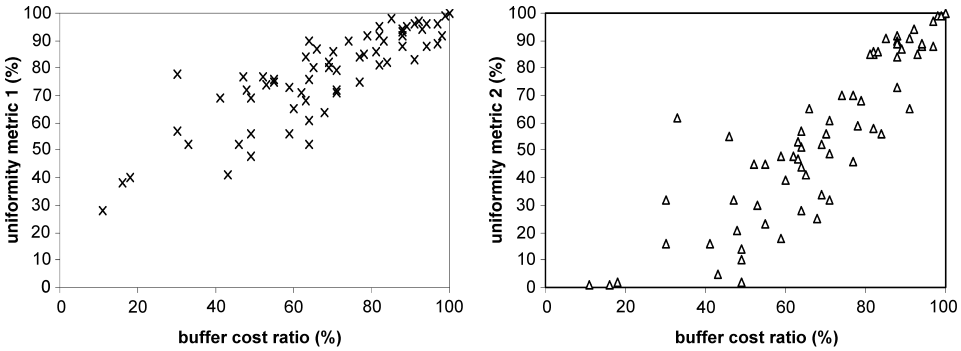


Fig. 14. Relationship between uniformity and buffer cost ratio for random graphs.

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; for such applications, the procedural implementation approach proposed in this paper is less favorable, unless buffer memory constraints are especially severe.

Employing VBMLB as the cost criterion for GDPPO gives better results than employing the BMLB as the cost criterion for GDPPO. Previously, the *BMLB* (*buffer memory lower bound*) has been used as the cost criteria for GDPPO in deriving buffer optimal SAS [Bhattacharyya et al. 1996]. In the new formulation here, as suggested in Section 4, we use the VBMLB as the cost criterion for GDPPO in calculating R-schedules for NEPS implementations. Though the R-schedules generated by GDPPO for the CD-DAT example in Figure 7 are the same under either criterion, GDPPO with VBMLB as the cost criterion can result in smaller buffer costs, as shown in Figure 15. We only show those benchmarks in Figure 15 for which there is a buffer cost reduction; the other examples all have the same cost under both criteria and, hence, no reduction. The buffer cost reduction caused by using VBMLB instead of BMLB is displayed in percentage as the bars' length. That is, the reduction is defined as

$$\frac{bufferCost(BMLB) - bufferCost(VBMLB)}{bufferCost(BMLB)}$$

where $bufferCost(X)$ means the buffer cost of NEPS taking DPPO results as the input R-schedule with the associated cost criterion X. The figure shows that there is a definite advantage in using VBMLB as the cost criterion for GDPPO under NEPS implementations.

7.1 Comparison with Minimum Buffer Cost Schedules

Thus, we have compared the NEPS implementation to the SAS implementation and shown that while the SAS has minimum overall code cost, it has considerably higher buffer cost compared to NEPS. Here we compare NEPS with the converse of SAS: schedules of minimum overall buffer cost that necessarily have poor code costs.

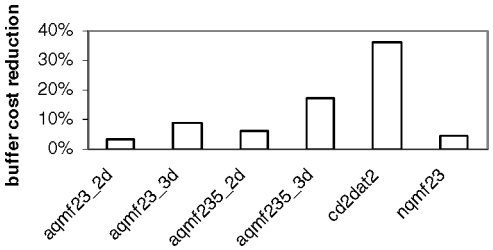


Fig. 15. Buffer cost reduction due to VBMLB as the GDPPO cost criterion compared to BMLB criterion.

An arbitrary, *raw* schedule for an SDF graph is an admissible actor firing sequence that does not have any form of schedule or code compression. For instance, organizing loops in the schedule is a form of schedule compression. For a raw schedule, the code cost is the sum of all actor firings counts (actors' repetitions). The repetitions of an actor may be exponential in the size of an SDF graph; hence, a raw schedule has exponential length in the size of the SDF graph. Whether the raw schedule is inlined or implemented via procedure calls does not matter; it will have very poor code cost. For example, the CD-DAT system in Figure 7 has 612 actor firings in a valid raw schedule. Even if each occurrence of a firing is replaced with a procedure call, the implementation will still demand large code space because of the 612 procedure calls required.

All known heuristics for generating minimum buffer schedules for SDF graphs generate them in raw form; see, for example, Bhattacharyya et al. [1996] and Cubric and Panangaden [1993]. *CDPPO* [Bhattacharyya et al. 1995] is an adaptation of GDPPO that minimizes schedule size for an arbitrary sequence of actor firings. Like GDPPO, CDPPO uses dynamic programming to build nested loops to compress repeated firings optimally. To demonstrate the effect of CDPPO, let us take the simple example in Figure 3a. The minimum buffer schedule is given by

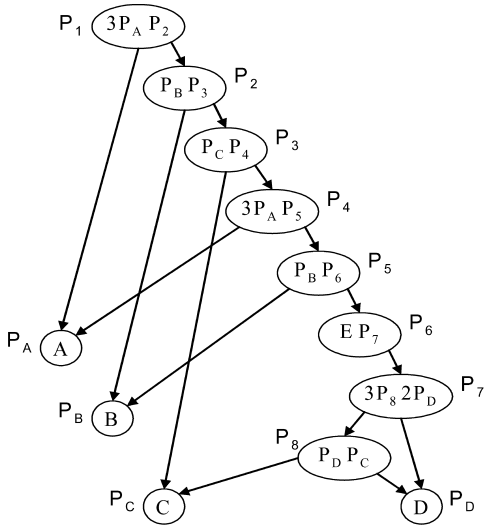
$$(AAABCAAABEDCDCDCDD)$$

CDPPO will organize loops in this schedule, thereby compressing it:

$$((3A)(B(C((3A)(B(E((3DC)(2D))))))))) \tag{6}$$

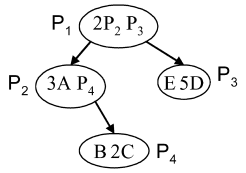
Since CDPPO does not change the firing sequence of the schedule, the buffer cost of the schedule is the same as before compression. Both of the above schedules result in a buffer cost of 35, compared to 43 given by SAS and NEPS.

We now want to investigate how NEPS implementations compare to optimally looped minimum buffer schedules returned by CDPPO applied to raw minimum buffer schedules. The loop hierarchy of the looped schedule suggests a natural way for a procedural implementation, where the loop nests become procedures recursively. The procedure call sequence of Eq. (6) is shown in Figure 16a. As can be seen in the figure, subschedules are implemented in procedures. Actors A, B, C, and D are implemented as procedures as well, because they appear multiple times in the schedule. Actor E is not implemented as a procedure, but is inlined instead in procedure p_6. This procedural implementation



CDPPO schedule:
((3 A) (B (C ((3 A) (B (E ((3 D C) (2 D))))))))
Total **12** procedures required.

(a)



NEPS schedule:
((2 (3 A) (B (2 C))) (E (5 D)))
Total **4** procedures required.

(b)

Fig. 16. Procedure call sequence for CDPPO and NEPS schedules of the SDF graph in Figure 3a.

is arguably the most code efficient realization of the looped minimum buffer schedule. The code cost of this implementation is the number of nodes in the procedure call graph.

Hence, the minimum buffer CDPPO schedule of Eq. (6) requires a total of 12 procedure calls, while our NEPS implementation requires 4 procedure calls, as shown in Figure 6b. The buffer cost overhead of NEPS over the minimum buffer CDPPO schedule is 23%, but there is a 67% reduction of procedure count of NEPS over that of the minimum buffer CDPPO schedule.

Experiments on random graphs show that this advantage that NEPS has over minimum buffer CDPPO schedules (of having a small buffer cost overhead in return for a large code size savings), holds, in general. The results are summarized in Figure 17. In the figure, buffer cost overhead is defined as

$$\frac{bufferCost(NEPS) - bufferCost(CDPPO)}{bufferCost(CDPPO)}$$

where $bufferCost(X)$ is the buffer cost computed by X and Y is either NEPS or minimum buffer CDPPO schedule. Procedure count reduction is formulated as

$$\frac{procCount(CDPPO) - procCount(NEPS)}{procCount(CDPPO)}$$

where $procCount(X)$ is the procedure count of the schedule X. In Figure 17, most of the points fall in the area of less than 30% buffer cost overhead and 70–90% procedure count reduction. Thus, with minor buffer cost overhead, NEPS

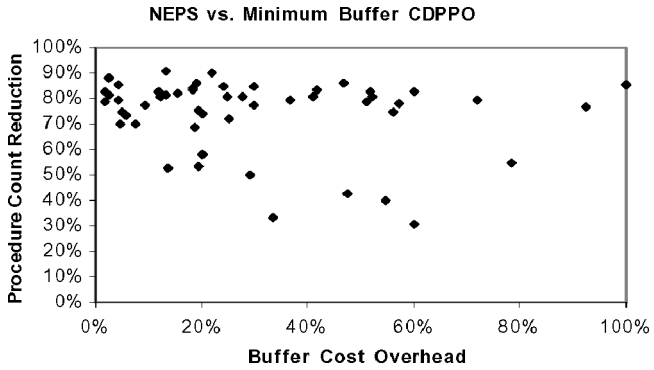


Fig. 17. Comparison of NEPS with the minimum buffer CDPPO.

can result in many fewer procedure calls and, therefore, a much smaller code space requirement.

Because the problem of computing minimum buffer schedules is NP-complete even for homogenous SDF graphs [Bhattacharyya et al. 1996], we have to rely on heuristics for generating these schedules, in general (if we want efficiency) [Ade et al. 1997; Bhattacharyya et al. 1996; Cubric and Panangaden 1993]. While some heuristics might produce optimal results for some subclasses of graphs, like trees, they will necessarily be suboptimal, in general. Thus, there does not appear to be a rigorous experimental evaluation of the quality of these minimum buffer heuristics on general graphs. In our experiment with random graphs, we observed that NEPS actually beats the minimum buffer heuristic of Bhattacharyya et al. [1996], in many cases. While this is yet another advantage for NEPS, in that it is apparently better than even a heuristic designed to purely return minimum buffer schedules, we believe that further evaluation is needed of minimum buffer heuristics before a more definitive claim is made of the ability of NEPS to actually function as a minimum buffer heuristic. Hence, in our comparison above, we have omitted the cases where NEPS outperformed the minimum buffer heuristic and only compared it to cases where it was worse. For future work we would like to do a thorough comparison of the heuristics of Ade et al. [1997], Bhattacharyya et al. [1996], and Cubric and Panangaden [1993] against NEPS and see where NEPS fits in.

A major advantage of NEPS is the polynomial-time computational complexity. Though CDPPO is $O(n^4)$, where n is the number of actor firings [Bhattacharyya et al. 1995], n is potentially exponential in the size of the SDF graph. Thus, CDPPO computation is inefficient for large graphs while NEPS remains efficient.

Some more experiments are conducted in order to compare the performance of minimum buffer CDPPO schedules with NEPS and SAS. The results are depicted in Figure 18. In the bar chart, where buffer costs are compared, buffer costs are normalized by the cost of SAS (computed by APGAN and GDPPO), because SAS has the worst buffer cost. That is, each bar has length

$$\frac{bufferCost(X)}{bufferCost(SAS)}$$

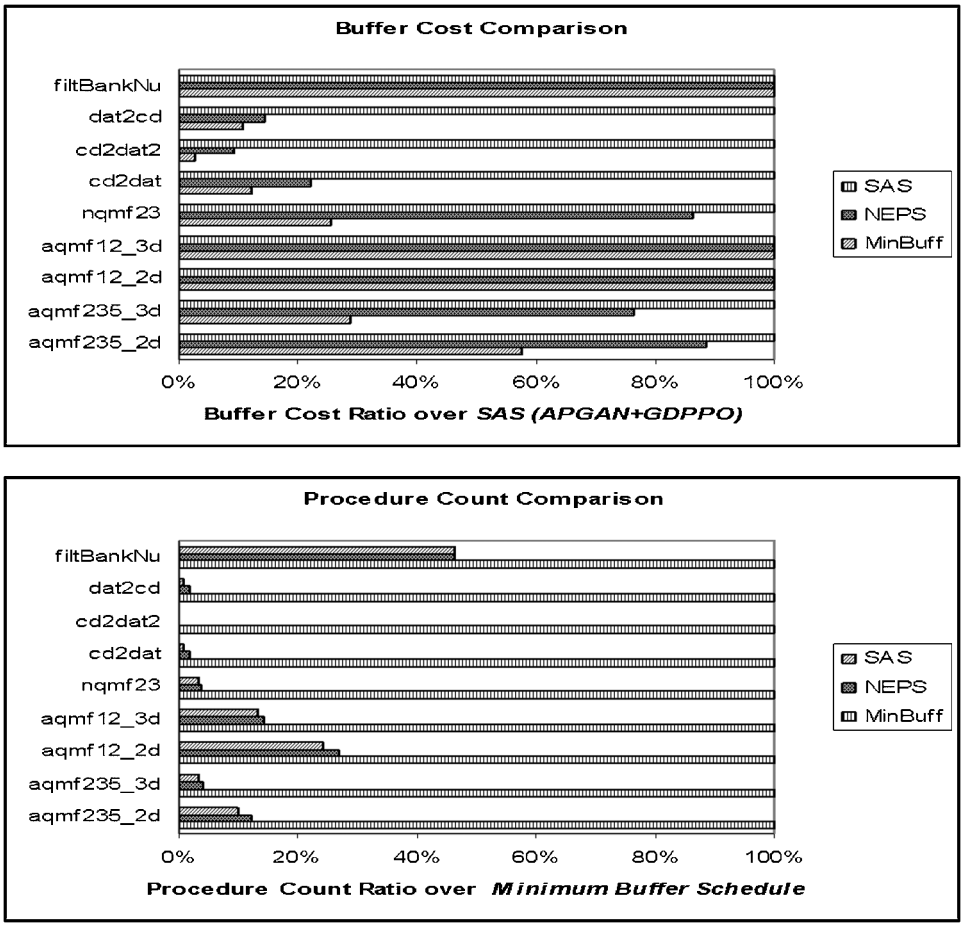


Fig. 18. Comparison of minimum buffer schedule, NEPS, and SAS (APGAN + GDPPO).

in percentage, where $bufferCost(X)$ is the buffer cost of schedule X for the particular benchmark. Similarly, procedure counts are also normalized by the largest one, which are produced by the minimum buffer CDPPO schedule. Each bar in the procedure count comparison chart has length

$$\frac{procCount(X)}{procCount(MinBuff)}$$

in percentage, where $procCount(X)$ is the number of procedure calls required in the procedural implementation of schedule X for the particular benchmark. The buffer cost chart in Figure 18 shows that NEPS is somewhere in between minimum buffer CDPPO and SAS, many times closer to minimum buffer CDPPO than SAS. The procedure cost chart shows that NEPS is again between SAS and minimum buffer CDPPO, but much closer to SAS, in all instances. Thus, NEPS gives us a nice trade-off between code size and buffer size and gives implementations where the overhead over the best possible for both metrics is low. In the space of optimizing for two opposing criteria, namely, buffer

size and code size, the NEPS implementation is a useful and efficient Pareto point.

8. CONCLUSION

In this paper, we have developed an efficient method 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 approach also provides for significantly lower code space requirements than efficiently looped minimum buffer schedules with not much buffer cost overhead. 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 [Karczmarek et al. 2003] and buffer merging [Murthy and Bhattacharyya 2004] are also useful directions for further investigation.

REFERENCES

ADE, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1997. Data Memory Minimization for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets. In *Proceedings of Design Automation Conference (34th DAC)*, Anaheim, CA. 64–69.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1995. Optimal Parenthesization of Lexical Orderings for DSP Block Diagrams. In *Proceedings of the International Workshop on VLSI Signal Processing*, Sakai, Osaka, Japan.

BHATTACHARYYA, S. S., MURTHY, P. K., AND LEE, E. A. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publ., Norwell, MA.

BUCK, J. AND VAIDYANATHAN, R. 2000. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the International Workshop on Hardware Software Codesign* (May), San Diego, CA. 142–146.

CUBRIC, M. AND PANANGADEN, P. 1993. Minimal memory schedules for dataflow networks. In *Proceedings of the 4th International Conference on Concurrency Theory* (CONCUR '93, Hildesheim, Germany, Aug. 1993), Lecture Notes in Computer Science 715, 368–383.

EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003.T aming heterogeneity—the Ptolemy approach. In *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software* (Jan.), 91, 1, 127–144.

KARCZMAREK, M., THIES, W., AND AMARASINGHE, S. 2003. Phased scheduling of stream programs. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems* (LCTES'03) (June), San Diego, CA. 103–112.

KO, M., MURTHY, P. K., AND BHATTACHARYYA, S. S. 2004. Compact procedural synthesis of DSP software through recursive graph decomposition. Tech. Rep. UMIACS-TR-2004-41, Institute for Advanced Computer Studies, University of Maryland at College Park.

LALGUDI, K. N., PAPAEFTHYMIOU, M. C., AND POTKONJAK, M. 2000. Optimizing computations for effective block-processing. *ACM Transactions on Design Automation of Electronic Systems* (July), 5, 3, 604–630.

LAUWEREINS, R., ENGELS, M., ADE, M., AND PEPERSTRAETE, J. A. 1995. Grape-II: A system-level prototyping environment for DSP applications. *IEEE Computer Magazine* (Feb.), 28, 2, 35–43.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous dataflow. In *Proceedings of IEEE* (Sept.), 75, 1235–1245.

LEUPERS, R. AND MARWEDEL, P. 1997. Time-constrained code compaction for DSP's.*IEEE Transactions on VLSI Systems* (Mar.), 5, 1, 112–122.

MURTHY, P. K. AND BHATTACHARYYA, S. S. 2004. Buffer merging: a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems* (Apr.), 9, 2, 212–237.

MURTHY, P. K., COHEN, E. G., AND ROWLAND, S. 2001. System Canvas: A new design environment for embedded DSP and telecommunication systems. In *Proceedings of the International Workshop on Hardware/Software Co-Design* (Apr.).

PANDA, P. R., CATTHOOR, F., DUTT, N. D., DANCKAERT, K., BROCKMEYER, E., KULKARNI, C., VANDERCAPPELLE, A., AND KJELDSBERG, P. G. 2001. Data and Memory Optimization Techniques for Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems* (Apr.), 6, 2, 149–206.

RITZ, S., PANKERT, M., ZIVOJNOVIC, V., AND MEYER, H. 1993. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application Specific Array Processors* (Oct.), 285–296.

ROBBINS, C. B. 2002. *Autocoding Toolset Software Tools for Automatic Generation of Parallel Application Software.* Tech. rep., Management, Communications & Control, Inc.

STEFANOV, T., ZISSULESCU, C., TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. 2004. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Design, Automation and Test in Europe Conference* (Feb.).

SUNG, W. AND HA, S. 2000. Memory efficient software synthesis using mixed coding style from dataflow graph. *IEEE Transactions on VLSI Systems* (Oct.), 8, 522–526.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction* (Apr.), Grenoble.

ZITZLER, E., TEICH, J., AND BHATTACHARYYA, S. S. 2000. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* (Feb.). 83–98.

Received February 2005; revised September 2005; accepted December 2005