Design Automation for Embedded Systems, 7, 307–323, 2002.

Intermediate Representations for Design Automation of Multiprocessor DSP Systems

NEAL BAMBHA nbambha@eng.umd.edu
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

VIDA KIANZAD vida@eng.umd.edu
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

MUKUL KHANDELIA mukulk@eng.umd.edu
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

SHUVRA S. BHATTACHARYYA ssb@eng.umd.edu
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

Abstract. Self-timed scheduling is an attractive implementation style for multiprocessor DSP systems due to its ability to exploit predictability in application behavior, its avoidance of over-constrained synchronization, and its simplified clocking requirements. However, analysis and optimization of self-timed systems under real-time constraints is challenging due to the complex, irregular dynamics of self-timed operation. In this paper, we review a number of high-level intermediate representations for compiling dataflow programs onto self-timed DSP platforms, including representations for modeling the placement of interprocessor communication (IPC) operations; separating synchronization from data transfer during IPC; modeling and optimizing linear orderings of communication operations; performing accurate design space exploration under communication resource contention; and exploring alternative processor assignments during the synthesis process. We review the structure of these representations, and discuss efficient techniques that operate on them to streamline scheduling, communication synthesis, and power management of multiprocessor DSP implementations.

Keywords: Dataflow graphs, embedded systems, digital signal processing, interprocessor communication, self-timed scheduling.

1. Background

Multiprocessor implementation of DSP applications involves the interaction of several complex factors including scheduling, interprocessor communication, synchronization, iterative execution, and more recently, voltage scaling for low power implementation. Addressing any one of these factors in isolation is itself typically intractable in any optimal sense; at the same time, with the increasing trend toward multi-objective implementation criteria in the synthesis of embedded software, it is desirable to understand the joint impact of these factors. In this paper, we examine several high-level, intermediate representations that have been developed to analyze and optimize various multiprocessor DSP implementation factors and manage their interactions.

The techniques discussed in this paper pertain to system specifications based on iterative synchronous dataflow (*SDF*) graphs [10]. Iterative SDF programming of DSP applications has been researched widely in the context of multiprocessor implementation, and numerous commercial DSP tools have been developed that incorporate SDF semantics. Examples of such tools include SPW by Cadence, COSSAP by Synopsys, and ADS by Hewlett-Packard.

In SDF, an application is represented as a directed graph in which vertices (***actors***) represent computational tasks, edges specify data dependences, and the numbers of data values (***tokens***) produced and consumed by each actor is fixed. Delays on SDF edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the kth invocation of actor A are consumed by the $(k + 2)$th invocation of actor B, then the edge (A, B) contains two delays. Actors can be of arbitrary complexity. In DSP design environments, they typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters. We refer to an SDF representation of an application as an ***application graph***.

In this paper, we use a form of SDF called *homogeneous* SDF (HSDF) that is suitable for dataflow-based multiprocessor design tools since it exposes parallelism more thoroughly. In HSDF, each actor transfers a single token to / from each incident edge. General techniques for converting SDF graphs into HSDF form are developed in [10]. We represent a dataflow graph by an ordered pair (V, E), where V is the set of actors and E is the set of edges. We refer to the source and sink actors of a dataflow edge e by $src(e)$ and $snk(e)$, we denote the delay on e by $delay(e)$, and we occasionally represent an edge e by the ordered pair $(src(e), snk(e))$. We say that e is an ***output edge*** of $src(e)$; e is an ***input edge*** of $snk(e)$; and e is ***delayless*** if $delay(e) = 0$. The execution time or estimated execution time of an actor v is denoted $t(v)$.

Mapping an application graph onto a multiprocessor architecture includes three important steps—assigning actors to processors (*processor assignment*), ordering the actors assigned to each processor (*actor ordering*), and determining when each actor should commence execution. All of these tasks can either be performed at run-time or at compile time to give us different scheduling strategies.

In relation to the scheduling taxonomy of Lee and Ha [9], we focus in this paper on the *self-timed* strategy and the closely-related *ordered transaction* strategy. These approaches are popular and efficient for the DSP domain due to their combination of robustness, predictability, and flexibility [17]. In self-timed scheduling, each processor executes the tasks assigned to it in a fixed order that is specified at compile time. Before executing an actor, a processor waits for the data needed by that actor to become available. Thus, processors are required to perform run-time synchronization when they communicate data. This provides robustness when the execution times of tasks are not known precisely or when they may exhibit occasional deviations from their compile-time estimates. Examples of an application graph and a corresponding self-timed schedule are illustrated in Figure 1.

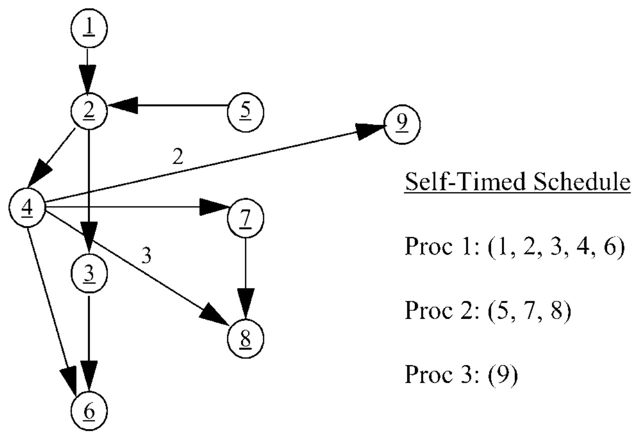


Figure 1. An example of an application graph and an associated self-timed schedule. The numbers on edges (4, 8) and (4, 9) denote nonzero delays.

The *ordered transaction* method is similar to the self-timed method, but it also adds the constraint that a linear ordering of the communication actors is determined at compile time, and enforced at run-time [18]. The linear ordering imposed is called the *transaction order* of the associated multiprocessor implementation. The transaction order, which is enforced by special hardware, obviates run-time synchronization and bus arbitration, and also enhances predictability. Also, if constructed carefully, it can in general lead to a more efficient pattern of actor/communication operations compared to an equivalent self-timed implementation [6].

2. Modeling Self-Timed Execution

In this section, we discuss two related graph-theoretic models, the *interprocessor communication graph (IPC graph)* G_{ipc} [17], [18] and the *synchronization graph* G_s [17], that are used to model the self-timed execution of a given parallel schedule for a dataflow graph. Given a self-timed multiprocessor schedule for G, we derive G_{ipc} by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge (x, y) in G that connects tasks that execute on different processors, an IPC edge is instantiated in G_{ipc} from x to y. Figure 2 shows the IPC graph that corresponds to the application graph and self-timed schedule of Figure 1. In this graph, the nodes labeled with "s" are nodes that send data and the nodes labeled with "r" are nodes that receive data.

Vertices in these graphs correspond to individual tasks of the application being implemented. Each edge in G_{ipc} and G_s is either an *intraprocessor edge* or an *interprocessor edge.* Intraprocessor edges model the ordering (specified by the given parallel schedule) of tasks assigned to the same processor. Interprocessor edges in G_{ipc}, called *IPC edges*, connect

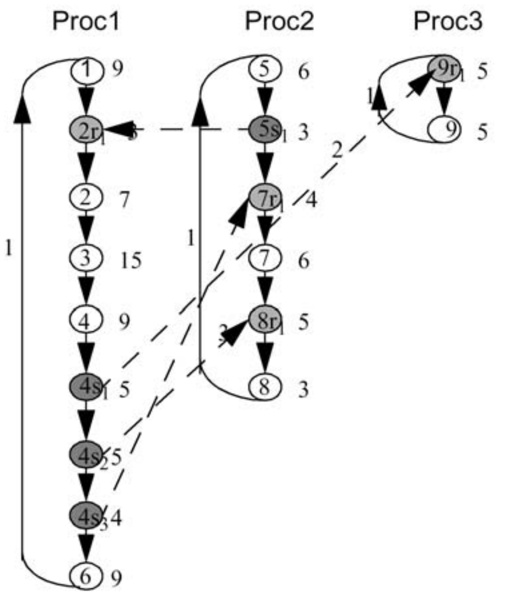


Figure 2. The IPC graph constructed from the application graph and schedule of Figure 1. Dashed edges represent IPC edges and the shaded actors are communication actors (send and receive actors) that perform interprocessor communication.

tasks assigned to distinct processors that must communicate for the purpose of data transfer, and interprocessor edges in G_s, called *synchronization edges*, connect tasks assigned to distinct processors that must communicate for synchronization purposes.

Each edge (v_j, v_i) in G_{ipc} represents the *synchronization constraint*

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) \quad \text{for all } k, \tag{1}$$

where $start(v, k)$ and $end(v, k)$ respectively represent the time at which invocation k of actor v begins execution and completes execution, and $delay(e)$ represents the delay associated with edge e.

Initially, the synchronization graph G_s is identical to G_{ipc}. However, various transformations can be applied to G_s in order to make the overall synchronization structure more efficient. After all transformations on G_s are complete, G_s and G_{ipc} can be used to map the given parallel schedule into an implementation on the target architecture. The IPC edges in G_{ipc} represent buffer activity, and are implemented as buffers in shared memory, whereas the synchronization edges of G_s represent synchronization constraints, and are implemented by updating and testing flags in shared memory. If there is an IPC edge as well as a synchronization edge between the same pair of tasks, then a synchronization protocol is executed before the buffer corresponding to the IPC edge is accessed to ensure sender−receiver synchronization. On the other hand, if there is an IPC edge between two tasks in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two tasks but no IPC edge, then no shared buffer is allocated between the two tasks; only the corresponding synchronization protocol is invoked.

Any transformation that we perform on the synchronization graph must respect the synchronization constraints implied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph (in conjunction with the IPC edges of G_{ipc}). If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), we say that G_1 ***preserves*** G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have

$$\rho_{G_1}(src(e), snk(e)) \leq delay(e), \tag{2}$$

where $\rho_G(x, y) \equiv \infty$ if there is no path from x to y in the synchronization graph G, and if there is a path from x to y, then $\rho_G(x, y)$ is the minimum over all paths p directed from x to y of the sum of the edge delays on p. Thus, G_1 preserves G_2 if for any new edge in G_2 (i.e., for any edge not in G_1), there is a path in G_1 directed from the source of the edge to the sink that has a cumulative delay that is less than or equal to the delay of the edge. The following theorem (developed in [17]) is fundamental to synchronization graph analysis.

THEOREM 1
The synchronization constraints (from (1)) of G_1 imply the constraints of G_2 if G_1 preserves G_2.

Theorem 1 underlies the validity of a variety of useful synchronization graph transformations, which include systematic removal of redundant synchronization edges; rearrangement of synchronization edges to trade-off latency and throughput; graph transformations for use of low-overhead synchronization protocols; and streamlined sizing of interprocessor communication buffers [17].

3. Ordering Communication

The IPC graph is an instance of Reiter's *computation graph* model [14], also known as the *timed marked graph* model in Petri net theory [13], and from the theory of such graphs, it is well known that in the ideal case of unlimited bus bandwidth, the average iteration period for the ASAP execution of an IPC graph is given by the *maximum cycle mean (MCM)* of G_{ipc}, which is defined by

$$MCM(G_{ipc}) = \max_{\text{cycle } C \text{ in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{Delay\,(C)} \right\}, \tag{3}$$

where $Delay(C)$ denotes the sum of the edge delays over all edges in the cycle C.

The MCM is thus the maximum over all directed cycles C of the sum of the task execution times in C divided by the sum of the edge delays in C. The quotient in (3) is referred to as the *cycle mean* of the associated cycle C. A variety of efficient, low polynomial-time algorithms have been developed for computing MCMs (e.g., see [5]).

IPC costs (estimated transmission latencies through the multiprocessor network) can be incorporated into the IPC graph model, and the performance expression (3), by explicitly including *communication* (*send* and *receive*) *actors*, and setting the execution times of these actors to equal the associated IPC costs. In this case, the performance estimate (3) is limited by any underlying uncertainties in the actor execution times, and run-time contention due to shared communication resources. Nevertheless, it has proven to be a useful estimate of performance during design space exploration for multiprocessor DSP.

A similar data structure, which is useful in analyzing ordered transaction implementations, is Sriram's *ordered transaction graph* model [18]. Given an ordering $O = \{o_1, o_2, \ldots o_p\}$ for the communication actors in an IPC graph $G_{ipc} = (V_{ipc}, E_{ipc})$, the corresponding ordered transaction graph $\Gamma(G_{ipc}, O)$ is defined as the directed graph $G_{OT} = (V_{OT}, E_{OT})$, where

$$V_{OT} = V_{ipc}, \quad E_{OT} = E_{ipc} \cup E_O, \tag{4}$$

$$E_O = \{(o_p, o_1), (o_1, o_2), (o_2, o_3), \ldots, (o_{p-1}, o_p)\}, \tag{5}$$

$$delay(o_i, o_{i+1}) = 0 \quad \text{for } 1 \le i < p,$$

$$\text{and} \quad delay(o_p, o_1) = 1. \tag{6}$$

Thus, an IPC graph can be modified by adding edges (the edges in E_O) obtained from the ordering O to create the ordered transaction graph.

A closely related data structure is the *transaction partial order graph* G_{TPO} that is computed from the IPC graph by first deleting all edges in G_{ipc} that have delays of one or more, and then deleting all of the computation actors. The transaction partial order graph represents the minimum set of dependencies imposed among different processors by the communication actors of the IPC graph. These dependencies must be obeyed by any ordering of the communication operations.

Figure 3 shows an example of a transaction partial order graph.

As described in Section 1, when the ordered transaction strategy is implemented using a hardware method such as a micro-controller that imposes the linear order, there is no need for synchronization and contention for shared communication resources is also eliminated. Therefore, if the execution time estimates for the actors are accurate or are true worst-case values, then the MCM of the ordered transaction graph gives us an accurate estimate or worst-case bound, respectively, of the iteration period of the associated application graph under the ordered transaction strategy. Such efficient, accurate performance assessment is useful for design space exploration in general, and it is especially useful when implementing applications that have real time constraints.

If interprocessor communication costs are negligible, an optimal transaction order can be computed in low polynomial time for a given self-timed schedule [18]. We call this method of deriving transaction orders the *Bellman-Ford Based* (BFB) method since it is

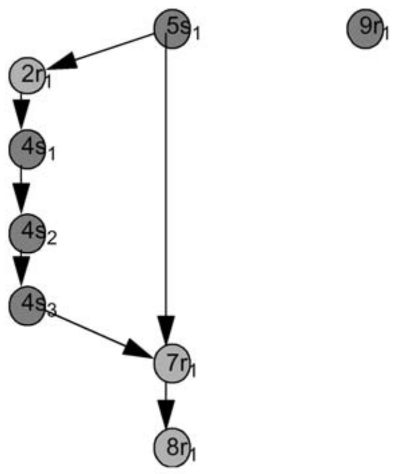


Figure 3. The transaction partial order graph constructed from IPC graph of Figure 2.

based on applying the Bellman-Ford shortest path algorithm to an intermediate graph that is derived from the given self-timed schedule.

However, when IPC costs are not negligible, as is frequently and increasingly the case in practice, the problem of determining an optimal transaction order is NP-hard [6]. This intractability has been shown to hold both under iterative and non-iterative execution of application graphs. Thus, under nonzero IPC costs, we must resort to heuristics for efficient solutions. Furthermore, the polynomial-time BFB algorithm is no longer optimal, and alternative techniques that account for IPC costs are preferable.

In the presence of non-negligible communication costs, an efficient transaction order can be constructed with the help of the transaction partial order graph G_{TPO} described earlier. The *transaction partial order algorithm* is one systematic approach for using transaction partial order graphs to construct efficient orderings of communication operations. This algorithm proceeds by considering—one by one—each vertex of G_{TPO} that has no input edges (vertices in the transaction partial order graph that have no input edges are called *ready* vertices) as a *candidate* to be scheduled next in the transaction order. Interprocessor edges are inserted from each candidate vertex to all other ready vertices in G_{ipc}, and the corresponding MCM is measured. The candidate whose corresponding MCM is the least when evaluated in this fashion is chosen as the next vertex in the ordered transaction, and deleted from G_{TPO}. This process is repeated until all communication actors have been scheduled into a linear ordering.

Figure 4 shows an example of an ordered transaction graph that is derived using the transaction partial order algorithm. The algorithm has been shown to perform consistently well on DSP application graphs [6].

While the ordered transaction method is useful in its total elimination of run-time synchronization and bus arbitration overhead, the transaction partial order heuristic is able to improve the performance beyond what is achievable by a self-timed schedule even if synchronization and arbitration costs are negligible compared to actor execution

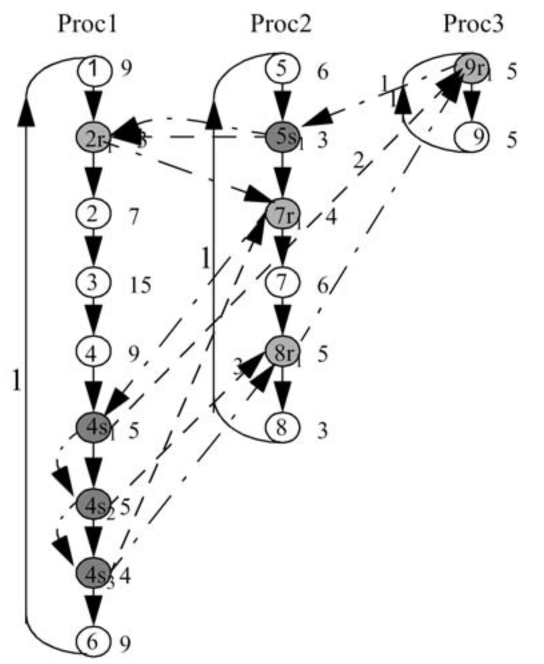


Figure 4. An example of an ordered transaction graph that is derived by the transaction partial order algorithm.

times [6]. Such performance benefit is achieved by strategic positioning of the communication operations in ways that do not evolve from the natural evolution of self-timed schedules [6], [7].

In summary, the ordered transaction and transaction partial order graphs are useful representations for ordering communication operations on platforms that provide support for enforcing such orderings. Optimal transaction orders can be derived in polynomial time if IPC costs are negligible; however, the performance of the self-timed schedule is an upper bound on the performance of corresponding ordered transaction schedules under negligible IPC costs. Conversely, when IPC costs are non-negligible, optimal transaction ordering is NP-hard, but the performance of a self-timed schedule can, in general, be exceeded significantly by a carefully-constructed transaction order. As the relative costs of global communication increase in embedded multiprocessor platforms, this latter relationship becomes an important design consideration.

4. The Period Graph Model

Recall that given predictable actor execution times, one can apply (3) to accurately assess system throughput in the absence of any contention for communication resources. However, with the use of shared buses, which are employed in many embedded multiprocessor architectures, the accuracy of estimates based on (3) can be expected to degrade with the level of bus contention that results at run-time. Fortunately, this does not affect the validity or utility of the communication and synchronization management techniques discussed in Section 2, since these techniques operate directly on the sets of interprocessor communication and synchronization edges, without need for performance estimation. Furthermore, this limitation is not encountered when using the ordered transaction model of Section 3, since contention is eliminated under this implementation model regardless of the medium used for communication.

However, accurate performance assessment of self-timed systems involving shared communication resources in general must be able to handle contention on these resources. In a typical embedded multiprocessor optimization scenario, such assessment is carried out repeatedly in order to evaluate candidate solutions for a given design. There are many optimizations for which one could explore variations in task execution times. Examples include exploring migrations between hardware and software, applying voltage scaling, exploring code size/performance trade-offs for a particular task in the application, or investigating alternative processor assignments in a heterogeneous multiprocessor. In order to evaluate a particular change to a given task, the performance impact of the change must be evaluated.

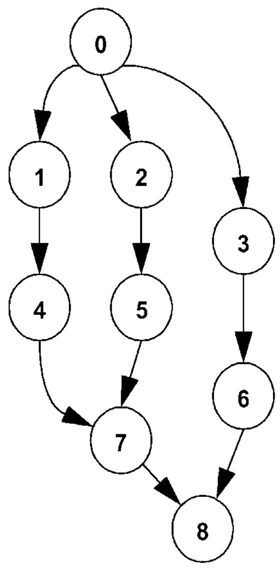
One consequence of communication resource contention in this context is that under iterative execution that is self-timed, there is no known method for deriving an analytical expression for the throughput of the system, and thus, simulation is required to get a clear picture of application performance. However, simulation is computationally expensive, and it is highly undesirable to perform simulation inside the innermost optimization loop during synthesis.

The *period graph* is an efficient estimator for the system throughput that can be employed to avoid such inner-loop simulation [2]. In particular, the reciprocal of the MCM of the period graph can be used as an efficient estimate of the throughput.

If communication resource contention is resolved deterministically, and execution times are constant, then self-timed evolution may lead to an initial transient state, but the execution will eventually become periodic [17]. This holds because the multiprocessor may be modeled as a finite-state system, and thus, aperiodic behavior—which implies the presence of infinitely many distinct states—cannot hold. In DSP systems, although execution times are not always constant, or known precisely, they typically adhere closely to their respective estimates with high frequency. Under such conditions, the periodic execution pattern obtained from the estimated execution times provides an estimate of overall system throughput based on the task-level estimates.

For self-timed systems, when we apply execution time estimates to assess overall throughput, it is necessary to simulate (using the execution time estimates) past the transient state until a periodic execution pattern (steady state) emerges. Unfortunately, the duration of the transient may be exponential in the size of the application specification [17], and this makes simulation-intensive, iterative synthesis approaches highly unattractive.

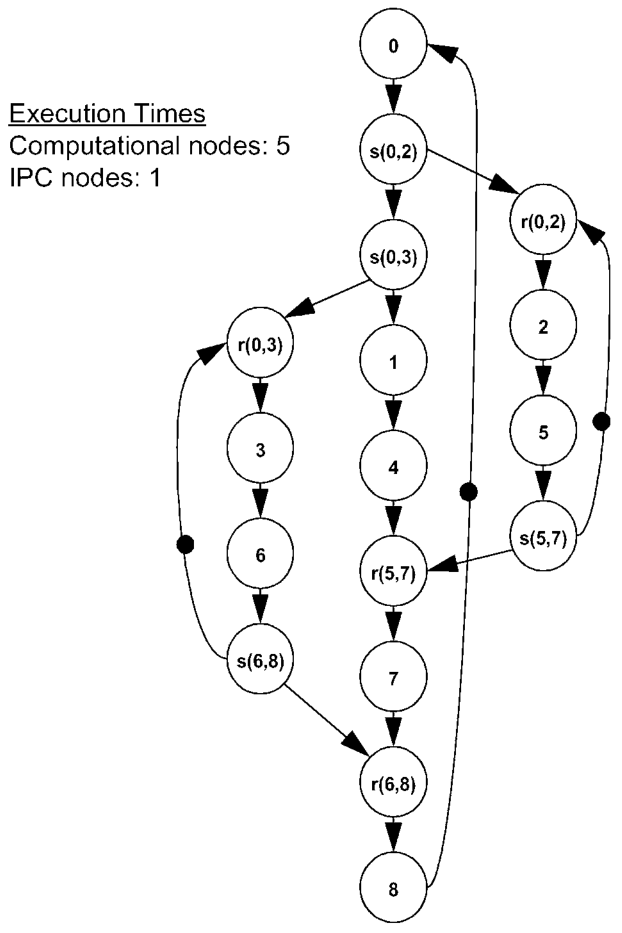
The period graph model greatly reduces the rate at which simulation must be carried out during iterative synthesis. Given an assignment v of task execution times, and a self-timed schedule, the associated period graph is constructed from the periodic, steady-state



(a)

Proc1	Proc2	Proc3
0	2	3
1	5	6
4		
7		
8		

(b)

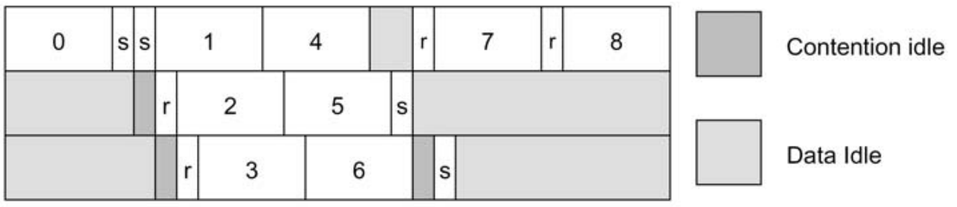


(c)

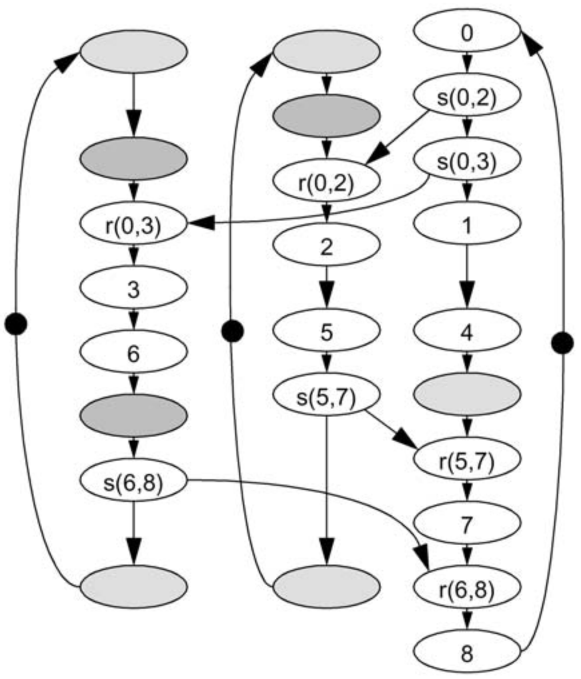
Figure 5. An example of an application and IPC graph that are used to illustrate the construction of period graphs. (a) Application graph; (b) schedule; and (c) IPC graph.

pattern of the resulting simulation. The MCM of the period graph (with certain adjustments) is then used as a computationally-efficient means of estimating the iteration period (the reciprocal of the throughput) as changes are explored within a neighborhood of *v*.

Figure 5(a) and Figure 5(b) illustrate an application graph along with a self-timed schedule and the associated IPC graph, as defined in Section 2, which we use to illustrate construction of period graphs. The solid black circles on the edges in the IPC graph represent delays, which model inter-iteration dependencies. Here, a node labeled $s(i, j)$ represents a communication actor that sends data from node i to node j. Similarly, a node labeled $r(i, j)$



(a)



(b)

Figure 6. (a) The periodic component of simulation output for the IPC graph in Figure 5. (b) The period graph constructed from (a).

represents an actor that receives data from node *i* destined for node *j*. The throughput can be determined by conducting a discrete event simulation of the IPC graph, given a model for resolution of communication resource contention. Figure 6 shows the periodic portion of the simulator output of the IPC graph of Figure 5, where the processors are connected by a shared bus, and lower numbered processors are granted priority in the case where two processors request the bus simultaneously.

The first step in the construction of the period graph is the identification of the period (steady state periodic execution pattern) from the simulator output. This can be performed by tracing backward through the simulation and searching for the latest intermediate time

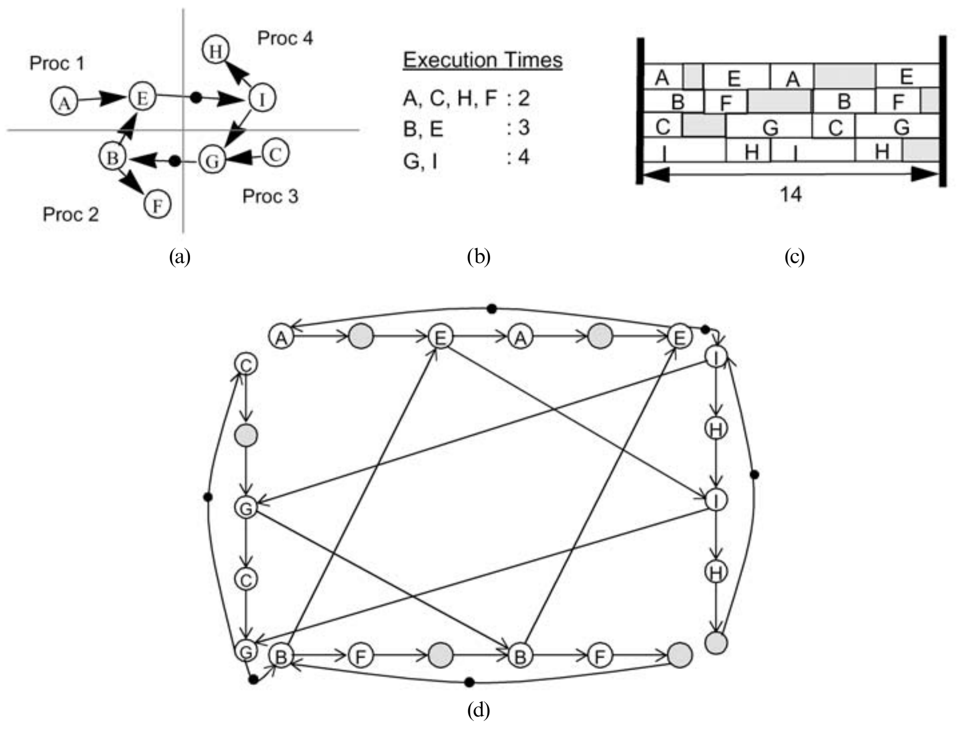


Figure 7. An illustration of a period graph model that spans multiple application iterations.

instant t_a at which the *system state* $S(t_a)$ equals the state $S(t_f)$ obtained at the end of the simulation (here, t_f denotes the simulation time limit). If no match is found, then the end of the first period exceeds t_f, and thus, the simulation needs to be extended beyond t_f. Otherwise, the region in the simulation profile (Gantt chart) that spans the interval $[t_a, t_f]$ constitutes a (minimal) period of the simulated steady state.

Here, the system state $S(t)$ contains the execution state of each processor, which is either "idle" or is represented by an ordered pair (A, τ), where A is the task being executed at time t, and τ denotes the time remaining until the current invocation of A is completed. The state $S(t)$ also contains the current buffer sizes of all IPC buffers, as well as any information (e.g., request queue status) that is used by the protocol for resolution of communication contention. The periodic part of the simulator output may span one iteration of all the tasks as shown in Figure 6, or multiple iterations of all the tasks as illustrated in Figure 7. As illustrated in Figure 6, the period graph consists of all the tasks comprising the period that was detected, with the idle time ranging between tasks (including those that are caused by communication contention) *also treated as nodes in the graph.* For purposes of analysis, it is useful to separate the idle nodes into two sets. When a node has the necessary data to execute, but is idle waiting for access to the bus, the associated idle node is classified as a *contention idle*. When a node is idle waiting for its predecessors' data, the associated idle node is classified as a *data idle*. Contention idles are a consequence of mutual exclusivity, while data idles are a form of condition synchronization. The nodes are connected by edges in the order that they appear in the period. An edge is placed from the last node in the period for each processor to the first node in the period. This edge is given a delay value of one (to model the associated transition between period iterations), while all of the other intraprocessor edges have delay values of zero. This is done for all the processors in the system. The period graph is completed by adding an edge from each send node to its corresponding receive node. Further details on period graph extraction are developed in [2].

Figure 7(a) and Figure 7(b) illustrate another application graph along with a self-timed schedule; Figure 7(c) shows the periodic steady state that results from the schedule of Figure 7(a) and the execution time estimates shown in Figure 7(b); and Figure 7(d) shows the resulting period graph. The nodes in Figure 7(d) that contain diagonal stripes correspond to idle time ranges in the period. Note that the steady state period may span multiple graph iterations (2 in this example), and in the period graph, this translates to multiple instances of each application graph task. For purposes of clarity in Figure 7, we have assumed negligible latency associated with IPC in this illustration.

Once the period graph has been constructed, it can be used as an efficient estimator for the throughput in any optimization for which the execution times of the nodes are varied (e.g., when exploring migrations between hardware and software, applying voltage scaling, or exploring alternative processor assignments in a heterogeneous multiprocessor). However, it is not obvious how one should adjust the idle times in the period graph. It is observed in [2] that the effects of contention can be captured efficiently with high estimation accuracy by ignoring (setting to zero) the data idles and leaving the contention idles constant as the computation times are scaled.

The period graph has been applied to the problem of voltage scaling for power reduction of multiprocessor DSP systems. It has been shown to increase overall power optimization efficiency significantly when used to explore voltage variations within a limited range around a given voltage vector (assignment of processor voltages) [2]. For larger changes in node execution times, the fidelity (accuracy) of the estimate decreases. In general, one would use the period graph in a local search, for which the fidelity is acceptable, and re-simulate and rebuild the period graph outside this region when necessary. This integration of period graph analysis with occasional re-simulation has been studied in [3].

5. Clusterization Function Representations

The concept of *clustering* has been widely applied to various applications and research problems such as parallel processing, load balancing and partitioning [11], [12]. Clustering is also often used as a front-end to multiprocessor system synthesis tools. In this context, clustering refers to the grouping of actors into subsets that execute on the same processor. The purpose of clustering is thus to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments.

In the context of embedded system implementation, one limitation shared by many existing clustering and scheduling techniques is that they have been designed for general purpose computation. In the general-purpose domain, there are many applications for which short compile time is of major concern. In such scenarios, it is highly desirable to ensure that an application can be mapped to an architecture within a matter of seconds. The internalization algorithm [15] and the dominant sequence algorithm [19] are examples of such low complexity algorithms.

Several probabilistic search approaches to multiprocessor scheduling have been proposed in the literature, such as *genetic algorithms*, that exploit the increased compile time tolerance available with embedded systems (e.g., see [1] for a general discussion of genetic algorithms, and [4] for an example of a recent genetic algorithm approach to scheduling). However, these approaches typically have complex, highly specialized solution representations in the underlying genetic algorithm formulation, and require "repair" mechanisms that further reduce their search efficiency. Specifically, the result of applying genetic operators, such as crossover or mutation, may generate an infeasible solution, which the GA must either discard or repair (to make it feasible). Repair mechanisms thus transform infeasible solutions into feasible ones. Repair may not always be successful.

The *clusterization function representation* is a mechanism for encoding candidate clustering solutions that is amenable to probabilistic search strategies, perhaps most notably to genetic algorithms, but that avoids the asymmetries and repair requirements that plague the effectiveness of conventional solution encodings that are used during scheduling [8]. The clusterization function concept is captured by the following definition.

DEFINITION 1

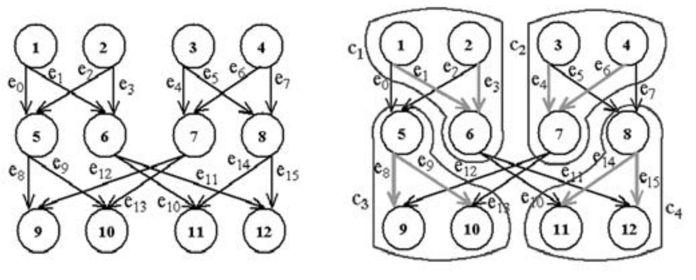
Suppose that β is a subset of application graph edges. Then f_β: $E \to \{0, 1\}$ denotes the clusterization function associated with β. This function is defined by:

$$f(e_i) = \begin{cases} 0 & \text{if } (e_i \in \beta) \\ 1 & \text{otherwise,} \end{cases} \tag{7}$$

where E is the set of application graph edges and $e_i \in E$.

By means of this definition, an enclosing search algorithm initially places actors in different clusters in some fashion (e.g., randomly) to generate an initial solution space, and then through successive refinement steps moves edges between clusters until solutions converge or some pre-defined limit on the number of refinement steps expires. When using a clusterization function to represent a clustering solution, the edge subset β is taken to be the set of edges that are contained in clusters. An illustration is shown in Figure 8.

This subset view of clustering develops a natural and efficient mapping into the framework of genetic algorithms. Derived from the *schema* theory of genetic algorithms (a schema denotes a similarity template that represents a subset of $\{0, 1\}^l$), canonical genetic algorithms (which use binary representation of each solution as fixed-length strings over the set $\{0, 1\}$ and efficiently handle optimization problems of the form f: $\{0, 1\} \to \Re$) provide near-optimal sampling strategies over subsequent generations. Furthermore,



$\beta_a = \varnothing$

$\beta_b = \{e_1, e_3\}$
$\cup \{e_4, e_6\}$
$\cup \{e_8, e_9\}$
$\cup \{e_{14}, e_{15}\}$
$= \{e_1, e_3, e_4, e_6, e_8, e_9, e_{14}, e_{15}\}$

(a) (b) (c)

	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	e_{13}	e_{14}	e_{15}
a.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
b.	1	0	1	0	0	1	0	1	0	0	1	1	1	1	0	0

Figure 8. (a) An application graph representation of an FFTand the associated clusterization function f_{β_a}; (b) a clustering of the FFTapplication graph, and f_{β_b} (c) the resulting subset β_b of clustered edges, along with the (empty) subset β_a of clustered edges in the original (unclustered) application graph.

binary encodings in which the semantic interpretations of different bit positions exhibit high symmetry (e.g., with the clusterization function, each bit corresponds to the existence or absence of an edge within a cluster) allow search techniques to leverage extensive prior research on genetic operators for symmetric encodings rather than forcing the development of specialized, less-thoroughly-tested operators to handle the underlying non symmetric, non traditional and sequenced-based representation. Accordingly, the clusterization function encoding scheme is favored both by schema theory, and significant prior work on genetic operators. Furthermore, by providing no constraints on genetic operators, clusterization functions preserve the natural behavior of genetic algorithms. Finally, a clusterization function encoding never generates an illegal or invalid solution, and thus saves repair-related synthesis time that would otherwise have been wasted in locating, removing or correcting invalid solutions.

The clusterization function approach has been applied to develop a genetic algorithm that schedules application graphs to minimize the latency of each application graph iteration (schedule makespan), while taking interprocessor communication overhead into account. In this approach, the initial genetic algorithm population is initialized with a random selection of clusterization functions (mappings from E into $\{0, 1\}$) and the fitness is evaluated using a modified version of list scheduling that abandons the restrictions imposed by a global scheduling clock, as proposed in [16]. This application of the clusterization function has been shown to significantly outperform existing clustering techniques, including the internalization algorithm, the dominant sequence algorithm, and randomized versions of the internalization and dominant sequence algorithms that were evaluated under equal amounts of synthesis time (equal amounts of time available for probabilistic search) [8].

Since clustering is widely applicable as a front-end to many multiprocessor design contexts, and the clusterization function formulation captures all possible clustering alternatives in an efficient and elegant representation, it is suitable for use in many types of tools for DSP system synthesis.

6. Summary

Designers of co-design and system synthesis tools for DSP can exploit the use of predictable, coarse-grain programming models, such as synchronous dataflow (SDF), which are considered too restrictive for general-purpose design tools. However, at the same time, multiprocessor DSP implementation is typically faced with an unusually complex range of design constraints and objectives. To help address this increasing trend toward high design complexity, this paper has discussed several SDF-based intermediate representations for self-timed implementation of multiprocessor DSP applications, including the interprocessor communication graph for modeling the placement of IPC operations; the synchronization graph for separating synchronization from data transfer during IPC; the ordered transaction and transaction partial order graphs for modeling and optimizing linear orderings of communication operations; the period graph for accurate design space exploration under communication resource contention; and the clusterization function concept for representing processor assignments during the scheduling process.

Acknowledgements

Portions of this research were sponsored by the U. S. National Science Foundation (9734275), the U. S. Army Research Laboratory (under Contract No. DAAL01-98-K-0075 and the MICRA program), and the Defense Advanced Research Projects Agency (MDA972-00-1-0023, through Brown University).

References

1. Back, T., U. Hammel, and H.-P. Schwefel. Evolutionary Computation: Comments on the History and Current State. *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1 pp. 3–17, April 1997.
2. Bambha, N. K., and S. S. Bhattacharyya. A Joint Power / Performance Optimization Technique for Multiprocessor Systems Using a Period Graph Construct. In *Proceedings of the International Symposium on Systems Synthesis*, pp. 91–97, Sept. 2000.
3. Bambha, N. K., S. S. Bhattacharyya, J. Teich, and E. Zitzler. Hybrid Search Strategies for Dynamic Voltage Scaling in Embedded Multiprocessors. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, pp. 243–248, Copenhagen, Denmark, April 2001.
4. Correa, R. C., A. Ferreira, and P. Rebreyend. Scheduling Multiprocessor Tasks with Genetic Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, vol. 0, pp. 825–837, 1999.
5. Dasdan, A., and R. K. Gupta. Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17 no. 10 pp. 889–899, Oct. 1998.

6. Khandelia, M., and S. S. Bhattacharyya. Contention-Conscious Transaction Ordering in Embedded Multiprocessors. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pp. 276–285, Boston, MA, July 2000.
7. Khandelia, M., and S. S. Bhattacharyya. Contention-Conscious Transaction Ordering in Embedded Multiprocessor Systems. Technical Report UMIACS-TR-2000-09, Institute for Advanced Computer Studies, University of Maryland at College Park, March 2000. Also Computer Science Technical Report CS-TR-4109.
8. Kianzad, V., and S. S. Bhattacharyya. Multiprocessor Clustering for Embedded Systems. In *Proceedings of the European Conference on Parallel Computing*, pp. 697–701, Manchester, United Kingdom, Aug. 2001.
9. Lee, E. A., and S. Ha. Scheduling Strategies for Multiprocessor Real Time DSP. In *Proceedings of the Global Telecommunications Conference*, Nov. 1989.
10. Lee, E. A., and D. G. Messerschmitt. Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, Feb. 1987.
11. Lieverse, P., E. F. Deprettere, A. C. J. Kienhuis, and E. A. De Kock. A Clustering Approach to Explore Grain-Sizes in the Definition of Processing Elements in Dataflow Architectures. *Journal of VLSI Signal Processing*, vol. 22, pp. 9–20, Aug. 1999.
12. Morse, J. N. Reducing the Size of the Nondominated Set: Pruning by Clustering. *Computers and Operations Research*, vol. 7, no. 1–2, pp. 55–66, 1980.
13. Peterson, J. L. *Petri Net Theory and Modeling of Systems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.
14. Reiter, R. Scheduling Parallel Computations. *Journal of the Association for Computing Machinery*, vol. 15, no. 4, pp. 590–599, Oct. 1968.
15. Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.
16. Sih, G. C., and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, 1993.
17. Sriram, S., and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.
18. Sriram, S., and E. A. Lee. Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors. *Journal of VLSI Signal Processing*, March 1997.
19. Yang, T. and, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, 1994.