Joint Application Mapping/Interconnect Synthesis Techniques for Embedded Chip-Scale Multiprocessors

Neal K. Bambha, *Member*, *IEEE*, and Shuvra S. Bhattacharyya, *Senior Member*, *IEEE*

Abstract—As transistor sizes shrink, interconnects represent an increasing bottleneck for chip designers. Several groups are developing new interconnection methods and system architectures to cope with this trend. New architectures require new methods for high-level application mapping and hardware/software codesign. In this paper, we present high-level scheduling and interconnect topology synthesis techniques for embedded multiprocessor systems-on-chip that are streamlined for one or more digital signal processing applications. That is, we seek to synthesize an *application-specific interconnect topology.* We show that flexible interconnect topologies utilizing low-hop communication between processors offer advantages for reduced power and latency. We show that existing multiprocessor scheduling algorithms can deadlock if the topology graph is not strongly connected, or if a constraint is imposed on the maximum number of hops allowed for communication. We detail an efficient algorithm that can be used in conjunction with existing scheduling algorithms for avoiding this deadlock. We show that it is advantageous to perform application scheduling and interconnect synthesis jointly, and present a probabilistic scheduling/interconnect algorithm that utilizes graph isomorphism to pare the design space.

Index Terms—Embedded multiprocessors, interconnect synthesis, scheduling, task graphs.

1 Introduction

Interconnect considerations are important for today's system-on-chip (SoC) designs. As transistor density increases, more functional units can be placed on a single chip, and the number of possible interconnections (links) between them increases. Standard bus architectures are not scalable for these designs. Long wires connecting functional units contribute to delay and limit the maximum achievable clock rate. Also, routing these interconnections is a significant challenge for the electronic design automation tools. A number of today's architectures for SoC provide special routing tracks for long interconnects. Networks on Chips (NoCs) have been proposed to replace bus interconnects in SoC (e.g., see [1], [2]). Future architectures may even incorporate optical interconnects. In this paper, we develop methods for deriving efficient dedicated-link interconnection networks in these architectures. The idea is that if we can incorporate routing constraints in the high-level front-end design stage, placement and routing can be improved in the back end of the design process and performance will increase.

Embedded systems typically run a limited and fixed set of applications. We can use this application-specific information to optimize the interconnection network. For our purposes, an optimal network is defined in the context of a set of applications and constraints. The constraints may include the latency, throughput, and power consumption for the given applications, along with cost and area constraints of the overall system.

One key distinguishing feature of our algorithm is that we focus on dedicated point-to-point (low-hop) interconnects. Existing cosynthesis techniques (e.g., see [3], [4]) use a more general interconnect model, in which point-to-point interconnects are a special case. Our focus on point-to-point interconnects motivates us to incorporate useful communication hop constraints in the joint synthesis, a new *flexibility metric* (Section 4.1) to help guide the scheduling algorithm, and the incorporation of graph isomorphism in the interconnect synthesis algorithm. Optical interconnect technology and NoC architectures are two emerging motivations for this focus.

Our general model for a SoC is one in which the chip is partitioned into regions that are connected with local interconnects, and these local regions are then connected through longer global interconnects. The global interconnect fabric is composed of point-to-point links that may be implemented by either a NoC or by optical interconnects.

The applications consist of *task graphs* [5] which are directed acyclic graphs (DAGs), where the individual tasks must fit fully into a local region. The graph vertices (*tasks* or *nodes*) in the acyclic task graphs represent computations while the edges represent data precedences between nodes. A computational cost is associated with each node, while a communication cost representing the amount of data that must be transferred between connected nodes is associated with each edge. An example task graph is shown in Fig. 1c.

One strength of the scheduling and interconnect synthesis algorithms described in this paper lies in the ability to handle an arbitrary interconnect topology, with constraints on interconnect fanout, number of interconnects, and the number of communication hops. Another strength is the

• N.K. Bambha is with the US Army Research Laboratory, 2800 Powder Mill Road, Adelphi, MD 20783. E-mail: nbambha@eng.umd.edu.
• S.S. Bhattacharyya is with the Department of Electrical and Computer Engineering and the Institute for Advanced Computer Studies, University of Maryland, College Park, College Park, MD 20742.
E-mail: ssb@eng.umd.edu.

Manuscript received 6 Feb. 2004; revised 15 July 2004; accepted 6 Aug. 2004; published online 20 Dec. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0040-0204.

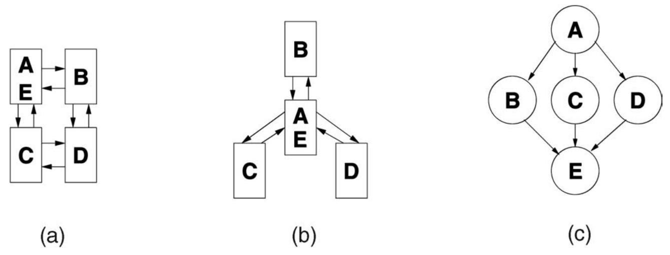


Fig. 1. (a)-(b) Two possible topologies with four processors and four bidirectional links. (c) Task graph.

ability to co-optimize the schedule and interconnect topology. For embedded systems, we seek to develop an application-specific interconnection network. An example illustrating the design flow, from an application topology mapping to interconnect synthesis, is given in the Appendix of this paper available online at http://www.computer.org/tpds/archives.htm.

2 Motivation and Previous Work

2.1 Network on Chip

Synchronization for future electrically-connected chips with a single clock source may not be feasible. One paradigm that has been proposed is to utilize locally synchronous regions with fully distributed, asynchronous global communication between them. In this case, communication on chip resembles a network. Murali and De Micheli [1] presented an algorithm that maps *cores*, or components of a SoC, onto a mesh NoC architecture, minimizing the average communication delay. Packet-switched and split routing is modeled. A *core graph* $G(V,E)$ is mapped onto a mesh NoC topology graph $P(U,F)$ with each $u \in U$ representing a node in the interconnect topology. Assignment of tasks to cores is not addressed.

Hu and Marculescu [6] presented an algorithm for mapping IPs onto a generic regular NoC architecture consisting of a network of *tiles*, each consisting of a processing core and a router. They constructed a deadlock-free deterministic routing function such that the total communication energy was minimized for a given application, taking its communication patterns into account. They demonstrated that flexible routing can be exploited to find solutions with lower link bandwidth requirements and significantly lower energy consumption.

Ho and Pinkston [7] developed design methodologies and algorithms for constructing application-specific, deadlock-free network topologies and routing functions that minimize resource cost and execution time by exploiting expected communication patterns of target application workloads. In their architecture, each processor is attached to the network via one physical link to a switch. Switches are connected by one or more physical links according to the topology generated by the algorithm. It was shown that these application-specific networks showed significant improvement in execution time over mesh networks while using up to 60 percent fewer resources.

In [6] and [7], the assignment and scheduling of tasks was done first, then profiling was done to derive the communication patterns of the application used in the topology synthesis and routing algorithms. In contrast, our algorithm performs task assignment and scheduling of a task graph $G(V,E)$ jointly with interconnect synthesis onto a dedicated-link (not switched) topology graph $T(P,L)$. In this way, the communication pattern of the application can be optimized together with the network topology.

2.2 Interconnect Synthesis

Most FPGA designs use a hierarchy of interconnect segments of differing lengths. In the Xilinx Virtex-II architecture, the configurable logic blocks, memory, and I/O blocks are all tied to a general routing matrix. The interconnect is programmable and hierarchical, with 24 vertical and horizontal long lines per row or column. In this paper, we express the relative distribution of local interconnects to global interconnects as a *fanout constraint*, which is simply the number of long (global) interconnects available for each local region.

Several research groups have proposed *pipelined* FPGA architectures [8], [9] which provide for long interconnects through a large number of registers. For example, in the RaPid architecture, short tracks are used to achieve local connectivity between functional units, while long tracks traverse longer distances along the datapath. Sharma et al. show that the area-delay product in this architecture is sensitive to the short track/long track ratio [10]. Hauck et al. compare crossbar, hierarchical crossbar, and mesh interconnect topologies for multi-FPGA systems, and demonstrate that the topology has a great effect on the area and delay of the resulting system [11].

The MOGAC hardware/software cosynthesis system [4] partitions and schedules task graphs for embedded systems using an adaptive multiobjective genetic algorithm. It models both bus and point-to-point communication links, as well as a large number of parameters associated with the processing elements including price, pin count, idle power consumption, peak power consumption, and power efficiency. A *link connectivity string* encodes both the type of each link and the interconnect topology as part of a candidate solution for the genetic algorithm. A simple list scheduling heuristic is used. Compared to the interconnect synthesis algorithm presented in our work, MOGAC optimizes a broader range of system parameters with a simpler scheduling algorithm and less focus on optimizing the interconnect topology. In contrast, our techniques are based on scheduling and synthesis strategies that emphasize exploiting the targeted class of point-to-point architectures.

2.3 Scheduling for Arbitrarily Connected Systems

The problem of scheduling parallel tasks with a given precedence relationship onto a multiprocessor architecture is known to be NP-complete [12]. The problem of scheduling onto a set of heterogeneous processors is more complicated than the case for homogeneous processors because the task execution times are dependent on which processor executes a given task. A vast range of scheduling heuristics for task graphs has been developed (e.g., Sriram [5] presents a review of several representative approaches).

The heuristics generally fall into the categories of priority-based list scheduling [13], [14], critical-path-based [15], cluster-based [16], [17], or task duplication-based scheduling [18], [19], [20].

In list scheduling, a priority list $\mathcal{L}$ of tasks is constructed. The priority list $\mathcal{L}$ is a linear ordering $(\nu_1, \nu_2, \ldots, \nu_{|V|})$ of the tasks in the task graph $G = (V,E)$ such that for any pair of distinct tasks ν_i and ν_j, ν_i is to be given higher scheduling priority than ν_j if and only if $i < j$. Each task is mapped to an available processor as soon as it becomes the highest-priority task according to $\mathcal{L}$ among all tasks that are ready. This process is repeated until all tasks are scheduled. Cluster-based heuristics divide the tasks into a set of clusters and assign the clusters to processors, so as to minimize interprocessor communication (IPC) costs.

IPC costs force a trade off between the degree of parallelism utilized and the communication overhead required. In order to address this trade off, one must consider the task grain size, the amount of parallelism in the graph, the processor interconnect topology, and IPC costs. The majority of algorithms that take IPC into account assume the processors to be fully-connected. No attention is paid to link contention or routing strategies used for communication.

One algorithm that does consider link contention and communication routing is the *Dynamic Level Scheduling* (DLS) algorithm presented by Sih and Lee [21]. DLS is a priority-based heuristic that addresses the processor mapping problem and the IPC traffic scheduling problem concurrently. We will present a DLS scheduling algorithm modified for arbitrary interconnect topology in Section 4.4.

Another such algorithm is the *Bubble Scheduling and Allocation* (BSA) algorithm [22]. In the BSA algorithm, the tasks are not fixed on one single list throughout the entire scheduling process. Initially, the tasks are all scheduled on a single processor. Then, each task is considered in turn for possible migration to the neighbor processors. Although both the DLS and BSA algorithms consider link contention and communication routing, they cannot handle the deadlock situations explained in this paper that arise with general topology graphs (deadlock is defined in Section 4 and the topology graph is defined in Section 3).

None of these heuristics take communication hop limits into account. However, they all can be modified as shown in this paper to account for these constraints. As shown later, such connectivity constraints can cause scheduling techniques to deadlock. One contribution of this paper is to develop a general framework for extending existing scheduling approaches to avoid deadlock, and to operate efficiently in the presence of connectivity constraints. We will apply this framework to jointly streamline the communication network and task graph mapping for a given application in Section 5. This framework can be used both for minimum-cost dedicated implementations, and for reconfigurable networks, where the goal is to reduce power consumption.

2.4 Optical Interconnects

Optical interconnects have been proposed as one solution to problems with scaling electrical interconnects on chip [23]. Optics could allow precise clock distribution, enable larger synchronization zones, higher bandwidth and density of long interconnections, and reduction of power consumption.

A number of algorithms have been developed for Optical Transpose Interconnection Systems (OTIS) [24] for general-purpose computation. In the OTIS architecture, processors are divided into groups where electronic interconnects are used to connect processors within the same group, while optical interconnects are used for intergroup communication. A number of interconnect topologies can be produced, including OTIS-Mesh, OTIS-Hypercube, and OTIS-Butterfly. Specific routing and broadcasting algorithms have been developed for these topologies. Our work is different from this work because we seek to synthesize an efficient schedule and interconnection network that is specific to a given application.

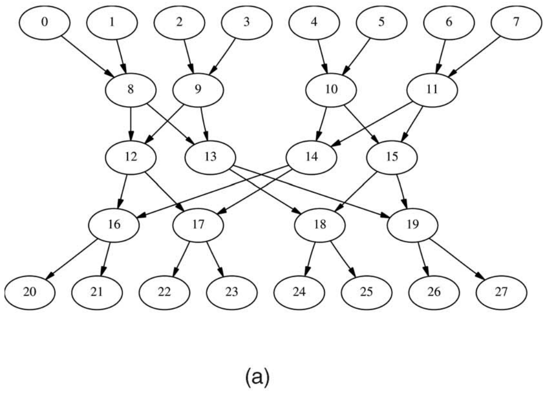
In optically connected systems, the power consumption of communication is relatively independent of distance, and largely dependent instead on the number of electrical-to-optical conversions that must be performed. Thus, it is advantageous to configure multiprocessor schedules in such a way that multihop communication is avoided, or limited to some maximum number of hops per communication operation, and the relative abundance of communication links is used instead to achieve the required communication flexibility. We will also see that this distance independence leads to simpler algorithms for interconnect synthesis.

3 Connection Topologies

Irregular interconnection patterns arise naturally when scheduling task graphs under the restriction of single-hop communication. A simple example of an irregular interconnection network is shown in Fig. 1. With four processors, two possible topologies are shown in Fig. 1a and Fig. 1b. The topology in Fig. 1a has a regular interconnection pattern, with each processor connected to two others. The topology in Fig. 1b is irregular, with one processor having degree six and the others having degree two. The topology in Fig. 1b allows a single-hop schedule, since all required communication can take place with only one hop. In the topology in Fig. 1a, two hops are required for communication from task A to task D and from D to E. The topology in Fig. 1b requires fewer link resources (six links versus eight links).

In a fully-connected interconnect topology (with a link between every pair of processors), all communication can be single-hop. However, for many applications, a fully-connected topology wastes resources on unused links, and fully-connected topologies do not scale well to large numbers of processors. One way to see this is to schedule a task graph on a fully-connected topology, then remove the unused links. We observe that the resulting topology often displays a highly irregular structure. For example, Fig. 2a shows the task graph for an FFT application [25]. This application was scheduled on eight processors using the DLS algorithm [26]. In Fig. 2b, we remove the unused links and show the resulting interconnect topology.

We define a topology graph $T(P, L)$ such that the nodes of T correspond to the "processors" P in the architecture and the edges L in T correspond to direct physical communication links between the processors. We define the set of all processors as P and label the processors $\{p_1, p_2, \ldots, p_{|P|}\}$. Then, T contains an edge (p_i, p_j) iff the interconnection network provides a direct (single-hop) communication link from p_i to p_j. If l is an edge in T, we say that $\mathrm{src}(l)$ is the *source* node of l, $\mathrm{snk}(l)$ is the *sink* node of l, l is *directed* from $\mathrm{src}(l)$ to $\mathrm{snk}(l)$, l is an *output edge* of $\mathrm{src}(l)$, and l is an *input edge* of $\mathrm{snk}(l)$. We assign edge weights $\mathrm{linkDelay}(l)$ to $l \in L$ which represent the propagation time for signals between $\mathrm{src}(l)$ and $\mathrm{snk}(l)$. For NoC networks such as mesh/torus, all links have the same propagation time. Also, for optical interconnects, all links have the same propagation time. In this case, T has equal edge weights. In more general electrical interconnection networks the propagation time can be determined by place-and-route methods. We denote the *degree* of a processor by the number



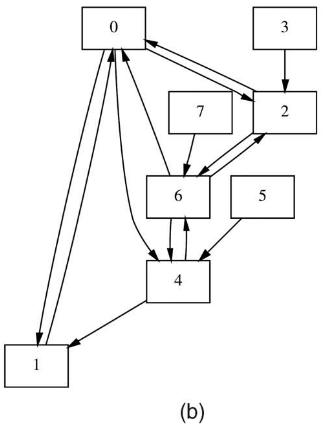


Fig. 2. Single-hop processor topology for DLS schedule of FFT1 application. (a) FFT1 task graph and (b) single-hop topology for FFT1.

of incident (physical) communication links. The degree of a node p in T is equal to the sum of the number of input and output edges of p. For example, each processor in a fully-connected system with $|P|$ processors has degree $2(|P|-1)$ (two links—one incoming and one outgoing—to each other processor). Furthermore, a *path* in $T(P,L)$ is a nonempty sequence $l_1, l_2, l_3, \ldots \in L$ such that $\mathrm{snk}(l_1) = \mathrm{src}(l_2), \mathrm{snk}(l_2) = \mathrm{src}(l_3), \ldots$ whose *path length* equals the number of edges in the sequence. T is said to be *strongly connected* if for each pair of distinct nodes (p_1, p_2) there is a path directed from p_1 to p_2 *and* there is a path directed from p_2 to p_1.

In addition, we will define T to be *strongly connected in h hops* if for each pair of distinct nodes (p_1, p_2) there is a path directed from p_1 to p_2 with path length less than or equal to h *and* there is a path directed from p_2 to p_1 with path length less than or equal to h.

We define two functions, $Rf^1(p,T)$ and $Rb^1(p,T)$, which are properties of the topology graph T. $Rf^1(p,T)$ returns a set of processors A such that for all $a \in A$ there exists a path in T from processor p to processor a with path length less than or equal to one. For example, in Fig. 4, $Rf^1(2,T) = \{1,2,3\}$. $Rb^1(p,T)$ returns a set of processors B such that for all $b \in B$ there exists a path in T from b to p with path length less than or equal to one. In Fig. 4, $Rb^1(2,T) = \{2\}$ and $Rb^1(3,T) = \{2,3\}$. We also define functions $Rf^h(p,T)$ and $Rb^h(p,T)$ that return sets of processors that are h hops away from p.

$$Rf^2(p,T) = Rf^1(Rf^1(p,T)), \quad Rb^2(p,T) = Rb^1(Rb^1(p,T))$$
$$Rf^h(p,T) = Rf^1(Rf^{h-1}(p,T)), \quad Rb^h(p,T) = Rb^1(Rb^{h-1}(p,T)).$$

In an arbitrary network, the relative variation in the degrees among different processors gives a measure of the level of irregularity of the associated interconnection pattern. For example, in the mapping of Fig. 2b, processors 0 and 6 have degree six, while processors 3 and 5 have degree one. The topology graph of Fig. 2b contains $|L| = 14$ links and $|P| = 8$ processors. The average degree is $1/|P|\sum_{p \in P}$ degree $(p) = 2|L|/|P| = 3.5$. The maximum degree is 6. For a fully connected graph the average and maximum degrees both equal $2(|P|-1) = 14$. For a regular topology in which all nodes are identical, the average and maximum degrees are equal.

This trend of highly irregular connection requirements occurs over a wide variety of task graph structures. To illustrate this, we scheduled 100 real and synthetic benchmark task graphs on different numbers of processors using the DLS scheduling algorithm for a fully connected topology graph. We then removed the unused links and examined the resulting topology graph. Fig. 3 plots the average and maximum degree for nodes in these topology graphs for different $|P|$. We normalized these values to those for a fully connected graph—i.e., the normalized average degree for the topology graph resulting from benchmark k is given by $|L_k|/|P|(|P|-1)$ and the average over all benchmarks is $1/100\sum_{k=1}^{100}$ $|L_k|/|P|(|P|-1)$. The large separation between average and maximum degrees shows that irregular topologies arise naturally for many applications. In addition, from the low values of normalized average node degree, we see that in a fully connected topology, many of the links would go unused. The synthetic benchmarks used in these experiments were generated using the graph generation techniques of Sih [26], which are designed to construct task graphs that resemble the dataflow structures found in DSP applications.

As motivated earlier, when developing automated mapping tools for SoC, we have several design constraints. It is desirable to map the application onto the architecture

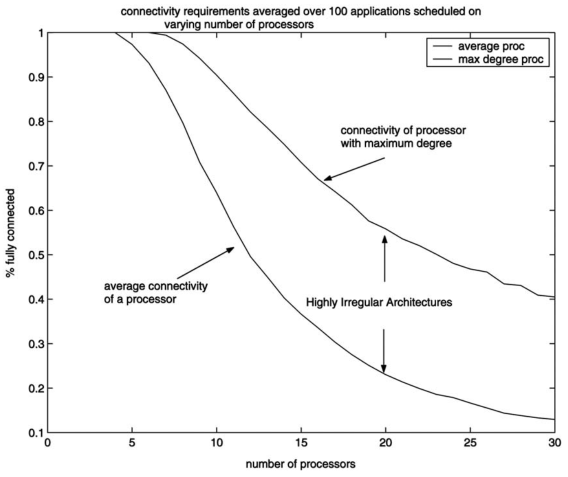
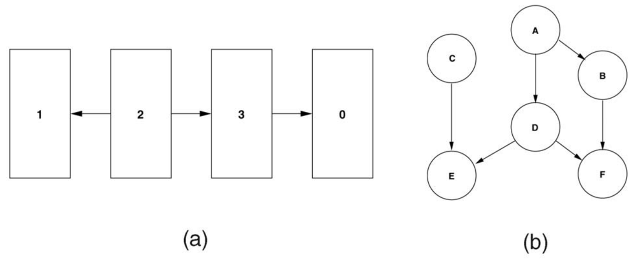


Fig. 3. Connectivity requirements of 100 benchmark applications.



(c)	(d)	(e)
Partial Schedule:	Partial Schedule:	Partial Schedule:
A on processor 2	A on processor 2	A on processor 2
B on processor 1	B on processor 2	B on processor 3
Constraint sets:	Constraint sets:	Constraint sets:
$\Psi[A] = \{2\}$	$\Psi[A] = \{2\}$	$\Psi[A] = \{2\}$
$\Psi[B] = \{1\}$	$\Psi[B] = \{2\}$	$\Psi[B] = \{3\}$
$\Psi[F] = \{1\}$	$\Psi[F] = \{1, 2, 3\}$	$\Psi[F] = \{0, 3\}$
$\Psi[D] = \{1, 2\}$	$\Psi[D] = \{1, 2, 3\}$	$\Psi[D] = \{1, 2, 3\}$
$\Psi[E] = \{1, 2, 3\}$	$\Psi[E] = \{0, 1, 2, 3\}$	$\Psi[E] = \{0, 1, 2, 3\}$
$\Psi[C] = \{1, 2, 3\}$	$\Psi[C] = \{0, 1, 2, 3\}$	$\Psi[C] = \{0, 1, 2, 3\}$
flexibility = 11/24	flexibility = 16/24	flexibility = 15/24

Fig. 4. Example requiring constraint information propagating both forward and backward. (a) Processor connection, (b) application graph, and (c), (d), and (e) depict constraint sets and flexibility metrics for different partial schedules.

without requiring multihop communication, while satisfying constraints on system throughput and latency. We also have limits on the maximum I/O fanout and degree of a single processor. In order to conserve space and power, we would also like to minimize the total number of communication links.

4 Scheduling and Deadlock

A schedule for a task graph $G(V, E)$ is a function $\tau : V \rightarrow \mathbb{Z}^+$ that satisfies for all edges $e = (\nu_i, \nu_j) \in E : \tau(\nu_j) \geq \tau(\nu_i) + \text{exec}(\nu_i)$, where $\tau(\nu_i)$ is the start time of node $\nu_i \in V$, the execution time of ν_i is $\text{exec}(\nu_i)$, and the finish time of ν_i is $\tau(\nu_i) + \text{exec}(\nu_i)$. The *schedule makespan* is defined as the maximum finish time for all $\nu \in V$. In addition, we define $\sigma(\nu)$ as the processor assignment for ν in the schedule.

Due to the desirability of low-hop communications as well as irregular interconnect topologies, as motivated in Section 3, it is important during codesign to employ scheduling techniques that carefully take into account the connectivity of candidate interconnection patterns and that can place an upper bound on the number of hops. In this section, we describe a scheduling algorithm that explicitly handles both of these—it inputs a hop constraint and an arbitrary topology graph T in addition to a task graph G. In Section 5, we will utilize this scheduling algorithm within a larger optimization algorithm that jointly optimizes both T and the schedule.

For topology graphs that are not strongly connected, each processor p can only send data to a subset of P, since there are pairs of processors (p_i, p_j) for which no path exists. That is, there exists some p_i such that $Rf^h(p_i, T) \neq P$ for any number of hops h. Likewise, each processor p can only receive data from a subset of P so there exists some p_j such that $Rb^h(p_j, T) \neq P$ for any h. Consider a task graph G, two tasks ν_1 and ν_2 in G that have been scheduled on processors p_1 and p_2, respectively, and a third task ν_3 that receives data from ν_1 and ν_2 (i.e., there exist edges (ν_1, ν_3) and (ν_2, ν_3) in G). If, during the scheduling algorithm, $Rf^h(p_1, T) \cap Rf^h(p_2, T) = \emptyset$ for all h, then we define the schedule to be *deadlocked*[1] since there is no processor on which to place ν_3 such that it can receive data from both ν_1 and ν_2.

If we impose a communication hop constraint $0 \leq h \leq n$ (n-hop schedule), then schedule deadlock becomes much more likely and possible even for strongly connected topology graphs. That is, if T is strongly connected but not strongly connected in n hops, we can encounter the same deadlock condition. Deadlock occurs when one task cannot communicate data (either because there is no path between the tasks, or because the hop constraint would be exceeded on all available paths) to its predecessor in the task graph.

We will show in Section 4.1 that scheduling a task in G can potentially cause a deadlock condition in any other part of G. It is therefore not feasible to simply "look ahead" a given number of moves in the scheduling algorithm to avoid deadlock. We must instead calculate the degree to which a given scheduling move constrains the processor choices for all other tasks in the graph.

4.1 Connectivity and Scheduling Flexibility

We define a *feasible set* of processors $\Psi[\nu]$ for a task ν in $G(V, E)$ as the largest subset of P on which ν can be scheduled without deadlock. We would like to have an algorithm to determine the feasible set of processors $\Psi[\nu]$ for all $\nu \in V$. In general, a constraint imposed on one task (scheduling it on a processor) may cause $\Psi[\nu]$ to be updated for all $\nu \in V$. This update consists of choosing a subset of the set $\Psi[\nu]$ that existed before the constraint—new members are never added.

We define the *communication flexibility* (or simply *flexibility* for short) of the system at any point during the scheduling process as the sum of the sizes of the sets $\Psi[\nu]$ for all $\nu \in V$ normalized over all processors and tasks:

$$\text{flexibility} = \frac{1}{|P||V|} \sum_{\nu \in V} |\Psi[\nu]|, \quad (1)$$

where $|P|$ is the number of processors in the system and $|V|$ is the number of tasks in the application. The flexibility (a value between zero and one) gives some measure of the degree of constraint imposed on all tasks by a given scheduling move—higher flexibility implies less constraint.

Fig. 4 depicts a simple example of a task graph with six tasks being scheduled on four processors. Partial schedules corresponding to scheduling task A on processor 2 and task B on processors 1, 2, or 3 are shown in Figs. 4c, 4d, and 4e. Scheduling task B has an effect on the feasible processor sets for tasks C, D, E, and F. Fig. 4c shows the constraint sets Ψ for each task after scheduling B on processor 1. We observe that processor 0 is infeasible for task C. Scheduling task C on processor 0 would cause deadlock no matter where the remaining tasks D, E, and F were placed. After task B is

1. Our definition of schedule deadlock is different from the stricter traditional definition—we only require that we reach a condition where it is not possible to place a task on any processor without violating the hop constraint.

```
Algorithm 6.1: BFSFORWARD($G, T, s, n$, endNodes, bottomNodes, $\Psi$)
input: application graph G
input: topology graph T
input: node s in G being considered
input: set of discovered endNodes
input: maximum hop communication allowed n
output: stack of newly discovered bottomNodes
input/output: array $\Psi$ of sets of feasible processors for each node in $G$
local variables: queue of nodes Q, array dist of distances for each node
local variables: set R of processor numbers, application graph nodes w, v
for all $w \in G$
  do {dist[$w$] = $-1$   dist[$s$] = 0
$Q \leftarrow \{s\}$
while ($Q \neq \emptyset$)
  do
    $v$ = head($Q$)
    for all $w \in$ Adj[$v$]
      do
        if dist[$w$] < 0
          then
            ENQUEUE($Q, w$)
            dist[$w$] = dist[$v$] + 1
            $\Psi[w] = \Psi[w] \cap Rf^n(\Psi[v], T)$
            if $\Psi[w] = \emptyset$
              then {return ($-1$)
            if outdegree($w$) = 0
              then {PUSH($w$, bottomNodes)
return (0)
```

(a)

```
Algorithm 6.1: BFSBACKWARD($\hat{G}, H, T, s, n$, endNodes, topNodes, $\Psi$)
input: edge-reversed application graph $\hat{\mathbf{G}}$, topology graph T
comment: for every node $u \in \hat{G}$, $v = \hat{G}[u]$ gives corresponding node in $G$
input: array of nodes H
comment: for any node $v \in G$, $u = H[v]$ references corresponding node in $\hat{G}$
input: node s in $G$ being considered
input: set of discovered endNodes
input: maximum hop communication n allowed
output: stack of newly discovered topNodes
input/output: array $\Psi$ of sets of feasible processors for each node in $G$
local variables: queue of nodes Q, array dist of distances for each node
local variables: set R of processor numbers, application graph nodes w, v, $\hat{\mathbf{s}}$
local variables: nodes in $\hat{G}$: $\hat{\mathbf{w}}$, $\hat{\mathbf{v}}$
for all $w \in \hat{G}$
  do {dist[$w$] = $-1$
$\hat{s} = H[s]$
dist[$\hat{s}$] = 0
$Q \leftarrow \{\hat{s}\}$
while ($Q \neq \emptyset$)
  do
    $v$ = head($Q$)
    for all $w \in$ Adj[$v$]
      do
        if dist[$w$] < 0
          then
            ENQUEUE($Q, w$)
            $\hat{w} = H[w]$
            $\hat{v} = H[v]$
            dist[$w$] = dist[$v$] + 1
            $\Psi[\hat{w}] = \Psi[\hat{w}] \cap Rb^n(\Psi[\hat{v}], T)$
            if $\Psi[\hat{w}] = \emptyset$
              then {return ($-1$)
            if outdegree($w$) = 0
              then {PUSH($\hat{w}$, topNodes)
return (0)
```

(b)

Fig. 5. Pseudocode for propagating constraint information. (a) bfsForward and (b) bfsBackward.

scheduled on processor 1, processor 0 becomes infeasible for task C, since scheduling task C on processor 0 confines task E to processor 0. Task F is confined to processor 1 since B is scheduled on 1. Task D sends data to both E and F, and there is no processor which can communicate with both processors 0 and 1 in a single hop.

Traditional list scheduling algorithms cannot detect these potential deadlock situations. In this case, task C would be considered a "start node" with no predecessors. There is no path between task B and task C in the task graph, so a technique that looks "forward" to detect deadlock will not work. Instead, we must propagate the processor constraint information to all other tasks in the graph. Although the example given in Fig. 4 is fairly simple, for more complicated topologies one cannot easily detect these deadlock conditions by inspection, since scheduling one task in the graph may possibly constrain any other task in the graph.

The flexibility measure for the partial schedule in Fig. 4c is equal to $11/24$. If B is instead scheduled on processor 2 (Fig. 4d), the processor constraint sets are different for the remaining unscheduled tasks, and the flexibility measure is $16/24$. Likewise, the flexibility measure for the partial schedule in Fig. 4e is $15/24$. A scheduling algorithm that utilizes the flexibility metric might choose the scheduling move in Fig. 4d since this choice constrains the remaining scheduling choices the least.

Our algorithm works by propagating constraint information forward and backward through the task graph G. An input n specifies the maximum number of hops allowed for two processors to communicate with each other. In our experiments, we concentrate on single-hop communication, where $n = 1$. First, an edge-reversed copy $\hat{G}$ of the task graph G is created. When making a scheduling move (introducing a new constraint at a task s), the constraint information is propagated forward using breadth first search from s through G. When an *endnode* (task with no successors) is discovered during the forward phase for the first time, it is added to a stack (named **endNodes**).

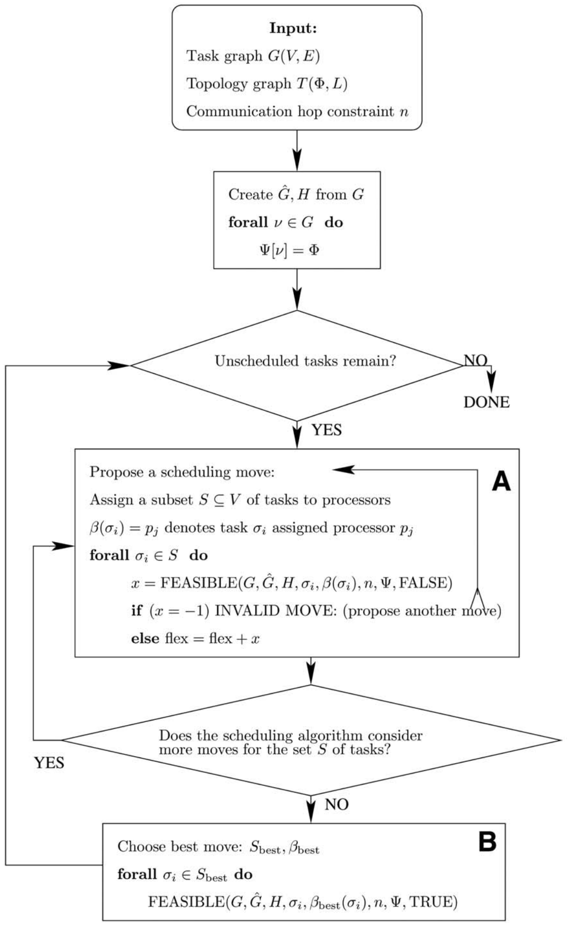
At the end of the forward phase, the backward phase begins. Each endnode is removed from the stack and the constraint information is propagated backward by performing breadth first search from the endnodes through $\hat{G}$. While propagating backward, newly discovered endnodes of $\hat{G}$ are added to a second stack. These endnodes are removed from the stack, and search continues in the forward direction. The process continues until there are no newly found endnodes.

We define the functions **bfsForward**() (Fig. 5a) and **bfsBackward**() (Fig. 5b) which use breadth first search to propagate constraint information for a task s in a graph G in the forward and backward direction.

Finally, the **feasible**() function described in Fig. 6a returns an integer equal to the sum of the sizes of the constraint sets for all nodes in the task graph G when scheduling a task G on a processor p, given an input n corresponding to the maximum number of communication hops allowed. If s is not feasible on p, the function returns -1.

```
Algorithm 6.1: FEASIBLE($G, \hat{G}, H, T, s, p, n, \Psi$, commit)
 input: application graph G, topology graph T
 input: edge-reversed application graph $\hat{G}$
 input: array of nodes H
 input: node s in $G$ being considered
 input: processor p being considered
 input: maximum hop communication n allowed
 input: boolean value commit determines if changes to $F$ are saved
 input/output: array $\Psi$ of sets of feasible processors for each node in $G$
 local variables: $\Psi_{local}$ (local copy of $\Psi$), set of discovered endNodes
 local variables: stack of nodes topNodes, application graph nodes v, $v_f$, $b_f$
 local variables: sets of processor numbers f and r
 if $p \notin \Psi[s]$
   then {return $(-1)$
 $\Psi_{local} = \Psi$
 $\Psi_{local}[s] = \{p\}$
 PUSH(topNodes, $s$)
 while topNodes $\neq \emptyset$
   do {
     while topNodes $\neq \emptyset$
       do {
         POP(topNodes, $v_f$)
         INSERT(endNodes, $v_f$)
         $x$ = BFSFORWARD($G, s, n$, endNodes, bottomNodes, $\Psi_{local}$)
         if $x = -1$
           then {return $(-1)$
     while bottomNodes $\neq \emptyset$
       do {
         POP(bottomNodes, $v_b$)
         INSERT(endNodes, $v_b$)
         $x$ = BFSBACKWARD($\hat{G}, H, s, n$, endNodes, topNodes, $\Psi_{local}$)
         if $x = -1$
           then {return $(-1)$
 if commit
   then {$\Psi = \Psi_{local}$
 flexibility = 0
 for all $v \in G$
   do {flexibility = flexibility + size($\Psi_{local}$)
 return (flexibility)
```

(a)


Input:
Task graph $G(V,E)$
Topology graph $T(\Phi, L)$
Communication hop constraint n
Create $\hat{G}, H$ from G
forall $\nu \in G$ do
$\Psi[\nu] = \Phi$
Unscheduled tasks remain?
NO
DONE
YES
Propose a scheduling move:
Assign a subset $S \subseteq V$ of tasks to processors
$\beta(\sigma_i) = p_j$ denotes task σ_i assigned processor p_j
forall $\sigma_i \in S$ do
x = FEASIBLE($G, \hat{G}, H, \sigma_i, \beta(\sigma_i), n, \Psi$, FALSE)
if $(x = -1)$ INVALID MOVE: (propose another move)
else flex = flex + x
A
Does the scheduling algorithm consider more moves for the set S of tasks?
YES
NO
Choose best move: S_{best}, β_{best}
forall $\sigma_i \in S_{best}$ do
FEASIBLE($G, \hat{G}, H, \sigma_i, \beta_{best}(\sigma_i), n, \Psi$, TRUE)
B

(b)

Fig. 6. Function FEASIBLE() determines feasibility and flexibility (degree of constraint) for scheduling task s on processor p. It can be used to allow existing scheduling algorithms to work with arbitrary interconnection topologies. (a) Feasible function and (b) scheduling using Feasible function.

4.2 Complexity of the Constraint Algorithm

The forward breadth first search function **bfsForward** is called once for the task being considered, and once for each task in G with no predecessors (endnode in $\hat{G}$). The backward breadth first search function **bfsBackward** is called once for each task in G with no successors (endnode in G). The complexity of breadth first search is $O(|V| + |E|)$ for a graph with $|V|$ nodes and $|E|$ edges. The **bfsForward** and **bfsBackward** functions require a set intersection of two sets of size $O(|P|)$, where $|P|$ is the number of processors in the system. This set intersection has complexity $O(|P| \log |P|)$. The overall complexity is

$$O(|V|(|V| + |E|)|P| \log |P|). \quad (2)$$

This is a reasonable complexity figure in the embedded systems domain, where compile/synthesis time tolerance is significantly higher compared to general-purpose computation (e.g., see [27]).

For topology graphs that are strongly connected in δ hops, the breadth first searches do not need to proceed for distances further than δ where δ is a property of the topology graph. In most cases, $\delta \ll (|V| + |E|)$. In this case, the complexity is only $O(|V| \cdot \delta \cdot |P| \cdot \log |P|)$.

4.3 Incorporating Feasibility and Flexibility into Scheduling

A wide range of existing scheduling algorithms can easily be adapted to produce single-hop (or n-hop) schedules by incorporating our constraint algorithm. This is advantageous because it allows us to leverage a large library of useful scheduling techniques.

The feasibility/flexibility framework, introduced in Section 4.1, can be integrated with a broad class of scheduling algorithms that iteratively apply scheduling moves or groups of moves, as illustrated in Fig. 6b. List scheduling algorithms are one subclass of this class of scheduling algorithms. For list scheduling, this amounts to restricting the set of candidate processors to include only those that are feasible at the given scheduling step, and by taking flexibility into account in designing the priority metric through which tasks are ordered.

Given a task graph $G(V,E)$ where V is the set of all tasks and a topology graph $T(P,L)$ where P is the set of all processors, we define a *scheduling move* as a set of processor assignments for a subset $S \subseteq V$ such that for each task $\nu_i \in S$, $\beta(\nu_i) = p_j$ denotes ν_i is to be scheduled on processor $p_j \in P$. We begin with all processors feasible for each task ν so that $\Psi[\nu] = P$ for all $\nu_i \in V$. Then, the scheduling move constrains the tasks in set S so that $\Psi[\nu_i] = \beta(\nu_i)$ for all $\nu_i \in S$. This constraint information is propagated to all the other constraint sets in both forward and backward direction using the FEASIBLE function. If one or more of the processor assignments $\beta(\nu_i)$ would result in a deadlock situation, the scheduling move must be rejected (box **A** in Fig. 6b). In addition, the decision on which scheduling move to accept might be influenced by the flexibility metric (box **B** in Fig. 6b).

In the context of limited-hop communication across arbitrary interconnection patterns, the incorporation of feasibility considerations is required (to avoid scheduler deadlock, as discussed in Section 4.1), while incorporation of flexibility is optional. Furthermore, there are many possible ways to consider flexibility in the task prioritizing process. We show in Section 4.4 that even simple techniques for incorporating flexibility information can lead to large performance improvements for a targeted class of architectures.

4.4 Scheduling Experiments Using Flexibility

As mentioned earlier, our scheduling technique operates in conjunction with a given list scheduling strategy. In our experiments, we employed the DLS algorithm [21] as the underlying list scheduling strategy, although, as explained in Section 4.1, any list scheduling algorithm could have been used.

In the task graph specification, each edge $e = (\nu_i, \nu_j)$ is assigned a weight $\mathrm{IPC}(e)$ representing the interprocessor communication cost. This is proportional to the number of bits that must be communicated from ν_i (scheduled on processor $\sigma(\nu_i)$) to ν_j (scheduled on $\sigma(\nu_j)$) if $\sigma(\nu_i) \neq \sigma(\nu_j)$. The DLS algorithm chooses a path in the topology graph $T(P, L)$ with path length $h(e)$ (obeying the hop constraint) from $\sigma(\nu_i)$ to $\sigma(\nu_j)$ denoted by $\mathrm{route}(e) = l_1 l_2 \dots l_{h(e)}$ with $l \in L$. The communication latency for e is then given by

$$\mathrm{latency}(\mathrm{route}(e)) = \sum_{l \in \mathrm{route}(e)} \mathrm{IPC}(e)\mathrm{linkDelay}(l). \quad (3)$$

If $\mathrm{linkDelay}$ is a constant, as in some NoC architectures or for optical interconnects, the latency of $\mathrm{route}(e)$ is given by $\mathrm{IPC}(e) \cdot h(e) \cdot \mathrm{linkDelay}$. If $\sigma(\nu_i) = \sigma(\nu_j)$, $h(e) = 0$ and the IPC cost is zero.

We examined a set of DSP application benchmarks and scheduled them using two different scheduling modes, one that incorporates only feasibility information (to avoid deadlock), and another that takes both feasibility and flexibility into account. We refer to these as the *feasibility-only* and *feasibility-flexibility* modes, respectively. To evaluate the performance across a range of connectivity levels, we scheduled the applications onto networks with varying degrees of connectivity.

In the feasibility-only mode, the processor p considered for a given task ν at each scheduling step was restricted to be in the feasible set $\Psi[\nu]$ for ν, as described in Section 4.1, and no modification was made to the task prioritizing metric of the underlying list scheduling strategy (DLS).

In the feasibility-flexibility mode, the processor P considered at each scheduling step was again restricted to be in the feasible set for ν; however, whenever two processor assignments for ν resulted in equal priority levels in the original DLS algorithm, additional priority was given to the assignment that resulted in a higher value of flexibility. In other words, priority was given to assignments that offered greater flexibility for future scheduling decisions.

For each application, we chose a number $|P|$ of processors, then generated a fully connected topology graph T with $|P|(|P|-1)$ links. We scheduled the task graph using both feasibility-only (resulting in makespan M_{noflex}) and feasibility-flexibility modes (resulting in makespan M_{flex}) given this topology. Next, we removed one link from T at random, and again scheduled the application using both scheduling modes on the new T. We continued this process of removing links until no links remained, resulting with all the tasks scheduled on a single processor. The result of this experiment for an FFT benchmark is shown in Fig. 7a, where we have normalized the makespan values to those for a uniprocessor (i.e., a value of 0.25 indicates 4 times faster running on multiple processors than on one processor).

We calculate the average makespan improvement, $\bar{\mathcal{I}}_M$, of the feasibility-flexibility mode over the feasibility-only mode by averaging over all $|P|(|P|-1)$ topology graphs generated:

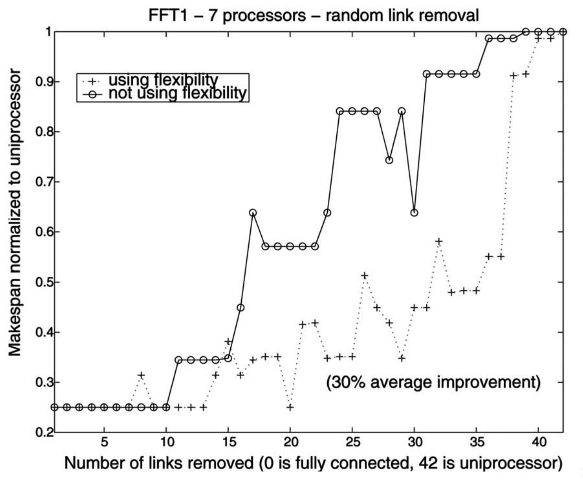
$$\bar{\mathcal{I}}_M = 1/|P|(|P|-1) \sum_{i=1}^{|P|(|P|-1)} \frac{M_{\mathrm{flex}}(T_i) - M_{\mathrm{noflex}}(T_i)}{M_{\mathrm{noflex}}(T_i)}. \quad (4)$$

If we compare the average makespan for the schedules generated by feasibility-flexibility mode (the top curve in Fig. 7a) with the average makespan of the schedules generated without incorporating flexibility (the bottom curve in Fig. 7a), we see a 30 percent average improvement ((4)) when the scheduling algorithm incorporates the flexibility metric. In Fig. 7b, we show the normalized makespan for a 3×3 electrical mesh topology as a function of the ratio of average task ($\nu \in V$) execution time to communication time (edge weight of $l \in L$).

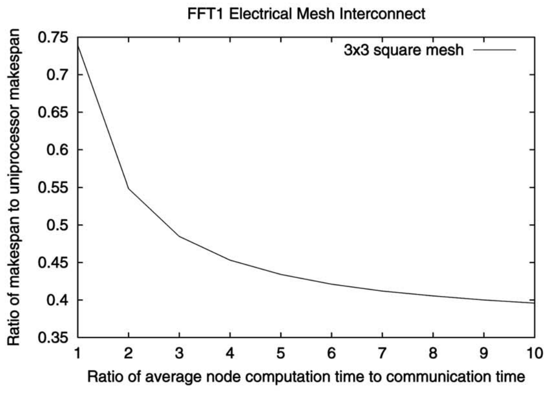
Table 1 summarizes the average makespan improvement $\bar{\mathcal{I}}_M$ ((4)) for several other DSP applications. We performed experiments with the following task graphs: FFT1, Irr, FFT3, Karp10, Qmf4, Laplace, Sum1, and NN16-3-4. The FFT graphs are different implementations of the fast Fourier transform from Kahn [25] and contain 28 nodes each. Karp10 refers to the Karplus-Strong music synthesis algorithm with 10 voices (21 nodes), and Qmf4 is a quadrature mirror filter bank with 14 nodes. Laplace is a Laplace transform, Irr is an adaptation of a physics algorithm, and sum1 is an upside down binary tree representing the sum of products computation. A neural network classifier algorithm with 16 input nodes, 3 intermediate layers, and 4 output nodes labeled NN6-3-4 was also tested.

4.5 Power Reduction with Low-Hop Communication

As mentioned earlier, it is advantageous for several reasons to limit interprocessor communication to a low number of hops (denoted as n). In order to investigate the effect of hop limits on power consumption, we scheduled the benchmark applications using our modified scheduling technique, which takes n as an input. We scheduled the benchmarks with $n = 1$ and $n = 3$ and compared the *communication*



(a)



(b)

Fig. 7. Makespan for FFT normalized to uniprocessor on (a) random topologies with and without flexibility metric and (b) 3×3 mesh using flexibility metric. (a) Normalized makespan on random topologies with and without flexibility and (b) normalized makespan on 3×3 mesh for different computation-to-communication ratios.

energy (energy required to communicate data between different processors) required. For a task graph $G(V, E)$ scheduled on a topology graph $T(P, L)$, we calculate the communication energy by summing the energy required for each data communication between processors. This includes intermediate hops for multiple-hop transfers. The communication energy $\mathcal{E}$ is given by

$$\mathcal{E}(G,T,n) = \sum_{e \in E}\left(\mathrm{IPC}(e) \sum_{l \in \mathrm{route}(e)} \epsilon_{\mathrm{bit}}(l)\right), \qquad (5)$$

where the energy required to communicate a bit over a link $l \in L$ is given by ϵ_{bit}, which encapsulates the technology-dependent parameters like wire resistance and capacitance for electrical interconnects and transmitter and receiver power for optical interconnects. In a topology graph where all links consume equal power, the communication energy is given by

$$\mathcal{E}(G,T,n) = \epsilon_{\mathrm{bit}} \sum_{e \in E} \mathrm{IPC}(e) \cdot h(e), \qquad 0 \leq h(e) \leq n. \qquad (6)$$

If $\mathrm{src}(e)$ and $\mathrm{snk}(e)$ are scheduled on the same processor, $h(e) = 0$ and that edge does not contribute to the communication energy. With a three-hop limit, the scheduler is free to choose any communication path that involves three or fewer hops and is thus less constrained in its scheduling choices than with a one-hop limit. Table 1 shows the reduction in the required communication energy, $\bar{\mathcal{R}}_E$, averaged over the different topologies generated,

$$\bar{\mathcal{R}}_E = 1/|P|(|P|-1) \sum_{i=1}^{|P|(|P|-1)} \frac{\mathcal{E}(G,T_i,3) - \mathcal{E}(G,T_i,1)}{\mathcal{E}(G,T_i,3)}, \qquad (7)$$

for single-hop schedules over three-hop schedules for the benchmark applications in the case where ϵ_{bit} is constant. In this case, ϵ_{bit} cancels out of (7).

A trade off exists when imposing a communication hop limit. On the one hand, with more hops allowed, the scheduler is less constrained—the set of moves available to the scheduler at any point using three hops is a superset of the moves available when limited to one hop. On the other hand, for low-hop schedules the average interprocessor communication time is lower. In general, higher hop limits result in the tasks being distributed over a larger set of processors. We have observed (Table 1) that this trade off usually tends to favor slightly better (lower) makespans for higher hop schedules. This comes at the expense of significantly higher communication energy. In two of the benchmarks (Irr and Laplace), the makespan was in fact better when we limited the scheduler to single hops, while the communication energy was lower in all cases.

5 Interconnect Synthesis

In this section, we describe an algorithm for jointly optimizing the task schedule and the interconnect topology for a given application. Our interconnect synthesis method

TABLE 1
Average Makespan Improvement, $\bar{\mathcal{I}}_M$, Using Flexibility; Reduction in Communication Energy, $\bar{\mathcal{R}}_E$, of Single Hop Schedule; Corresponding Makespan Increase Using Single-Hop Schedule

Application	$\|P\|$	$\|V\|$	$\bar{\mathcal{I}}_M(\%)$	$\bar{\mathcal{R}}_E(\%)$	Single-hop makespan increase(%)
FFT1	7	28	30	16	8
Karp10	6	21	26	24	4
Irr	8	41	17	16	(2)
Qmf4	7	14	19	32	3
NN16-3-4	8	71	21	58	2
Sum1	6	15	8	1	4
Laplace	7	16	23	4	(3)
FFT2	7	28	15	12	2

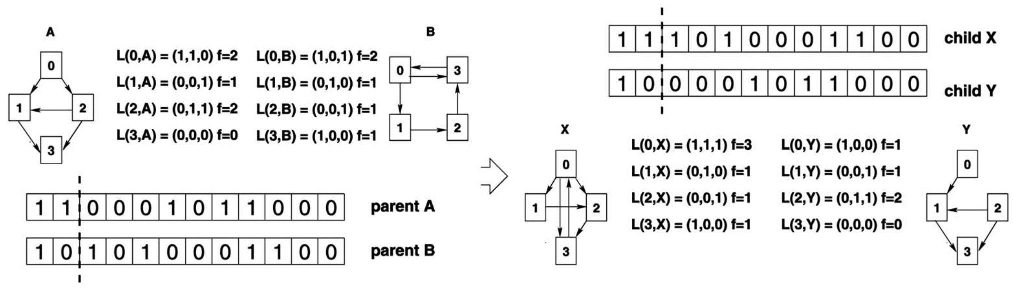


Fig. 8. Crossover operation for link synthesis using the binary string representation for $\vec{l}$. Link fanout constraint is not preserved for child X, where the fanout of processor 0 is $f_{0X} = 3$.

utilizes a genetic algorithm (GA) operating in conjunction with a list scheduling algorithm. Genetic algorithms are capable of both broad search (exploration) and local search (exploitation) of a search space. They are often preferred over gradient search methods because they avoid local minima, and do not require a smooth search space. The scheduling algorithm is a dynamic level scheduling (DLS) algorithm [21] modified for arbitrary interconnection networks, as explained in Section 4.3. The GA produces candidate topology graphs $T(P, L)$. For electrical interconnects, we must perform some type of place-and-route optimization on each T in order to determine the edge weights corresponding to interconnect delays for $l \in L$. The scheduler inputs T, a task graph G, and a hop constraint and produces a schedule with makespan M. The fitness of each solution candidate is determined by M and by two properties of T—the total number of links $|L|$ in T and the maximum fanout $f_{\max}$ (degree of node $p \in P$). Using a single-objective evolutionary algorithm, we can either minimize M for a given $|L|$ and $f_{\max}$ or minimize $|L|$ for a given M and $f_{\max}$. The optimization extensively applies the feasibility/flexibility notions, and associated scheduling mechanisms, discussed in Section 4.

The algorithm applies in principle to both electrical and optical interconnects, although optical interconnects offer a significant simplification. Due to the independence of interconnect delay on length for optical interconnects, we do not need to perform the place-and-route on each candidate T. Our experiments are limited to this simpler case.

5.1 Problem Representation

In our algorithm, the individuals are bit vectors corresponding to a topology graph. The fitness function for a chromosome in our interconnect synthesis algorithm is described by

$$\text{fitness} = M(1 + \mathcal{P}_f + \mathcal{P}_l), \tag{8}$$

where M is the makespan (latency) calculated by the modified DLS algorithm for the interconnect topology of the chromosome, $\mathcal{P}_f$ is a penalty based on violating the fanout constraint $f_{\max}$, and $\mathcal{P}_l$ is a penalty based on violating the maximum link constraint $l_{\max}$.

We define a *link vector* as a bit vector with one entry for each possible interconnection between two processors. For a system with $|P|$ processors, there are $|P|(|P| - 1)$ entries in the link vector. The link vector for a four processor system would be denoted as $\vec{l} = (l_{01}l_{02}l_{03}l_{10}l_{12}l_{13}l_{20}l_{21}l_{23}l_{30}l_{31}l_{32})$, where l_{ij} equals one if there is a connection from processor i to processor j and zero otherwise. We define $l_{ij} \equiv 0$ if $i = j$. We also write $\vec{l}$ as $\vec{l} = (\vec{l}_0\vec{l}_1 \dots \vec{l}_{|P|-1})$, where $\vec{l}_k$ describes the (outgoing) connections for processor k. We will refer to the $\vec{l}_k$ as *processor link vectors*. We define the fanout of processor i by f_i, number of links n_l, and fanout penalty $\mathcal{P}_f$:

$$f_i = \sum_{j=0}^{|P|-1} l_{ij} \doteq \|\vec{l}_i\|, \quad n_l = \sum_{i=0}^{|P|-1} f_i, \quad \mathcal{P}_f = \sum_{i=0}^{|P|-1} \mathcal{P}_i,$$

where $P_i = \max(0, (f_i - f_{\max}))$. The link penalty is given by $\mathcal{P}_l = \max(0, (n_l - l_{\max}))$.

5.2 Fanout Constraints

There is a trade off between chip area and the number of long interconnects in any given SoC architecture. In our model of local regions (or processors) interconnected by long (global) interconnects, the total number of global interconnects is equal to the average fanout of a processor times the number of processors on the chip. Any given SoC architecture will have a fanout constraint for the processors. Therefore, as mentioned above, it is important to have a link synthesis algorithm that can conform to fanout constraints. Our GA is able to incorporate these constraints in a straightforward manner by implementing the initialization, crossover, and mutation operators as described below.

5.3 Crossover and Mutation Operators

We first note that if an individual topology is represented as a binary string, then the typical crossover operations like array one-point crossover or two-point crossover will not preserve the fanout constraint. This is illustrated in Fig. 8, where both parents obey a fanout constraint $f_{\max} = 2$, but processor 0 of child X has fanout $f_{0X} = 3$. This is because the crossover point can be chosen at any point. If we instead choose to represent the topology by the vector representation for $\vec{l}$, fanout constraints are preserved in the crossover operation, since the link vectors for individual processors $\vec{l}_i$ are never altered. The crossover operation only rearranges the relative position of these link vectors. This is illustrated in Fig. 9.

We also must ensure that the initial population obeys the link constraint. The initialization operator generates random processor link vectors which each satisfy the fanout constraint. $|P| - 1$ of these vectors are then concatenated to form the link vector.

The mutation operator simply chooses a random bit in the link vector, and sets its value to zero. This removes a link if one existed at this point. Since the mutation operator only removes links, the fanout constraint is preserved.

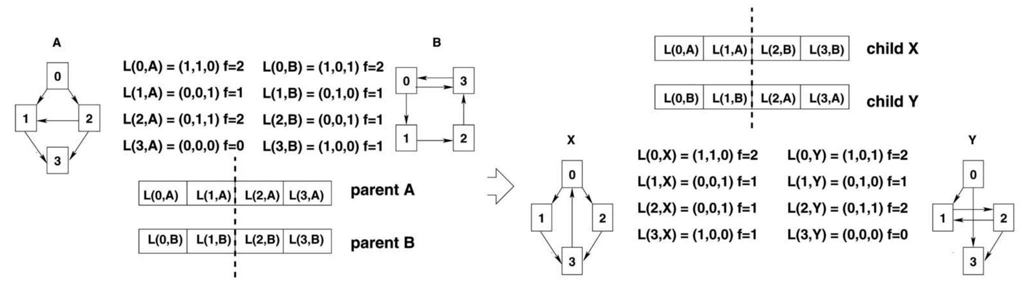


Fig. 9. Crossover operation for link synthesis using the vector representation for $\vec{l}$. The fanout constraint $f_{\max} = 2$ is preserved in the children.

5.4 Using Graph Isomorphism

If we consider systems in which all the processors are identical (homogeneous processor set), then we can pare the design space significantly if we only consider isomorphically unique topology graphs. Two graphs $T = (P, L)$ and $T'(P', L')$ are isomorphic if we can relabel the vertices of T to be vertices of T', maintaining the corresponding edges in T and T'.

Consider a topology graph T with $|L|$ edges and $|P|$ nodes where each node corresponds to a processor and each edge corresponds to a link between two processors. The maximum number of edges in T is $L_{\max} = |P|(|P| - 1)$ corresponding to a fully connected graph (full crossbar interconnect). We can represent the graphs with either an adjacency list or adjacency matrix and label each different representation. Then, for a graph with $|L|$ edges the number of different labellings is given by

$$n_g = \binom{L_{\max}}{|L|} = \frac{L_{\max}!}{|L|!(L_{\max} - |L|)!} = \frac{|P|(|P|-1)!}{|L|!(|P|(|P|-1) - |L|)!}, \tag{9}$$

which increases exponentially with $|P|$. The maximum value of n_g occurs at $|L| = L_{\max}/2$. However, the number of isomorphically unique graphs n_{unique} is much less than n_g. For very small $|P|$, we can enumerate the different possibilities to show this. Fig. 10 depicts the different isomorphic graphs for $|P| = 4$ processors and different numbers of links. For $|L| = 3$ there are 20 different graph labellings, but we observe that most are isomorphic—only 3 are isomorphically unique.

For larger $|P|$, n_g increases rapidly according to (9). We enumerated the possibilities and tested for isomorphism for $|P| = 5$ and $|P| = 6$ using Brendan McKay's *nauty* program [28], which is currently the fastest published graph isomorphism testing program. The results are shown in Fig. 11. For $|P| = 6$ and $|L| = 12$, we observe that there is a 3 order magnitude difference between the number of graph labellings n_g and those that are isomorphically unique (n_{unique}). Also, this ratio increases with n_g.

We exploit this property to improve our genetic algorithm. As mentioned earlier, each generation in the GA consists of a predetermined number of individuals derived from the previous generation by crossover and mutation operators. The initial population is generated randomly. However, the results from Fig. 11 imply that a large fraction of the individuals in any randomly generated population will be isomorphically equivalent, and that we would be wasting computation time by operating on equivalent interconnection topologies. Instead, we employ an efficient online graph isomorphism test when generating the population, and only accept new individuals that are isomorphically unique. By reducing the solution space by orders of magnitude, the GA can search it more thoroughly in a given amount of computation time, and produce better results.

The graph isomorphism test is advantageous for the link synthesis algorithm only if the isomorphism testing is efficient. The complexity of the graph isomorphism problem is still an open question—there exists no known P algorithm for graph isomorphism testing, although the problem has also not been shown to be NP-complete. It is thought that the problem falls in the area between P and NP-complete, if such an area exists [29]. However, McKay's nauty [28] program has been proven to be very efficient in practice. Although its worst-case runtime is exponential [30], an empirical test of a large number of randomly generated graphs produced run times of $1.2|P|^2$ microseconds on a 1 GHz Pentium III machine [28]. By comparison, the DLS scheduling algorithm has complexity $O(|V|^3|P|)$, where $|V|$ is the number of nodes in the *task graph* [21]. We modify the DLS scheduling algorithm by adding a flexibility calculation at each scheduling step. The complexity of the flexibility algorithm is $O(|V|(|V| + |E|)\,|P| \log |P|)$, so the overall complexity scheduling an arbitrary graph using the modified DLS scheduling algorithm is

$$O(|V|^4(|V| + |E|)|P|^2 \log |P|). \tag{10}$$

The number of tasks in the application will be much greater than the number of processors in practice, so $|V| >> |P|$ and $|E| >> |P|$. For randomly generated graphs, the nauty

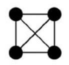
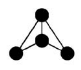
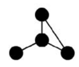
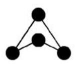
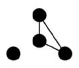
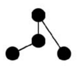
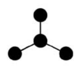
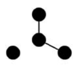
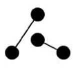
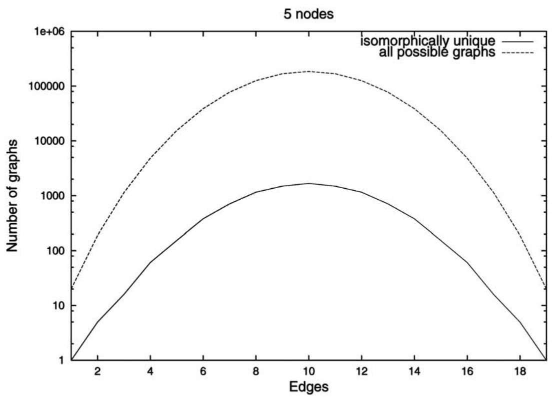
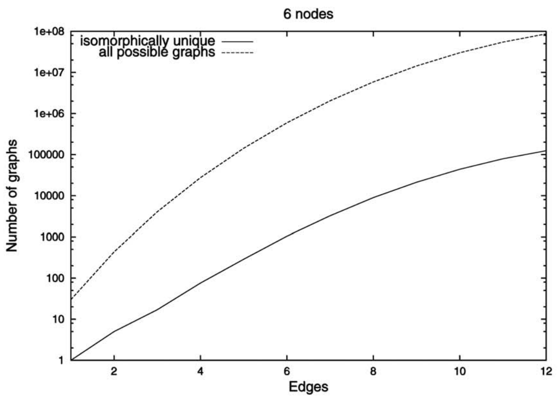
E	n_g	n_{unique}	
6	1	1	
5	6	1	
4	15	2	 
3	20	3	  
2	15	2	 

Fig. 10. Isomorphically unique graphs containing E edges for $N = 4$ processors. Here, depict undirected graphs in order to make the figure clearer.



(a)



(b)

Fig. 11. A comparison of the number of possible graph labellings n_g given by (9) with the number of these graphs that are isomorphically unique. (a) $|P| = 5$ and (b) $|P| = 6$.

program is therefore much faster than the modified DLS scheduling algorithm and we achieve significant speedup by detecting and exploiting graph isomorphism.

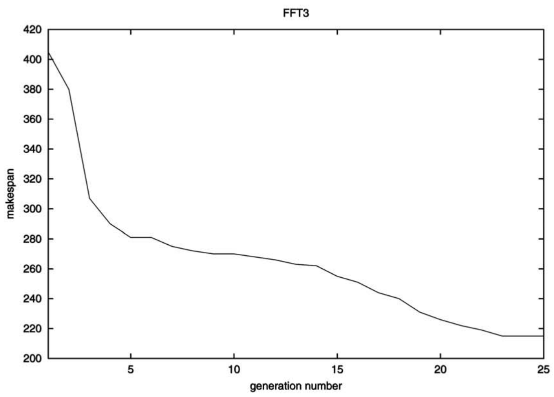
5.5 Interconnect Synthesis Experiments

We evaluated our interconnect synthesis algorithm on several DSP benchmark task graphs. Fig. 12a shows the convergence of the GA versus generation number for an FFT3 application, with population size $N = 100$ and a maximum allowed fanout of 4. In this plot, the y-axis refers to the schedule makespan of the best interconnection topology found for a given generation. We see that the GA was able to reduce the makespan by almost a factor of two over the best topology in the initial population. Fig. 12b shows how the makespan improves as the maximum fanout constraint is increased. As explained in [10], there is an increasing area overhead associated with a greater number of global interconnects. In our model, the number of global interconnects is equal to the number of processors times the average fanout per processor. Therefore, increasing the maximum fanout constraint amounts to an area/performance trade off.

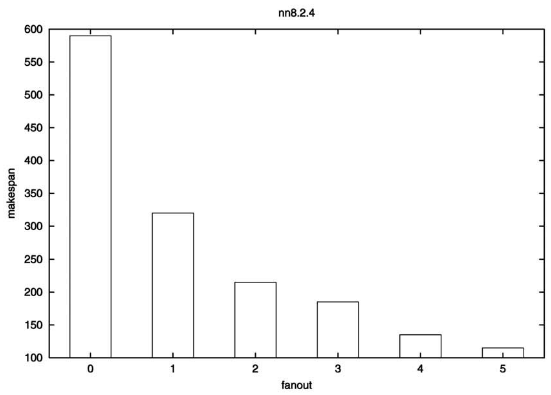
To illustrate the benefits of the feasibility/flexibility framework in the context of interconnect synthesis, we ran the GA for a fixed optimization time both with and without employing the feasibility algorithm. When the feasibility algorithm is not used, the scheduler can deadlock, resulting in an invalid solution. Valuable optimization time is wasted on these individuals. In addition, without the flexibility metric to help guide the scheduler, the valid (nondeadlocked) solutions are of lower quality. As a result, we observe a significant improvement for the GA utilizing feasibility/flexibility. Table 2 summarizes results for eight benchmark applications. The improvement values quoted in Table 2 correspond to the ratio of the lowest makespan of all topologies in the initial population to the makespan of the best topology (lowest makespan) found by the GA.

6 Conclusion

New interconnect technologies are promising for global communication in embedded multiprocessors, since the interconnection patterns can flexibly be streamlined and reconfigured to match the target applications. We have shown that new algorithms are needed to handle the resulting irregular interconnect topologies. Existing multiprocessor list scheduling algorithms do not take the effects of these irregular topologies into account. We have demonstrated an efficient algorithm for determining the set of feasible processors that will avoid schedule deadlock, and a useful metric, called communication flexibility, for the degree to



(a)



(b)

Fig. 12. (a) GA output versus generation and (b) GA output versus processor fanout constraints.

TABLE 2
Link Synthesis GA Employing Scheduler Both with and without Feasibility/Flexibility Metric

Application	Processors	Links	Improvement utilizing feas./flex.	Improvement without feas./flex.
FFT1	5	10	3.79	1.70
Karp10	8	28	4.43	1.16
Irr	8	35	2.20	1.83
Qmf4	4	6	1.73	1.45
NN16-3-4	8	24	5.56	2.74
Sum1	6	12	2.81	1.44
Laplace	7	15	3.65	2.18
FFT3	6	15	3.20	2.09

which a given scheduling decision constrains future decisions (in the context of the given communication topology). We used this algorithm and the flexibility metric in conjunction with the DLS algorithm to map several DSP applications across a wide range of interconnect topologies. The results demonstrate both the soundness of our feasibility computation techniques, and the utility of our flexibility metric in guiding the scheduling process.

We have also shown the utility of producing multiprocessor schedules in which all data transfers occur in a limited number of hops. Such single-hop or low-hop schedules can result in lower latency and lower power. With single-hop schedules the overhead associated with routing data through intermediate processors is eliminated. Due to the power consumption characteristics of optical links, it is useful to restrict communication across them to low-hop transfers. Our flexibility algorithm is able to produce these schedules.

Interconnect synthesis is becoming an increasingly important problem for designers of systems-on-chip as the designs become larger. We presented a genetic algorithm for synthesizing efficient interconnection networks for SoC. The algorithm works in conjunction with a list scheduling algorithm to jointly optimize both the schedule and the interconnect topology. The algorithm is able to account for different distributions of local versus global (long) interconnect routing tracks via a processor fanout constraint. It uses graph isomorphism to significantly pare the search space in order to search more efficiently.

ACKNOWLEDGMENTS

This research was supported in part by the US Defense Advanced Research Projects Agency (DARPA) under Contract number MDA972-00-1-0023, through Brown University.

REFERENCES

[1] S. Murali and G.D. Micheli, "Bandwidth-Constrained Mapping of Cores onto NoC Architectures," *Proc. Conf. Design Automation and Test in Europe,* pp. 896-901, Feb. 2004.

[2] S. Kumar, A. Jantsch, J.P. Soinen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A Network On Chip Architecture and Design Methodology," *Proc. IEEE Symp. Very Large Scale Integration,* pp. 105-112, Apr. 2002.

[3] T. Blickle, J. Teich, and L. Thiele, "System-Level Synthesis Using Evolutionary Algorithms," *Kluwer J. Design Automation for Embedded Systems,* vol. 3, pp. 23-62, 1998.

[4] R.P. Dick and N.K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 17, no. 10, pp. 920-935, Oct. 1998.

[5] S. Sriram and S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker Inc., 2000.

[6] J. Hu and R. Marculescu, "Exploiting the Routing Flexibility for Energy/Performance Aware Mapping of Regular NoC Architectures," *Proc. Conf. Design, Automation, and Test in Europe,* Mar. 2003.

[7] W.H. Ho and T.M. Pinkston, "A Methodology for Designing Efficient On-Chip Interconnects on Well-Behaved Communication Patterns," *Proc. Ninth Int'l Symp. High-Performance Computer Architecture,* pp. 377-388, Feb. 2003.

[8] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD: Reconfigurable Pipelined Datapath," *Proc. Sixth Int'l Workshop Field-Programmable Logic and Applications,* pp. 126-135, 1996.

[9] A. Singh, A. Mukherjee, and M. Marek-Sadowska, "Interconnect Pipelining in a Throughput-Intensive FPGA Architecture," *Proc. ACM/SIGDA Ninth Int'l Symp. Field-Programmable Gate Arrays,* pp. 153-160, 2001.

[10] A. Sharma, C. Ebeling, and S. Hauck, "Piperoute: A Pipelining-Aware Router for FPGAs," *Proc. 11th ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays,* pp. 68-77, 2003.

[11] S. Hauck, G. Borriello, and C. Ebeling, "Mesh Routing Topologies for Multi-FPGA Systems," *IEEE Trans. Very Large Scale Integration Systems,* vol. 6, no. 3, pp. 400-408, Sept. 1998.

[12] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1991.

[13] J. Hwang, Y. Chow, F. Anger, and C. Lee, "Scheduling Precedence Graphs in Systems with Inter-Processor Communication Times," *SIAM J. Computing,* vol. 18, no. 2, pp. 244-257, Apr. 1989.

[14] H. El-Rewini, T. Lewis, and H. Ali, *Task Scheduling in Parallel and Distributed Systems.* Englewood Cliffs, N.J.: Prentice Hall, 1994.

[15] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, May 1996.

[16] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *J. Parallel and Distributed Computing,* vol. 16, no. 4, pp. 276-291, Dec. 1992.

[17] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke, "Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 9, pp. 967-985, Sept. 2001.

[18] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 9, pp. 872-892, Sept. 1998.

[19] J. Colin and P. Chretienne, "C.P.M. Scheduling with Small Computation Delays and Task Duplication," *Operations Research,* pp. 680-684, 1991.

[20] S. Darbha and D. Agrawal, "Optimal Scheduling Algorithm for Distributed Memory Machines," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 1, pp. 87-95, Jan. 1998.

[21] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 75-87, Feb. 1993.

[22] Y.-K. Kwok and I. Ahmad, "Link Contention-Constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors," *Cluster Computing,* vol. 3, no. 2, pp. 113-124, Sept. 2000.

[23] D.A. Miller, "Rationale and Challenges for Optical Interconnects to Electronic Chips," *Proc. IEEE,* vol. 88, no. 6, pp. 728-749, June 2000.

[24] K. Day and A. Al-Ayyoub, "Topological Properties of OTIS-Networks," *IEEE Trans. Parallel and Distributed Systems,* vol. 13, no. 4, Apr. 2002.

[25] A. Kahn, C. McCreary, J. Thompson, and M. McArdle, "A Comparison of Multiprocessor Scheduling Heuristics," *Proc. 1994 Int'l Conf. Parallel Processing,* vol. 2, pp. 243-250, 1994.

[26] G.C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," PhD dissertation, Dept. of Electrical Eng. and Computer Science, Univ. of California at Berkeley, Apr. 1991.

[27] P. Marwedel and G. Goosens, *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[28] B. McKay, "Nauty User's Guide," Technical Report TR-CS-90-02, Australian Nat'l Univ., 1990.

[29] S. Skiena, *Graph Isomorphism, Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Reading, Mass.: Addison-Wesley, 1990.

[30] T. Miyazaki, "The Complexity of McKay's Canonical Labeling Algorithm," *Groups and Computation II,* vol. 28, pp. 239-256, 1997.

Neal K. Bambha received the BS degrees in physics and electrical engineering (with honors) from Iowa State University and the MS degree in electrical engineering from Princeton University. He received the PhD degree in electrical engineering from the University of Maryland, College Park, in 2004. He is a member of the technical staff at the US Army Research Laboratory. His research interests include hardware/software codesign, signal processing, optical interconnects within digital systems, and evolutionary algorithms. He is a member of the IEEE and the IEEE Computer Society.

Shuvra S. Bhattacharyya received the BS degree from the University of Wisconsin at Madison, and the PhD degree from the University of California at Berkeley. He is an associate professor in the Department of Electrical and Computer Engineering, and the Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. He is also an affiliate associate professor in the Department of Computer Science. Dr. Bhattacharyya is coauthor of two books and the author or coauthor of more than 60 refereed technical articles. His research interests include signal processing, embedded software, and hardware/software codesign. Dr. Bhattacharyya has held industrial positions as a researcher at the Hitachi America Semiconductor Research Laboratory, and as a compiler developer at Kuck & Associates. He is a senior member of the IEEE and the IEEE Computer Society.