A HIERARCHICAL MULTIPROCESSOR SCHEDULING FRAMEWORK FOR SYNCHRONOUS DATAFLOW GRAPHS

José Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee

Technical Report: UCB/ERL M95/36

May 30, 1995

ABSTRACT

This paper discusses a hierarchical scheduling framework to reduce the complexity of scheduling synchronous dataflow (SDF) graphs onto multiple processors. The core of this framework is a clustering algorithm that reduces the number of nodes before expanding the SDF graph into a precedence DAG (directed acyclic graph). The internals of the clusters are then scheduled with uniprocessor SDF schedulers which can optimize for memory usage. The clustering is done in such a manner as to leave ample parallelism exposed for the multiprocessor scheduler. The advantages of this framework are demonstrated with several practical, real-time examples.

This research was partially funded as part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the Semiconductor Research Corporation (project 94-DC-008), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bellcore, Bell Northern Research, Dolby Laboratories, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, and Rockwell. José Luis Pino is also supported by AT&T Bell Laboratories as part of the Cooperative Research Fellowship Program.

J. L. Pino and E. A. Lee are with the Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, California 94720, USA.

S. S. Bhattacharyya is with the Semiconductor Research Laboratory, Hitachi America, Ltd., 201 East Tasman Drive, San Jose, California 95134, USA.

1. Introduction

Dataflow is a natural representation for signal processing algorithms. One of its strengths is that it exposes parallelism by expressing only the actual data dependencies that exist in an algorithm. Applications are specified by a dataflow graph in which the nodes represent computations, and data tokens flow between them along the arcs of the graph. Ptolemy [2] is a framework that supports dataflow programming (as well as other computational models, such as discrete event).

Generating a stand-alone application from a dataflow graph description requires two phases: scheduling and synthesis [3]. In the scheduling phase, the dataflow graph is partitioned for parallel execution. We splice send and receive nodes into the graph for interprocessor communication. These nodes do the synchronization necessary for a self-timed implementation [4]. For each target processor, a sequence of node firings is determined. In the synthesis phase, the code segments associated with each node are stitched together, following the order specified by the scheduler. Commercial systems that use this “threading” technique include Comdisco’s DPC [5] and CADIS’s Descartes [6]. The techniques we describe here are complementary to those in DPC and Descartes, and could, in principle, be used in combination with them.

There are several forms of dataflow defined in Ptolemy. In synchronous dataflow (SDF) [7], the number of tokens produced or consumed in one firing of a node is constant. This property makes it possible to determine execution order and memory requirements at compile time. Thus these systems do not have the overhead of run-time scheduling (in contrast to dynamic dataflow) and have very predictable run-time behavior. The production/consumption property on the arcs also provides a natural representation of multirate signal processing blocks [8]. In this paper, we will focus on scheduling SDF graphs onto multiple processors.

In the following sections, we will review scheduling of SDF graphs, including uniprocessor scheduling and DAG construction. Then we will present the SDF composition theorem which is needed to apply clustering heuristics on the SDF graph. Following that we discuss the clustering techniques that comprise the hierarchical scheduling framework and the automated hierarchical scheduling algorithm.

2. Background

Figure 1 shows a simple SDF graph. In this graph, node A produces two tokens and node B consumes three tokens for each firing. In a valid SDF schedule, the first-in/first-out (FIFO) buffers on each arc return to their initial state after one schedule period. Balance equations are written for each arc and an integral repetitions vector is found that solves this system of equations [7]. In this simple example, the balance equation for the arc is: $2 \times R_A = 3 \times R_B$. Any vector of the form $\begin{bmatrix} R_A & R_B \end{bmatrix} = \begin{bmatrix} 3n & 2n \end{bmatrix}, n \in Z^+$ is a solution to the balance equation. For a given SDF graph, either the balance equations do not have a nontrivial solution (a solution other than the zero vector), or there exists a unique minimal solution whose components are all positive integers [7]. This unique minimum vector is called the **repetitions vector** and it is denoted by the symbol $\mathbf{q}$. For the example in figure 1, the repetitions vector is given by $\mathbf{q} = \begin{bmatrix} \mathbf{q}(A) & \mathbf{q}(B) \end{bmatrix} = \begin{bmatrix} 3 & 2 \end{bmatrix}$. Note that our convention is to represent the component of the vector $\mathbf{q}$ that corresponds to a node x using the functional notation $\mathbf{q}(x)$.

An SDF graph is **consistent** if it is not deadlocked, and a repetitions vector exists. Given a consistent SDF specification, we can construct a schedule at compile-time that can be iterated an indefinite number of times without requiring unbounded memory. Such a schedule can be constructed by invoking each actor x exactly $\mathbf{q}(x)$ times, and ensuring that the data precedences defined by the SDF graph are respected. For figure 1, one such schedule is $AABAB$.

2.1 Notation

We use the following notational conventions when working with SDF graphs.

- $G = (V, E)$: A directed graph, G, made up of the set of nodes V, and set of arcs E[1].
- κ_α: The number of samples consumed on the SDF arc α, per sink invocation.
- ρ_α: The number of samples produced on SDF arc α, per source invocation.

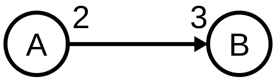


Figure 1. A simple SDF graph.

- δ_α : The number of initial samples (“delay”) on the SDF arc α.
- α_x : The set of all SDF arcs that are connected to node *x*.
- $src(\alpha)$: The node that produces the tokens on arc α.
- $snk(\alpha)$: The node that consumes the tokens produced on arc α.
- A **valid schedule** for a consistent SDF graph is a schedule that respects the data dependencies in the graph (does not deadlock), and invokes each node x exactly $\mathbf{q}(x)$ times.
- If Z is a subset of actors in a consistent SDF graph G, we define

  $$\chi(Z) \equiv gcd(\{\mathbf{q}(z) \mid (z \in Z)\}),$$

  where *gcd* denotes the greatest common divisor. The quantity $\chi(Z)$ can be viewed as the number of times that a valid schedule for G invokes the subsystem corresponding to Z [16].
- Similarly, if x and y are distinct nodes in a consistent SDF graph, then

$$Q(x, y) \equiv \frac{\mathbf{q}(x)}{\chi(\{x, y\})} = \frac{\mathbf{q}(x)}{gcd(\{\mathbf{q}(x), \mathbf{q}(y)\})}.$$

We can view $Q(x, y)$ as the number of times that node x is invoked in a single invocation of the subsystem $\{x, y\}$.

- Two nodes in a directed graph are **adjacent** if there is an arc directed from one of the nodes to the other.
- Given a consistent SDF graph G, a subset $Z = \{z_1, z_2, \ldots, z_n\}$ of nodes in G is a **uniform repetition count (URC)** subset if $\mathbf{q}(z_1) = \mathbf{q}(z_2) = \ldots = \mathbf{q}(z_n)$. If Z is a URC subset, the associated subgraph is called a **URC subgraph**.
- A **path** in a directed graph is a finite, nonempty sequence of arcs $(e_1, e_2, \ldots, e_n)$ such

[1] Technically, practical SDF graphs may arise that are directed *multigraphs* — graphs in which more than one arc can exist for the same pair of source and sink nodes — rather than directed graphs. However, the distinction between directed multigraphs and directed graphs is not important for the developments of this paper: our results can easily be extended to handle directed multigraphs. For clarity however, we restrict our discussion to directed *graphs*, rather than *multigraphs*.

that $snk(e_i) = src(e_{i+1})$ for $i = 1, 2, \ldots, (n-1)$. We say that a path $(e_1, e_2, \ldots, e_n)$ is **directed from** $src(e_1)$ **to** $snk(e_n)$; we say that this path **traverses** $src(e_1), src(e_2), \ldots, src(e_n), snk(e_n)$; and we may write $src(e_1) \to src(e_2) \to \ldots \to src(e_n) \to snk(e_n)$ as an alternative representation of the path. The path $(e_1, e_2, \ldots, e_n)$ is **simple** if $src(e_1), src(e_2), \ldots, src(e_n)$ are all distinct, and it is a **cycle** if $src(e_1) = snk(e_n)$.

- We denote the set of positive integers by Z^+.
- The precedence graph (defined below) of an SDF graph G is denoted by $PRG(G)$.
- Given a problem P, if $p_1, p_2, \ldots, p_n$ are real-valued parameters of a problem instance, and A is an algorithm to solve P, then A is $O(f(p_1, p_2, \ldots, p_n))$ ($\Omega(f(p_1, p_2, \ldots, p_n))$) if for sufficiently large $p_1, p_2, \ldots, p_n$, the number of elementary computational steps required by A is bounded above (below) by a constant multiple of $f(p_1, p_2, \ldots, p_n)$.
- If r is a real number, $\lfloor r \rfloor$ denotes the largest integer that is less than or equal to r.

2.2 SDF graph to DAG translation

To schedule SDF graphs onto multiple processors, a **precedence graph** is constructed from the original SDF graph. In general, the SDF graph exposes some of the functional parallelism in the algorithm; the precedence graph may reveal more functional parallelism, and in addition, it exposes the data parallelism available. The precedence graph for the SDF graph of figure 1 is shown in figure 2. Notice that for each node in the original SDF graph, there are multiple nodes in the precedence graph corresponding to the repetition counts derived from the balance equations.

Formally, the precedence graph is constructed by first instantiating $\mathbf{q}(x)$ nodes, labeled $x_1, x_2, \ldots, x_{\mathbf{q}(x)}$, for each node x of the SDF graph. Each precedence graph node x_i corresponds to the ith invocation of x in an iteration of a valid schedule. For each arc α in the SDF graph, an arc in the precedence graph is instantiated from $src(\alpha)_i$ to $snk(\alpha)_j$ for each ordered pair (i, j) that satisfies $1 \le i \le \mathbf{q}(src(\alpha))$, $1 \le j \le \mathbf{q}(snk(\alpha))$, and at least one of the following two conditions:

$$(i-1)\,\rho_\alpha + \delta_\alpha \le (j-1)\,\kappa_\alpha < i\rho_\alpha + \delta_\alpha, \underline{or} \qquad (1)$$

$$(j-1)\,\kappa_\alpha < (i-1)\,\rho_\alpha + \delta_\alpha < j\kappa_\alpha\,. \qquad (2)$$

The following fact is a straightforward consequence of the developments in [7].

Fact 1: An SDF graph is deadlocked if and only if its precedence graph contains a cycle.

Thus, if an SDF graph is consistent, then its precedence graph is guaranteed to be acyclic. We refer to the precedence graph of a consistent SDF graph G as the **precedence DAG** (directed acyclic graph) of G, or simply as the **DAG** of G.

Unfortunately, the expansion due to the repetition count of each SDF node can lead to an exponential growth of nodes in the DAG. This growth has been overlooked in previous SDF multiprocessor scheduling work [9, 10]. An SDF graph that exhibits this growth is shown in figure 3. It is easily seen that the number of nodes in the corresponding DAG is

$$1 + M + M^2 + \ldots + M^{N-1}\,.$$

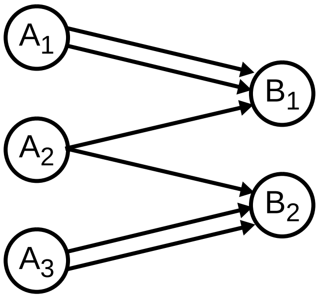


Figure 2. The precedence graph for the SDF graph of figure 1.

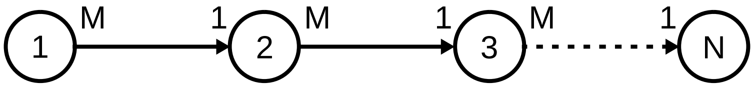


Figure 3. A family of SDF graphs in which the number of DAG nodes increases exponentially with respect to the number of nodes in the SDF graph.

Another example can be found in [17], where a five-node SDF representation of a compact-disc to digital audio tape sample rate conversion system expands to a DAG that contains over 600 nodes. This growth is undesirable, especially considering that known optimal multiprocessor scheduling algorithms under precedence constraints have complexity that is exponential in the number of nodes in the DAG [11]. Most uniprocessor SDF schedulers, on the other hand, do not require a DAG to be generated for scheduling purposes.

2.3 Clustering

To limit the explosion of nodes when translating an SDF graph into a DAG graph, we apply *clustering* of connected subgraphs into larger grain *composite* nodes. The composite nodes will then be scheduled with one of the available uniprocessor schedulers. We cluster the nodes in a manner that simplifies the DAG without hiding much exploitable parallelism.

The concept of clustering in an SDF graph is illustrated in figures 4(a-b). Here, the graph on the right is obtained by clustering the subset of nodes $\{B, C\}$. The “D” on arc (B, C) specifies a unit delay ($\delta_{(B, C)} = 1$).

Formally, clustering a subset Z of nodes in an SDF graph $G = (V, E)$ into a single composite node Ω produces a new SDF graph, denoted $cluster(Z, G)$ that consists of the set of nodes $(V - Z + \{\Omega\})$. The set of arcs E' in $cluster(Z, G)$ can be expressed as

$$E' = E - \{e \mid ((src(e) \in Z) \textbf{ or } (snk(e) \in Z))\} + E^*,$$

where E^* is a “modification” of the set of arcs that connect actors in Z to actors outside of Z. If for each $e \in E$ such that $src(e) \in Z$ and $snk(e) \notin Z$, we define $\tilde{e}$ by

$$src(\tilde{e}) = \Omega,\ snk(\tilde{e}) = snk(e),\ \kappa_{\tilde{e}} = \kappa_e,\ \delta_{\tilde{e}} = \delta_e, \text{ and } \rho_{\tilde{e}} = \frac{\mathbf{q}(src(e))}{\chi(Z)}\rho_e;$$

and similarly, for each $e \in E$ such that $snk(e) \in Z$ and $src(e) \notin Z$, we define $\tilde{e}$ by

$$snk(\tilde{e}) = \Omega,\ src(\tilde{e}) = src(e),\ \rho_{\tilde{e}} = \rho_e,\ \delta_{\tilde{e}} = \delta_e, \text{ and } \kappa_{\tilde{e}} = \frac{\mathbf{q}(snk(e))}{\chi(Z)}\kappa_e,$$

then we can specify E^* by

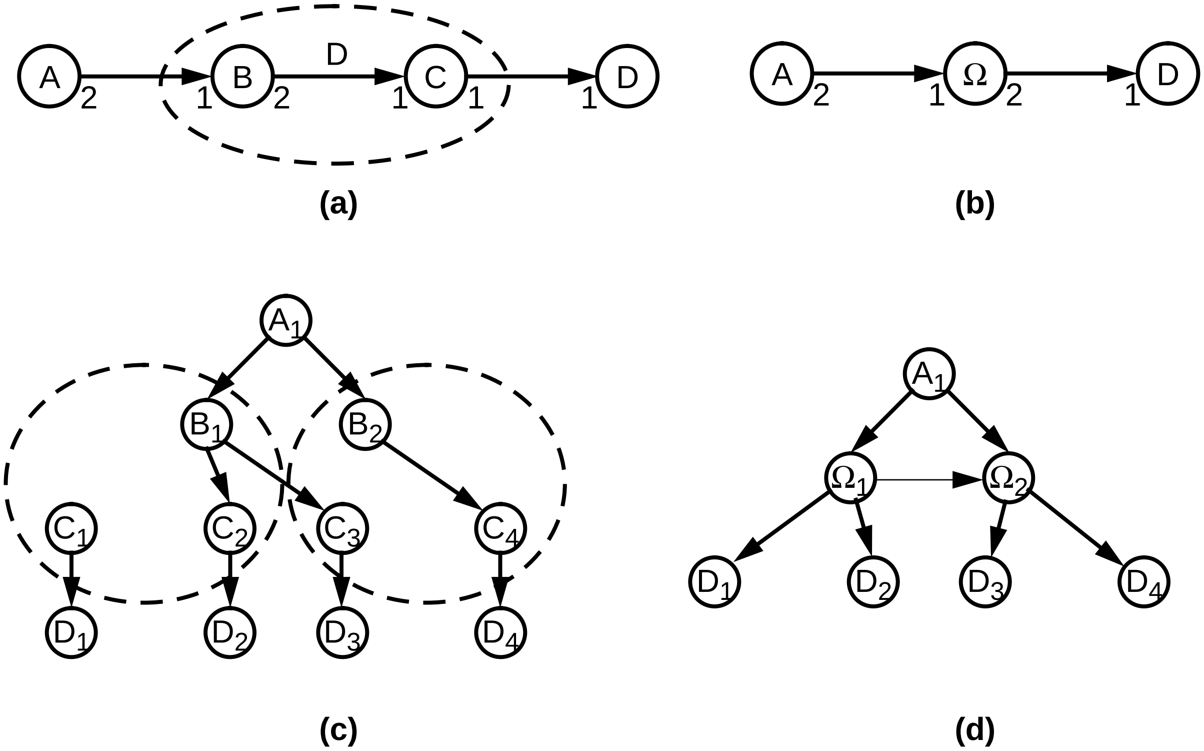
$$E^* = \{\tilde{e} \mid ((src(e) \in Z) \textbf{ and } (snk(e) \notin Z)) \textbf{ or } ((snk(e) \in Z) \textbf{ and } (src(e) \notin Z))\}.$$

This precise interpretation of clustering in SDF graphs was introduced in [23].

The precedence graph of $\mathit{cluster}(Z, G)$ can be derived from the precedence graph of G by consolidating each subset of invocations

$$\{z_i | ((z \in Z) \textbf{ and } (k-1)\frac{\mathbf{q}(z)}{\chi(Z)} < i \le k\frac{\mathbf{q}(z)}{\chi(Z)})\} \qquad (3)$$

into the single precedence graph node Ω_k, for $k = 1, 2, \ldots, \chi(Z)$. This is illustrated in figures 4(c-d) for the clustering operation shown in figures 4(a-b).



(a)

(b)

(c)

(d)

Figure 4. An illustration of clustering. Parts (a) and (b) illustrate how a clustering operation transforms an SDF graph, and parts (c) and (d) illustrate the corresponding transformation on the precedence graph.

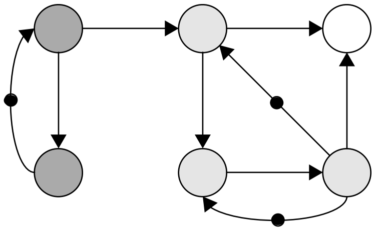


Figure 5. Nodes with the same pattern belong to the same strongly connected component. Note that the clear node is a trivial strongly connected component.

2.4 Strongly connected components

A subgraph, $G' = (V', E')$, is a **nontrivial strongly connected component** of a directed graph $G = (V, E)$ if:

- $V' \subseteq V$ and $E' \subseteq E$;
- $\forall v_1, v_2 \in V'$ there is a path directed from v_1 to v_2 and there is a path directed from v_2 to v_1; and
- $|V'| > 1$ (G' contains more than one node).

Figure 5 shows a graph with the strongly connected components marked. Tarjan in [19] developed an efficient algorithm to find strongly connected components in linear time with respect to the number of arcs and nodes in the SDF system

If (V', E') is a nontrivial strongly connected component, we may also say that V' is a nontrivial strongly connected component.

3. Multiprocessor DAG scheduling

DAG multiprocessor schedulers that minimize the interprocessor communication (IPC) costs typically have two distinct scheduling phases [12-15]:

1. A clustering phase to minimize IPC costs, by improving the parallel time at each clustering step.
2. A processor assignment phase, to map and schedule the clusters onto the available processors.

During initialization each DAG node is mapped onto a separate processor. Then, the nodes are labeled with the computation times and the arcs are labeled with the IPC costs. As groups of nodes are clustered together (mapped onto the same processor), the corresponding arc costs are set to zero. The *parallel time* (PT) can then be defined as the length of the longest path in the graph. Figure 6 shows an initial labeled DAG and the result of one clustering step.

It is important to note that each resultant cluster is mapped onto a single processor. This observation motivates the modification of parallel time minimization clustering heuristics for use on the SDF graph. By clustering the SDF graph we also have the opportunity to use specialized uniprocessor SDF schedulers, which can optimize for such parameters as code size, buffer memory, and context switch overhead [18, 23, 26, 27, 28, 29].

4. Some properties of precedence graphs

In this section, we introduce several properties of precedence graphs. In the following section, we will apply these properties to develop an efficient test for whether or not a given clustering operation introduces deadlock.

Lemma 1: Suppose that G is an SDF graph, α is an arc in G, and Q is a positive integer such that $\rho_\alpha = k_1 Q \kappa_\alpha$, where k_1 is a positive integer, and $\delta_\alpha = k_2 Q \kappa_\alpha$ where k_2 is a nonnega-

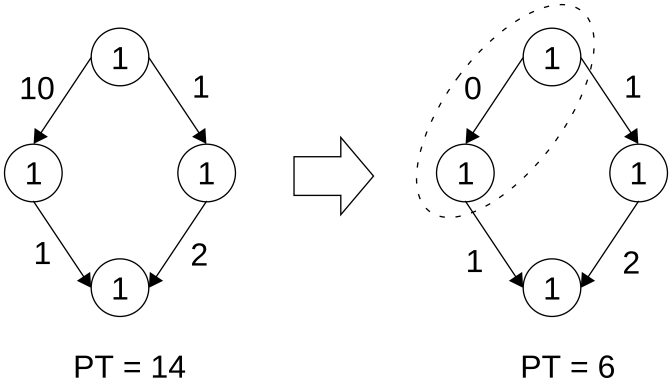


Figure 6. The graph on the left shows an initially labelled DAG. The graph on the right shows the result after one clustering step.

tive integer. Suppose that we divide the invocations of $snk(\alpha)$ into groups of Q:

$$I_j = \{snk(\alpha)_{(j-1)Q+1}, snk(\alpha)_{(j-1)Q+2}, ..., snk(\alpha)_{jQ}\}, j = 1, 2, \ldots, k_1\mathbf{q}(src(\alpha)) \qquad (4)$$

Then for each j, there is at most one invocation y of $src(\alpha)$ such that an arc in $PRG(G)$ exists that is directed from $src(\alpha)_y$ to at least one member of I_j. That is, no more than one invocation of $src(\alpha)$ has a precedence graph output arc directed to a member of I_j. Furthermore, if $src(\alpha)_y$ has an output (precedence graph) arc directed to some member of I_j, then $src(\alpha)_y$ has output arcs directed to all members of I_j

As an example of lemma 1, consider figure 7. Let $\alpha = (A, B)$, and observe that for this choice of α, the assumptions of lemma 1 are satisfied with $Q = 2$, $k_1 = 3$ and $k_2 = 2$. Each group of invocations I_j is shown by one of the dashed ovals that encircles a group of $Q = 2$ adjacent invocations of B. We see that the members of I_1 and I_2 do not have any input precedence edges from any of the invocations of A; the members of I_3, I_4, I_5 have input precedence edges from exactly one invocation — invocation A_1 — of A; and similarly, the members of I_6 have input precedence edges only from A_2. Thus, the example of figure 7 is consistent with lemma 1.

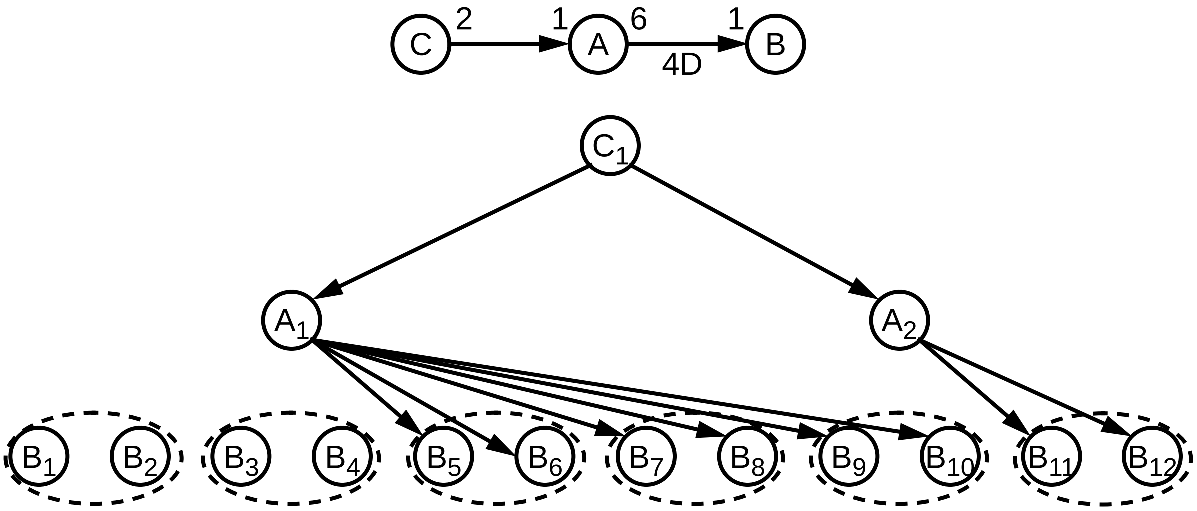


Figure 7. An illustration of lemma 1.

Proof of lemma 1: Let $A = src(\alpha)$ and $B = snk(\alpha)$, and recall from (1) and (2) that an arc in $PRG(G)$ exists from A_m to B_n if and only if one of the following two conditions hold:

$$(m-1)\rho_\alpha + \delta_\alpha \le (n-1)\kappa_\alpha < m\rho_\alpha + \delta_\alpha \text{, or} \tag{5}$$

$$(n-1)\kappa_\alpha < (m-1)\rho_\alpha + \delta_\alpha < n\kappa_\alpha . \tag{6}$$

Now, from the given assumptions, (6) becomes

$$(n-1)\kappa_\alpha < (m-1)k_1Q\kappa_\alpha + k_2Q\kappa_\alpha < n\kappa_\alpha , \tag{7}$$

or equivalently,

$$(n-1) < (m-1)k_1Q + k_2Q < n . \tag{8}$$

Since there are no integers "between" $(n-1)$ and n, clearly (8) cannot hold for any pair of positive integers (m, n), and thus, we conclude that a precedence graph arc exists from A_m to B_n if and only if (5) holds.

Now from the given assumptions, (5) is equivalent to

$$(m-1)k_1Q\kappa_\alpha + k_2Q\kappa_\alpha \le (n-1)\kappa_\alpha < mk_1Q\kappa_\alpha + k_2Q\kappa_\alpha , \tag{9}$$

or

$$r_mQ \le (n-1) < (r_m + k_1)Q \text{, where } r_m = (m-1)k_1 + k_2 . \tag{10}$$

Clearly, (10) is satisfied if and only if

$$n \in \{(r_mQ+1), (r_mQ+2), \ldots, (r_mQ + k_1Q)\} , \tag{11}$$

and from the definition of the sequence $I_1, I_2, \ldots$, we conclude that

there is a precedence graph edge from A_m to B_n iff for some $k \in \{1, 2, \ldots, k_1\}$, $B_n \in I_{r_m + k}$.(12)

Thus, if there is a precedence graph edge from A_m to B_n, and $B_n \in I_j$, then there is a precedence graph edge from A_m to all members of I_j.

Now, from the definition of r_m in (10), we see that if μ and μ' are distinct positive integers, and λ, λ' are members of $\{1, 2, \ldots, k_1\}$, then $(r_{\mu} + \lambda) \neq (r_{\mu'} + \lambda')$. Thus, it follows from (12), that for a given I_j, there is at most one invocation of A that has a precedence graph output edge directed into I_j (directed to some member of I_j).❑

The following lemma is analogous to lemma 1, with the roles of the source and sink nodes interchanged.

Lemma 2: Suppose that G is an SDF graph, α is an arc in G, and Q is a positive integer such that $\kappa_{\alpha} = k_1 Q \rho_{\alpha}$, where k_1 is a positive integer, and $\delta_{\alpha} = k_2 Q \rho_{\alpha}$ where k_2 is a nonnegative integer. Suppose that we divide the invocations of $src(\alpha)$ into groups of Q:

$$I_j = \{src(\alpha)_{(j-1)Q+1}, src(\alpha)_{(j-1)Q+2}, \ldots, src(\alpha)_{jQ}\}, j = 1, 2, \ldots, k_1 \mathbf{q}(snk(\alpha)) .$$

Then for each j, there is at most one invocation y of $snk(\alpha)$ such that an arc in $PRG(G)$ exists that is directed to $snk(\alpha)_y$ from a member of I_j. Furthermore, if $snk(\alpha)_y$ has an input arc directed from some member of I_j, then $snk(\alpha)_y$ has input arcs directed from all members of I_j.

Proof of lemma 2: Follows by symmetry from lemma 1. ❑

Lemma 3: Suppose that C is a connected, consistent SDF graph that contains at least two nodes, and whose edges form a simple cycle. Suppose that α is an arc in C, β is the arc whose source node is $snk(\alpha)$, and Q is a positive integer such that

$$\delta_{\alpha} = kQ\kappa_{\alpha}, \text{ where } k \in (Z^{+} \cup \{0\}) , \tag{13}$$

and

$$\rho_{\alpha} = jQ\kappa_{\alpha} \text{ where } j \in Z^{+} . \tag{14}$$

Define α' and β', respectively, by

$$src(\alpha') = src(\alpha) , snk(\alpha') = snk(\alpha) , \delta_{\alpha'} = \delta_{\alpha}, \rho_{\alpha'} = \rho_{\alpha}, \kappa_{\alpha'} = Q\kappa_{\alpha}, \text{ and} \tag{15}$$

$$src(\beta') = src(\beta)\ ,\ snk(\beta') = snk(\beta)\ ,\ \delta_{\beta'} = \delta_{\beta},\ \rho_{\beta'} = Q\rho_{\beta},\ \kappa_{\beta'} = \kappa_{\beta}\,. \tag{16}$$

Suppose that C' is the SDF graph that results from replacing α, β with α', β' in C, respectively. Then C' is consistent.

As an example of lemma 3, consider figure 8 with $\alpha = (A, B)$, $\beta = (B, C)$ and

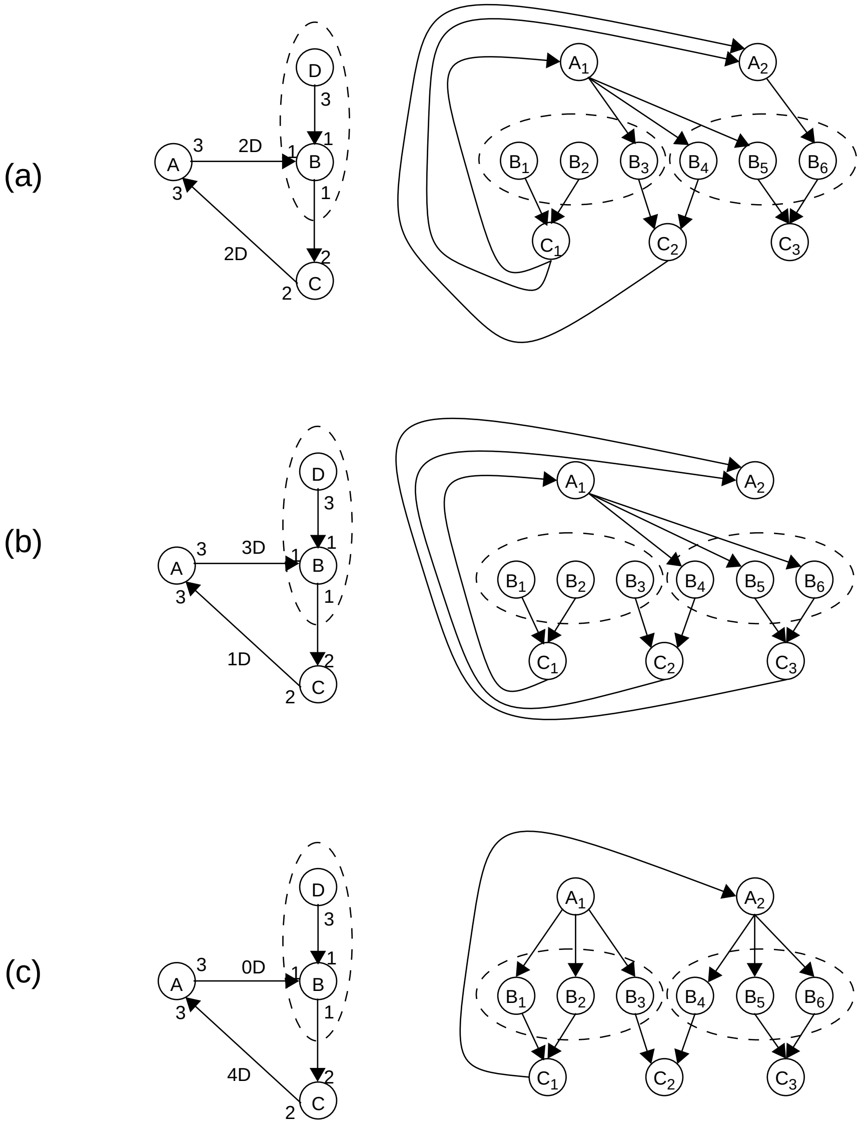


Figure 8. An illustration of lemma 3

$Q = 3$, which corresponds to the clustering operation shown. Clearly (14) is satisfied with $j = 1$. However $\delta_\alpha \neq kQ\kappa_\alpha$ in figure 8(a), while $\delta_\alpha = kQ\kappa_\alpha$ in figures 8(b) and 8(c). Accordingly, the clustering operation introduces a precedence graph cycle (deadlock) in figure 8(a), while (as guaranteed by lemma 3) the clustering operation leaves the cycle consistent in figures 8(b) and 8(c).

Proof of lemma 3: It is easily seen that the vector $\mathbf{b}$ defined by

$$\mathbf{b}(x) = \left(\begin{array}{l} \mathbf{q}_C(x) \text{ if } x \neq snk(\alpha) \\ \dfrac{\mathbf{q}_C(x)}{Q} \text{ if } x = snk(\alpha) \end{array}\right) \tag{17}$$

satisfies the balance equations for C'. Thus, a repetitions vector exists for C', and it remains only to be shown that C' is not deadlocked. We show this by contraposition.

Suppose that C' is deadlocked. Then from Fact 1, there exists a cycle in $PRG(C')$ that traverses some invocation, say invocation a, of $snk(\alpha)$. From lemma 1,

there is exactly one invocation, say invocation b, of $src(\alpha)$ such that in $PRG(C')$, $src(\alpha)_b$ is connected to $snk(\alpha)_a$ by an arc. (18)

Now observe that

$PRG(C')$ can be obtained by consolidating $\{snk(\alpha)_1, snk(\alpha)_2, ..., snk(\alpha)_Q\}$, $\{snk(\alpha)_{1+Q}, snk(\alpha)_{2+Q}, ..., snk(\alpha)_{2Q}\}$, … in $PRG(C)$. (19)

From lemma 1 we know that

in $PRG(C)$, there is an arc directed from $src(\alpha)_b$ to each member of $\{snk(\alpha)_{aQ}, snk(\alpha)_{aQ-1}, ..., snk(\alpha)_{aQ-(Q-1)}\}$. (20)

Now since there is a cycle in $PRG(C')$ that traverses $snk(\alpha)_a$, we have from (18) that there is a path in $PRG(C')$ directed from $snk(\alpha)_a$ to $src(\alpha)_b$. Thus, from (19), it follows that in $PRG(C)$, there is a path directed from some member $snk(\alpha)_c$ of $\{snk(\alpha)_{aQ}, snk(\alpha)_{aQ-1}, ..., snk(\alpha)_{aQ-(Q-1)}\}$ to $src(\alpha)_b$, and from (20), we conclude that in $PRG(C)$, there is a cycle that traverses $snk(\alpha)_c$. This contradicts the assumption that C is consistent.❑

Lemma 4: Assume the same hypotheses as lemma 3, except replace (13) and (14) with

$$\delta_\beta = kQ\rho_\beta \text{ where } k \in (Z^+ \cup \{0\}) \text{, and} \tag{21}$$

$$\kappa_\beta = jQ\rho_\beta \text{ where } j \in Z^+ . \tag{22}$$

Then the SDF graph that results from replacing α, β with α', β' in C is consistent.

Proof: Follows by symmetry from lemma 3. ❑

Lemma 5: Suppose that C is a consistent, connected SDF graph that contains at least three nodes, and whose edges form a simple cycle $v_1 \to v_2 \to \ldots \to v_N = v_1$; let e_p denote the arc that has v_p as its source node, for each p; and suppose that for some $j \in \{1, 2, \ldots, N-1\}$, we have

$$\mathbf{q}(v_{j+1}) = k\mathbf{q}(v_j) \text{, where } k \in Z^+ \text{, and} \tag{23}$$

$$\delta_{e_j} = 0 , \tag{24}$$

where $\mathbf{q}$ is the repetitions vector of C. Then $cluster(\{v_j, v_{j+1}\}, C)$ is consistent.

Proof: A repetitions vector for the clustered graph can always be derived from the repetitions vector of the SDF graph [16]. Thus, clustering always preserves the existence of a repetitions vector. It remains to be shown that clustering $\{v_j, v_{j+1}\}$ does not introduce deadlock. We show this by contraposition.

Let $C' = cluster(\{v_j, v_{j+1}\}, C)$, and suppose that C' is deadlocked. Let Ω denote the clustered node in C'. Then from Fact 1, there exists a cycle in $PRG(C')$ that traverses some invocation, say invocation x, of Ω.

From (3), $PRG(C')$ can be obtained by clustering each of the subsets[1]

[1] Here, $(v_g)_h$ represents the hth invocation of node v_g.

$$\{ (v_j)_1, (v_{j+1})_1, (v_{j+1})_2, \ldots, (v_{j+1})_k \}, \{ (v_j)_2, (v_{j+1})_{k+1}, (v_{j+1})_{k+2}, \ldots, (v_{j+1})_{2k} \}, \ldots$$

in $PRG(C)$.

Thus, in $PRG(C)$,

there is a path from $(v_{j+1})_{mk+i}$ to $(v_j)_{m+1}$, for some i, m, where $0 \le m < \mathbf{q}(v_j)$, and $1 \le i \le k$.(25)

Now from (23), (24) it is easily seen that there are arcs in $PRG(C)$ from $(v_j)_{m+1}$ to every member of $\{ (v_{j+1})_{mk+1}, (v_{j+1})_{mk+2}, \ldots, (v_{j+1})_{mk+k} \}$, and it follows from (25) that there is a cycle in $PRG(C)$ containing the arc $((v_j)_{m+1}, (v_{j+1})_{mk+i})$. This contradicts the assumption that C is consistent. ❑

Lemma 6: Assume the same hypotheses as lemma 5, except replace (23) with

$$\mathbf{q}(v_j) = k\mathbf{q}(v_{j+1}) . \tag{26}$$

Then $cluster(\{v_j, v_{j+1}\}, C)$ is consistent.

Proof: Follows by symmetry from lemma 5.❑

5. The SDF composition theorem

Unfortunately, SDF lacks the composition property. That is, if we cluster two arbitrary SDF nodes, we may introduce deadlock into the SDF graph. To compose two SDF nodes into a valid SDF cluster, we have developed the SDF composition theorem. This theorem presents four conditions that guarantee that deadlock is not introduced into the clustered SDF graph.

Thus, the SDF composition theorem provides a sufficient condition that a clustering operation does not introduce deadlock. Currently there is no known *exact* condition (both necessary and sufficient) that can be evaluated in polynomial time with respect to the number of nodes in the SDF graph. In [30], Karp and Miller give an exact algorithm for determining whether or not an arbitrary computation graph deadlocks. When this algorithm is applied to an SDF graph that has a repetitions vector, the following steps are effectively carried out for each strongly connected component:

(a) For each simple cycle C, a weighted sum of the arc delays is compared against the inner product of two vectors[1]. The weights of the delays and the two vectors are functions of the production and consumption parameters on the arcs in C. If for each simple cycle, the weighted sum of the delays exceeds the corresponding inner product, then one can conclude that the given SDF graph is not deadlocked.

(b) Otherwise, a second pass over the simple cycles is initiated. An iterative method is applied to each cycle that is examined. For an SDF graph that has a repetitions vector, this iterative method is equivalent (in terms of complexity) to first computing the repetitions vector $\mathbf{q}$ of the cycle, and then starting at any node $snk(\alpha)$ of the cycle that satisfies $delay(\alpha) \geq \kappa_{\alpha}$; repeatedly traversing the cycle by moving from the current node to its successor in the cycle at each traversal step; and simulating (updating the number of tokens on the appropriate arcs) exactly $\lfloor b(\alpha') / \kappa_{\alpha'} \rfloor$ invocations of each node when it is visited, where α' is the input arc to the node in the cycle, and $b(\alpha')$ is the number of tokens on α'. The test for a given cycle terminates when either the simulated system deadlocks, in which case we know that the overall SDF graph is deadlocked, or a state is reached where some node x has been executed $\mathbf{q}(x)$ or more times, in which case we know that the current cycle is not deadlocked.

Step (b) entails two levels of running time explosion since (1) the number of simple cycles in a directed graph is not polynomially bounded by the number of nodes, and (2) there is no polynomial bound (in the number of SDF nodes) on the number of node visits that occur in the simulation of a given cycle.

We justify claim (2) with the aid of figure 9. Assuming that $n > 1$, it can be shown that the graph in figure 9 is deadlocked if an only if $\delta \geq 2(n-1)$. Now since the production parameter of each arc is equal to the consumption parameter on the other arc, it follows that throughout any execution of this graph, the total number tokens queued on both arcs remains constant. Thus, the number of tokens residing on edge (A, B) is always less than or equal to δ.

[1] In [30], this test is expressed in a form that is suitable for general computation graphs. This form involves the comparison of the zero vector against the product of a square matrix with a column vector. However for the special case in which the cycle in question is an SDF graph for which a repetitions vector exists (unity *gain*), it is easily verified that the rows in the matrix are constant multiples of one another, and thus, all but one (any one) of them can be discarded from the test.

Now suppose that $\delta = 2(n-1)$, which implies that $\delta < 2\kappa_{(A, B)}$. Then each time node B is visited during the simulation defined in step (b) above, exactly one invocation of B is executed, and thus node B is visited a total of $\mathbf{q}(B)$ times in the simulation. Since $\mathbf{q}(B) = (n-1)$, we see that the time required to complete the simulation increases without bound as the parameter n tends to infinity. Thus, we see that for arbitrary SDF graphs, there is no bound on the completion time, let alone a polynomial bound, that is a function of the number of SDF nodes alone.

An alternative exact condition can be evaluated by applying the loop scheduling algorithm of [18], and then using a class-S scheduling algorithm [7] to process the resulting *tightly interdependent components*. Families of graphs exist for which the technique of [18] requires $\Omega(m^2)$ time, where m is the number of SDF nodes; one such family is illustrated in figure 10. For this family of graphs, it can be shown that in each application, the *subindependence partitioning* step in [18] separates out the left-most node of the strongly connected component that it is applied to, thereby reducing the size of the strongly connected component by exactly one node. Thus, n subindependent partitions must be constructed. Furthermore each application of subindependence partitioning involves an application of Tarjan's algorithm for finding strongly connected components [19], which requires $\Omega(n')$ time, where n' is number of nodes in the SDF subgraph that it is applied to. From these observations, it is easily seen that the technique of [18] requires $\Omega(n^2)$ time for figure 10.

For general SDF graphs, the net running time attributed to the use of a class S scheduling

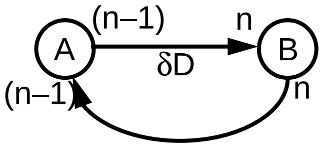


Figure 9. A family of SDF graphs that is used to show that there is no upper bound on the number of steps required by Karp and Miller's iteration scheme that is a function of the number of SDF nodes.

algorithm in the technique of [18] is

$$\Omega\left(\sum_{x\in T}\mathbf{q}(x)\right)\text{ per clusterization test,}$$

where T is the set of SDF nodes that are contained in the *tightly interdependent components*. It has been observed that T is usually empty in practice. However, the potential for $\Omega(m^2)$ behavior evidenced by the example in figure 10 proves prohibitively expensive for hierarchical scheduling since in such cases, the $\Omega(m^2)$ cost must be incurred at least once for *each* clusterization step, and there may be up to e total clusterization steps, where e is the number of edges in the original SDF graph.

A third exact condition is described by Bilsen, Engels, Lauwereins, and Peperstraete [31] for a computational model called *cyclo-static dataflow*, which is an extension of SDF. As in the approach of Karp and Miller, evaluating this condition for a general SDF graph requires examining each simple cycle separately. For a cycle that is not deadlocked, the tightest known upper bound, time complexity expression for the test applied to each cycle is

$$O(n\times min(\{\mathbf{q}(x)|x\text{ is traversed by }C\}))\text{ ,}$$

where $\mathbf{q}$ is the repetitions vector of C and n is the number of nodes traversed by C. It is easily shown that there is no upper bound on the minimum repetitions vector component that is polynomial in n, and thus, the approach of [31] exhibits the potential for a similar form of “two-level

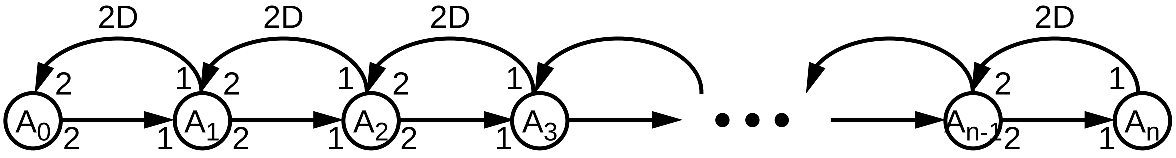


Figure 10. A family of SDF graphs for which the running time required by the loop scheduling algorithm of [18] is $\Omega((\text{number of SDF nodes})^2)$.

running time explosion" as is present in the approach of Karp and Miller.

Two sufficient — but not necessary — clustering conditions that were developed previously — the *merge pass* conditions of Buck [26], and How's clustering of *uniform frequency* subgraphs [27] — are too restrictive for our purposes since these conditions require that the nodes involved in a given SDF cluster must have identical repetition counts (components of $\mathbf{q}$). Thus, they cannot be used to reduce the explosion in the size of the DAG that arises from multirate subgraphs such as the example in figure 3. The clustering techniques described in [18, 25, 28] all permit clustering across changes in repetition count; however, the first of these maintains the cluster hierarchy on the DAG, the second technique is restricted to acyclic graphs, and the third technique was designed under the restriction that the existence of a single appearance schedule must be preserved. Thus, these three approaches are not in alignment with our primary objectives in clustering for hierarchical multiprocessor scheduling, which are (a) to avoid constructing a DAG until the entire cluster hierarchy is constructed; (b) to limit the size of the DAG resulting from the final cluster hierarchy as much as possible; and (c) to handle arbitrary topologies, including graphs that contain cycles. After our clustering pass has constructed its decomposition of the input graph, individual components of the resulting hierarchy may subsequently be processed by any of the alternative clustering techniques described above; however, our objectives in constructing the initial hierarchy require a cluster selection process that is significantly more general (less restrictive) than the previous approaches.

The following theorem, which we call the *SDF composition theorem*, establishes four clustering criteria that together provide a sufficient condition that a given clustering operation involving two adjacent nodes does not produce deadlock. The first three conditions prevent the introduction of cycles into the precedence graph. The last condition prevents the introduction of new cycles into both the SDF graph and the precedence graph.

Theorem: Suppose that *G* is a consistent, connected SDF graph, and (x, y) is an ordered pair of distinct, adjacent nodes in *G*. Then $\mathit{cluster}(\{x, y\}, G)$, the graph that results from clustering $\{x, y\}$ into a single node Ω, is consistent if the following four conditions all hold.

1. **Precedence shift condition** A: If x is in a nontrivial strongly connected component C, then:

for each $\alpha \in \{\alpha' | (snk(\alpha') = x) \textbf{ and } (src(\alpha') \in C) \textbf{ and } (src(\alpha') \notin \{x, y\})\}$,

there exists a positive integer k_1 and a nonnegative integer k_2 such that

$$\rho_\alpha = k_1 Q(x, y)\kappa_\alpha \text{ and } \delta_\alpha = k_2 Q(x, y)\kappa_\alpha, \underline{\mathit{or}} \tag{27}$$

for each $\alpha \in \{\alpha' | (src(\alpha') = x) \textbf{ and } (snk(\alpha') \in C) \textbf{ and } (snk(\alpha') \notin \{x, y\})\}$,

there exists a positive integer k_1 and a nonnegative integer k_2 such that

$$\kappa_\alpha = k_1 Q(x, y)\rho_\alpha \text{ and } \delta_\alpha = k_2 Q(x, y)\rho_\alpha. \tag{28}$$

2. **Precedence shift condition B**: If y is in a nontrivial strongly connected component C, then:

for each $\alpha \in \{\alpha' | (snk(\alpha') = y) \textbf{ and } (src(\alpha') \in C) \textbf{ and } (src(\alpha') \notin \{x, y\})\}$,

there exists a positive integer k_1 and a nonnegative integer k_2 such that

$$\rho_\alpha = k_1 Q(x, y)\kappa_\alpha \text{ and } \delta_\alpha = k_2 Q(x, y)\kappa_\alpha, \underline{\mathit{or}} \tag{29}$$

for each $\alpha \in \{\alpha' | (src(\alpha') = y) \textbf{ and } (snk(\alpha') \in C) \textbf{ and } (snk(\alpha') \notin \{x, y\})\}$,

there exists a positive integer k_1 and a nonnegative integer k_2 such that

$$\kappa_\alpha = k_1 Q(x, y)\rho_\alpha \text{ and } \delta_\alpha = k_2 Q(x, y)\rho_\alpha. \tag{30}$$

3. **Hidden delay condition**: If x and y are in the same strongly connected component, then (a) at least one arc from x to y has zero delay, <u>*and*</u> (b) for some positive integer k, $\mathbf{q}(x) = k\mathbf{q}(y)$ or $\mathbf{q}(y) = k\mathbf{q}(x)$.
4. **Cycle introduction condition**: There is no simple path from x to y that contains more than one arc.

Figure 11 illustrates graphs that violate the conditions of the SDF composition theorem.

Note that the conditions given in the SDF composition theorem may be satisfied for the ordered pair (y, x), even though they are not satisfied for (x, y). Thus, in general, both orderings should be tried before ruling out a clustering operation.

Proof of the SDF composition theorem: Let $G' = cluster(\{x, y\}, G)$. As mentioned in the proof of lemma 5, clustering does not cancel the existence of a repetitions vector, so we need only show that G' is not deadlocked. As in lemmas 3 and 5, we prove this by contraposition.

Suppose G' is deadlocked.

From condition 4, no new cycles are *introduced* by the clustering operation. That is, for every cycle $C = a_1 \to a_2 \to \ldots \to a_{m-1} \to a_m = \Omega \to a_{m+1} \to \ldots \to a_N = a_1$ in G' that contains Ω, there is a unique corresponding cycle in G, and this cycle has one of the following forms:

(a) $a_1 \to a_2 \to \ldots\ x\ \ldots \to a_N = a_1$ (same as C, but with Ω replaced by x);

(b) $a_1 \to a_2 \to \ldots\ y\ \ldots \to a_N = a_1$ (same as C, but with Ω replaced by y);

(c) $a_1 \to a_2 \to \ldots \to x \to y \to \ldots \to a_N = a_1$ (same as C, but with Ω replaced by $x \to y$).

For example, figure 12(a) corresponds to scenario (a) above; figure 12(b) corresponds to

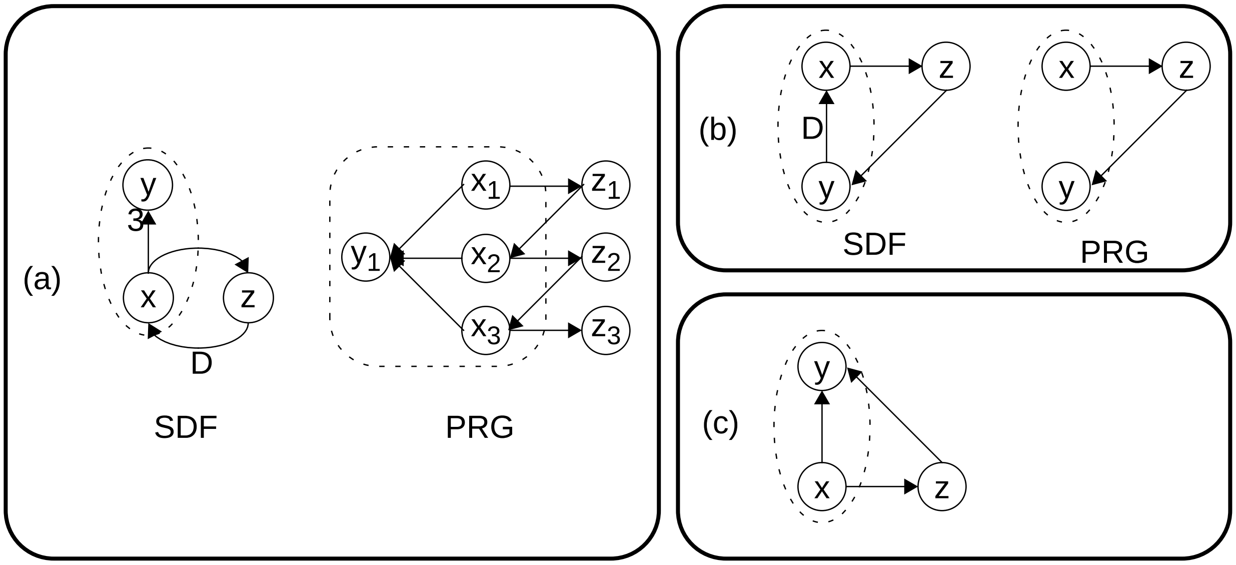


Figure 11. Systems that violate the SDF composition theorem. System (a) violates precedence shift condition A; system (b) violates the hidden delay condition; and system (c) violates the cycle introduction condition. Notice that the clusterings in systems (a) and (b) introduce cycles in the precedence graph, while the clustering in system (c) introduces cycles in both the SDF graph and its corresponding precedence graph. Here, in the SDF graphs, $\rho_\alpha = 1$ and $\kappa_\alpha = 1$ for all arcs α, with the exception that $\kappa_{(x, y)} = 3$ in (a).

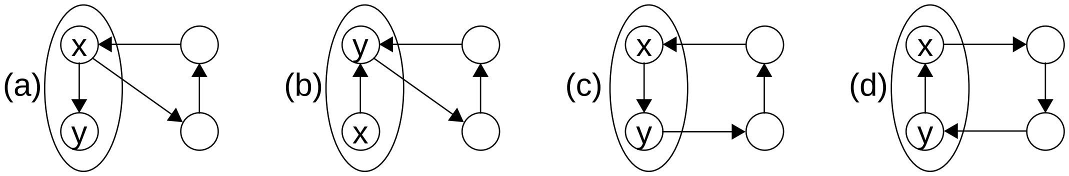


Figure 12. Examples used in the proof of the SDF composition theorem.

scenario (b); figure 12(c) corresponds to scenario (c); and the situation in figure 12(d) cannot arise due to condition 4.

Since G' is deadlocked, at least one of the cycles that traverse Ω must be deadlocked. Let C^* be a deadlocked cycle in G' that traverses Ω. Suppose that C^* is of form (a). Then from lemmas 3 and 4, and condition 1, it follows that the corresponding cycle in G must be deadlocked, and thus that G must be deadlocked. Similarly, if C^* is of form (b), then it follows from lemmas 3 and 4, and condition 2 that the G must be deadlocked. Finally, if C^* is of form (c), then lemma 5, lemma 6 and condition 3 guarantee that G is deadlocked. Thus the assumption that G' is deadlocked implies that G is deadlocked. But this contradicts our assumption that G is consistent.❑

Clearly, both of the precedence shift conditions and the hidden delay condition can be checked exactly in an efficient manner. A number of exact and approximate (conservative) tests are possible for the cycle introduction condition. Currently, we are trying to determine which test for the cycle introduction condition yields the best trade-off between accuracy and efficiency for practical applications.

6. Clustering techniques

In this section, we review our clustering techniques for SDF graphs. There are currently four clustering techniques: user specified, resource constraint limited, *well-ordered* URC subgraphs (an acyclic graph is well ordered if it has only one topological sort), and the parallel time reduction heuristic.

The first clustering technique is by far the simplest: we allow the user to specify clusters that will be mapped onto a single processor. This clustering technique empowers the user with fundamental scheduling decisions. A potential problem is that the user can introduce artificial deadlock. However, this error is easily caught at compile time [20]. We have implemented this technique in Ptolemy, where it has enabled the development of multiprocessor applications that have previously been impossible to synthesize using other SDF multiprocessing techniques. When we automatically cluster subgraphs, we must ensure that the constructed clusters do not introduce artificial deadlock. We can accomplish this by using the SDF composition theorem.

The next clustering technique takes into account resource constraints. When mapping SDF graphs onto heterogeneous processors, a group of connected nodes may be required to be mapped onto a particular processor. Here, we are free to cluster these SDF subgraphs as long as we do not introduce artificial deadlock.

The third clustering technique groups the nodes in a well-ordered, URC SDF subgraph where the nodes do not have internal state (or equivalently, have self loop arcs). One source node is connected to all the input arcs and one destination node is connected to all the output arcs. Thus, when clustering well ordered SDF subgraphs, we do not group over *branch* or *merge* SDF nodes (nodes that have multiple sources or destinations), where functional parallelism is exposed. This clustering does not hide any of the available parallelism that will be exposed in the final DAG. An example is shown in figure 13.

Finally, the last clustering technique is based on an adaptation of Sarkar's multiprocessor DAG scheduling heuristic to SDF graphs [14]. Sarkar's algorithm is described below, and an example is shown in figure 14:

1. Sort arcs of a DAG in descending order of arc costs

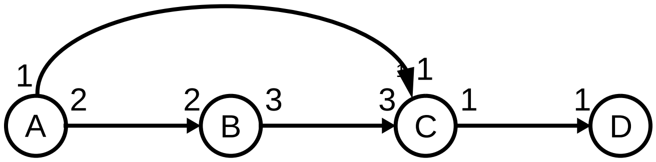


Figure 13. A well-ordered, URC

2. Zero the arc with the maximum weight if the parallel time does not increase
3. Repeat step 2 for all arcs

To apply Sarkar's algorithm to SDF graph, we must first construct an *acyclic version* of the SDF graph. We construct an acyclic version of our SDF graph by:

1. Removing all arcs α that satisfy $\delta_\alpha \geq \kappa_\alpha \mathbf{q}(snk(\alpha))$.
2. Clustering all remaining strongly connected components in the modified SDF graph.

7. Hierarchical scheduling algorithm

We are now ready to present the proposed hierarchical scheduling algorithm. This algorithm will be implemented in the months to come within the Ptolemy project.

7.1 Initialization

1. Cluster nodes that are on SDF well-ordered URC subgraphs without internal state [1].
2. Cluster nodes that share resource constraints which satisfy the SDF composition theorem.
3. Compute the repetitions vector, $\mathbf{q}$, $O(|V| + |E|)$.
4. Construct the *acyclic SDF graph.*
5. Compute the total IPC cost for each arc on the *acyclic SDF graph.*

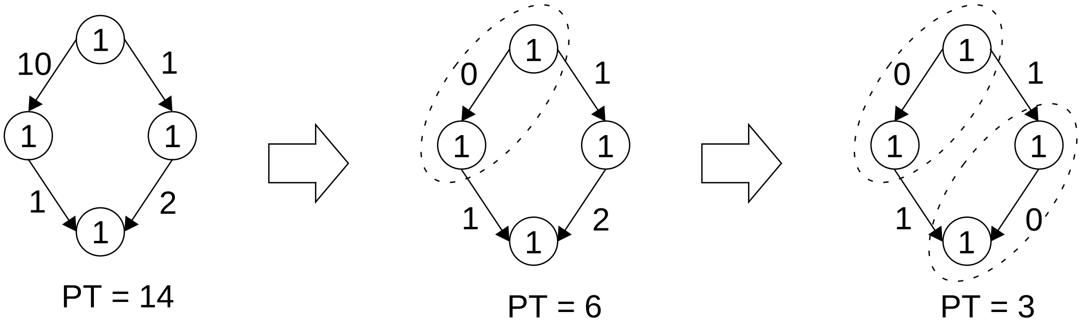


Figure 14. An illustration of Sarkar's algorithm.

7.2 Main loop

1. Apply one step of Sarkar's multiprocessor clustering heuristic on the *acyclic SDF graph.*
2. Using the SDF composition theorem, test the resulting cluster candidate to make sure it does not introduce deadlock.
3. If the candidate does not introduce deadlock, then perform the corresponding clustering operation, and update the repetitions vector.
4. Repeat 1,2 until we reach a stopping condition. We plan on using a stopping condition similar to: $\sum_{v_i \in V} \mathbf{q}(v_i) < K\max(|V|, P)$.

7.3 Wrap up

1. Schedule SDF uniprocessor clusters with the loop scheduler of reference [18], which has time complexity that is $O(m^2 + \sum_{v \in T} \mathbf{q}(v))$, where $m = \max(|V|, |E|)$, and T is the set of nodes that are contained in the *tightly interdependent components* of the SDF graph [16].[1]
2. Schedule user specified clusters with the given scheduler.
3. Schedule the clustered system with the user specified multiprocessor scheduler

8. Performance

The hierarchical scheduling framework for user specified clustering has been implemented in Ptolemy [1]. Four signal processing applications have been synthesized for a heterogeneous multiprocessor consisting of a RISC and a DSP processor. An example system that was designed within this framework will be detailed in the following section. A table comparing the results of user specified hierarchical scheduling versus full DAG expansion multiprocessor scheduling is given in table 1, and this data is depicted visually in figures 15 and 16. Note that for all systems, the hierarchical scheduling time was around 1 - 2 orders of magnitude faster, while the

[1] It has been observed that tightly interdependent components appear to be nonexistent in most practical SDF graph, and thus, the complexity is often simply $O(m^2)$ [16].

generated code was 1 - 2 orders of magnitude smaller. Also, three of the four systems scheduled using the full DAG expansion scheduling techniques exceed the DSP processor memory resources. The DSP card only has 16K available while all of the modem examples used at least 29K. For hierarchical scheduling the modem examples needed 1.5K or less. Finally, there was virtually no penalty for doing hierarchical scheduling, as can be seen in the makespans of the final multiprocessor schedules.

9. Acoustical modem example

In this section we detail a 320 bps quadrature amplitude modulation (4-QAM) acoustical modem [22] that is scheduled onto two heterogeneous processors (RISC, DSP). The SDF specification is shown in figure 17. A pseudo-random bit stream is generated on the workstation and then packed into a DSP word stream(22 bits/word). The stream of words is sent to a DSP which unpacks each word to form a bit stream. These bits are then encoded into a symbol (2 bits/symbol). The DSP transmits and then receives the symbol stream over a analog channel. The received symbols are then decoded, packed and sent back to the workstation, where the errors are displayed to the user. The user can control the alignment of the symbol period and examine the resultant constellation and eye diagram using the peek/poke mechanism described in [24]. All of

System	**SDF Graph Size**	**DAG Size**	**Scheduling Time in CPU Seconds**	**Makespan**	**P1: DSP Code Size Assembly**	**P2: Sparc Code Size C**
FM-Synthesis 128 pt. spectrum	44	14 / 806 57 x smaller	0.47 / 4.35 9.25 x faster	28832 / 28832 no difference	408 / 408 same	34K / 420K 12 x smaller
bpsk (530 bps)	31	9 / 2628 292 x smaller	0.37 / 14.71 40 x faster	41566 / 41368 < 1% difference	424 / 32045 75 x smaller	14K / 56K 4 x smaller
4-QAM (320 bps) eye diagram	59	15 / 9267 618 x smaller	0.91 / 80.87 87 x faster	150123 / 150123 no difference	1421 / 87533 62 x smaller	38K / 63K 1.7 x smaller
4-QAM (640 bps)	52	10 / 3490 349 x smaller	0.69 / 20.1 29 x faster	40037 / 39707 < 1% difference	848 / 29720 35 x smaller	35K / 56K 1.6 x smaller

Table 1. Performance of the hierarchical scheduling framework for user-specified clustering.

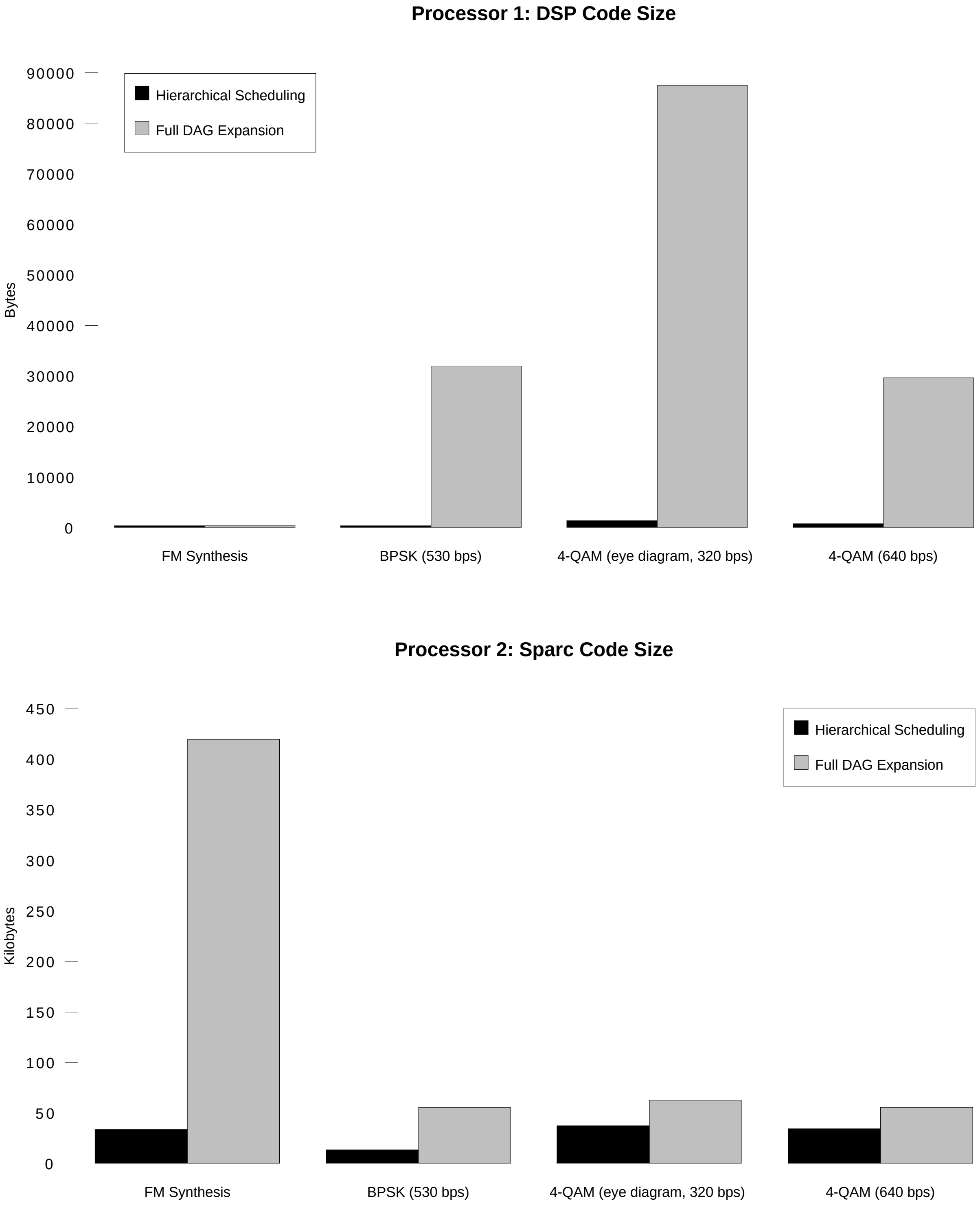


Figure 15. Performance of the hierarchical scheduling framework for user-specified clustering — code size.

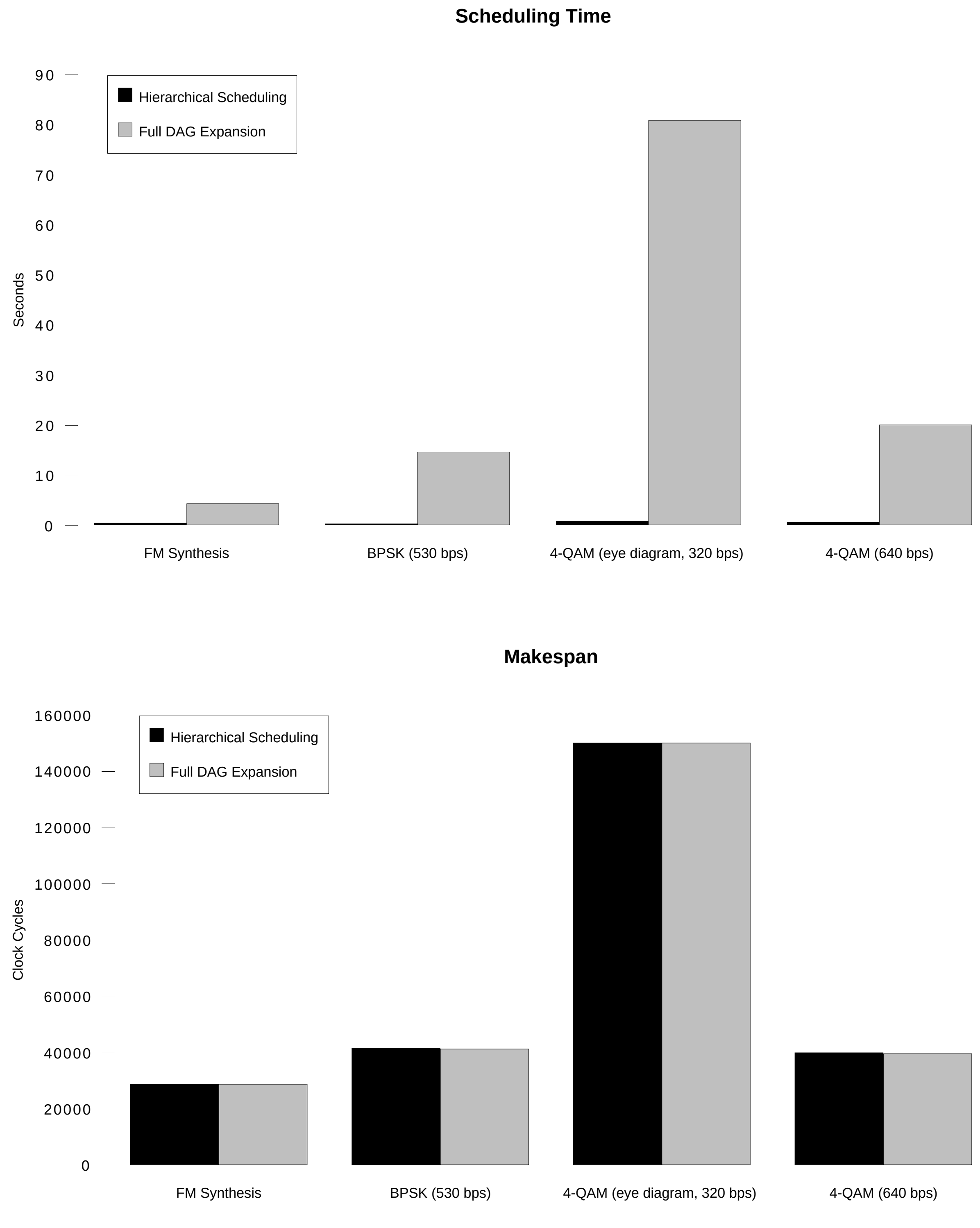


Figure 16. Performance of the hierarchical scheduling framework for user-specified clustering — scheduling time and makespan.

the transmitter and receiver filters are polyphase FIR filters with interpolation and decimation factors of 50 samples respectively.

Note that the SDF graph shown in figure 17 is expressed hierarchically. There are a total of 59 SDF nodes; the corresponding DAG has a total of 9267 nodes. Since we are able to use SDF uniprocessor schedulers on the SDF subgraph clusters, for this example, we are able to obtain a *single appearance schedule* which leads to very compact code. A single appearance schedule is an SDF schedule in which each node only appears once [18]. To obtain the single appearance schedule, three uniprocessor schedulers and one multiprocessor scheduler were used by the hierarchical scheduling framework. By using the cluster hierarchy, the multiprocessor scheduler only had to schedule a DAG with 8 nodes. The multiprocessor schedule generated from the fully

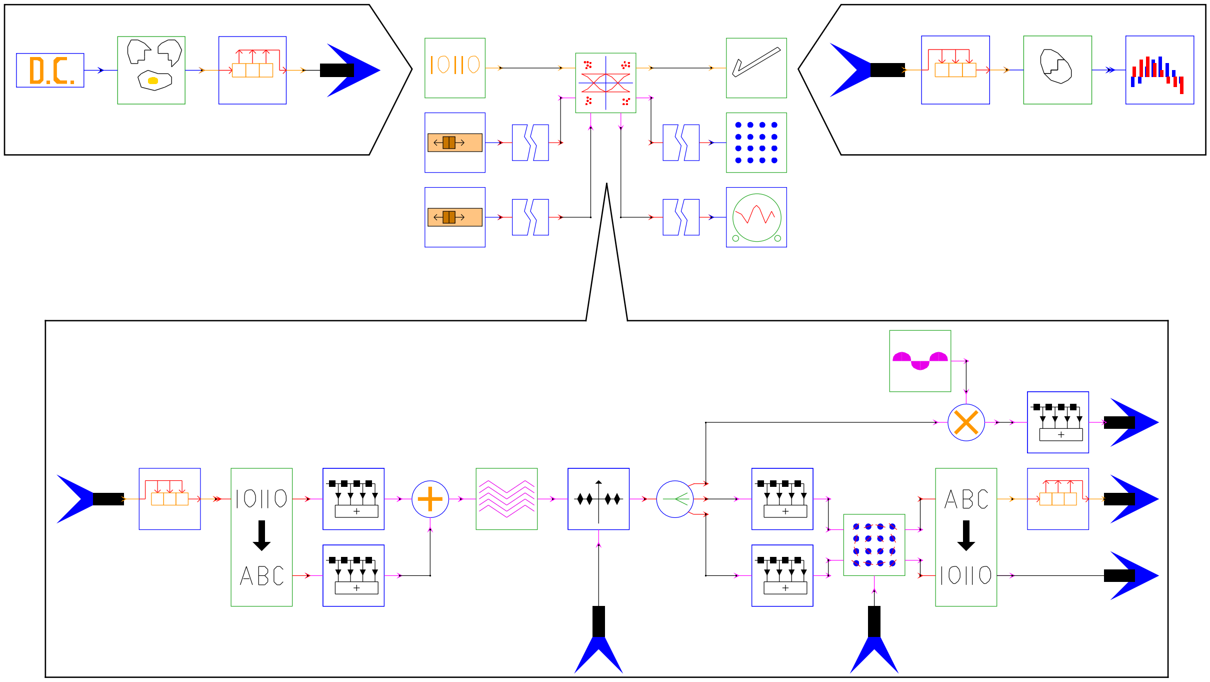


Figure 17. A 4-QAM acoustical modem. The top center block diagram is the top-level modem schematic. The hierarchy of *Pseudo-Random Bits*, *DSP Modem*, and *Error Display* blocks is expanded in the accompanying block diagrams. All of the blocks except for the *DSP Modem* execute on the host workstation. The *DSP Modem* executes on the Ariel S-56X DSP board.

expanded DAG has one function call (or inlined procedure) for each of its 9267 nodes as compared to only 59 function calls for the hierarchical schedule.

10. Conclusions

In this paper, we have introduced a hierarchical scheduling framework for SDF graphs being mapped onto multiple processors. Using user specified clustering, this framework has dramatically improved the scheduling time and reduced the memory requirements needed in the generated system. In some cases, the hierarchical scheduling framework enabled the synthesis of applications previously impossible using full DAG expansion multiprocessor scheduling techniques.

We plan to implement automated clustering heuristics for use on the SDF graph before the SDF to DAG translation. These will be inspired by the DAG clustering heuristics found in multiprocessor schedulers. The objective is to hide only that parallelism that would not be exploited, and in doing so, simplify the DAG.

References

[1] J.L. Pino and E.A. Lee, “Hierarchical static scheduling of dataflow graphs onto multiple processors,” *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Detroit, Michigan, IEEE, 1995.

[2] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *International Journal of Computer Simulation, special issue on Simulation Software Development*, vol. 4, 1994, p. 155-182.

[3] J.L. Pino, S. Ha, E.A. Lee, and J.T. Buck, “Software synthesis for DSP using Ptolemy,” *Journal of VLSI Signal Processing to appear in special issue on Synthesis for DSP*, 1993.

[4] E.A. Lee and S. Ha, “Scheduling strategies for multiprocessor real-time DSP,” *IEEE Global Telecommunications Conference and Exhibition. Communications Technology for the 1990s and Beyond*, vol. 2, Dallas, TX, USA, IEEE, 1989, p. 1279-1283.

[5] D.G. Powell, E. A.Lee, and W.C. Newman, “Direct synthesis of optimized DSP assembly code from signal flow block diagrams,” *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, San Francisco, CA, IEEE, 1992, p. 553-556.

[6] S. Ritz, M. Pankert, and H. Meyr, “High level software synthesis for signal processing systems,” *International Conference on Application Specific Array Processors*, Berkeley, CA, USA, IEEE Computer Society Press, 1992, p. 679-693.

[7] E.A. Lee and D.G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, 1987, p. 1235-1245.

[8] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Multirate signal processing in Ptolemy,” *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, Toronto, Ont., Canada, IEEE, 1991, p. 1245-1248.

[9] H. Printz, *Automatic mapping of large signal processing systems to a parallel machine*, Ph.D. Dissertation CMU-CS-91-101, Carnegie Mellon, 1991.

[10] G.C. Sih and E.A. Lee, “Dynamic-level scheduling for heterogeneous processor networks,” *Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, USA, IEEE Computer Society Press, 1990, p. 42-49.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, New York: W.H. Freeman, 1991.

[12] A. Gerasoulis and T. Yang, “A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, 1992, p. 276-291.

[13] S.J. Kim and J.C. Browne, “A general approach to mapping of parallel computations upon multiprocessor architectures,” *International Conference on Parallel Processing*, vol. 3, University Park, PA, USA, Pennsylvania State Univ, 1988, p. 1-8.

[14] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*, Cambridge, Mass.: MIT Press, 1989.

[15] G.C. Sih and E.A. Lee, “Declustering: A new multiprocessor scheduling technique,” *IEEE Transactions on Parallel and Distributed Systems*, 1992.

[16] S.S. Bhattacharyya, *Compiling dataflow programs for digital signal processing*, Ph.D. Dissertation UCB/ERL M94/52, University of California at Berkeley, 1994.

[17] P.K. Murthy, S.S. Bhattacharyya, and E.A. Lee, *Combined code and data minimization for synchronous dataflow programs*, Memorandum UCB/ERL M94/93, Electronics Research Laboratory, University of California at Berkeley, December,1994.

[18] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, “Generating compact code from dataflow specifications of multirate signal processing algorithms,” *IEEE Transactions on Circuits and Systems — I: Fundamental Theory and Applications*, vol. 42, no. 3, p. 138-150, March, 1995.

[19] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, p. 146-160, 1972.

[20] J.L. Pino, T.M. Parks, and E.A. Lee, “Automatic code generation for heterogeneous multiprocessors,” *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2,

Adelaide, South Australia, 1994, p. 445-448.

[21] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to algorithms*, New York: MIT Press, 1990.

[22] E.A. Lee and D.G. Messerschmitt, *Digital communication*, Boston: Kluwer Academic Publishers, 1994.

[23] E. A. Lee, *A coupled hardware and software architecture for programmable digital signal processors*, Ph. D. thesis, Dept. of Electrical Engineering and Computer Sciences, University of California at Berkeley, May, 1986.

[24] J.L. Pino, T.M. Parks, and E.A. Lee, “Mapping multiple independent synchronous dataflow graphs onto heterogeneous multiprocessors,” *IEEE Asilomar Conference on Signals, Systems, and Computers,* Pacific Grove, CA, 1994.

[25] S. S. Bhattacharyya and E. A. Lee, “Scheduling synchronous dataflow graphs for efficient looping,” *Journal of VLSI Signal Processing*, December 1993.

[26] J. T. Buck, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Ph.D. thesis, Memorandum UCB/ERL M93/69, September, 1993.

[27] S. How, *Code generation for multirate DSP systems in Gabriel*, Memorandum UCB/ERL M94/82, Electronics Research Laboratory, University of California at Berkeley, October, 1994.

[28] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Two complementary heuristics for translating graphical DSP programs into minimum memory implementations*, Memorandum UCB/ERL M95/3, Electronics Research Laboratory, University of California at Berkeley, January, 1995.

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

[30] R. M. Karp and R. E. Miller, “Properties of a model for parallel computations: determinacy, termination and queuing,” *SIAM Journal on Applied Mathematics,* vol. 14, no. 6, p. 1390-1411, November, 1966.

[31] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static data flow,” *Proceeding of the International Conference on Acoustics, Speech, and Signal Processing*, Detroit, May, 1995.