Hybrid Scheduling of Dynamic Task Graphs with Selective Duplication for Multiprocessors under Memory and Time Constraints

Pravanjan Choudhury, Rajeev Kumar, *Senior Member*, *IEEE*, and
P.P. Chakrabarti, *Senior Member*, *IEEE*

Abstract—This paper presents a hybrid scheduling methodology for task graphs to multiprocessor embedded systems. The proposed methodology is designed for task graphs that are dynamic in nature due to the presence of conditional tasks and tasks whose execution times are unpredictable but bounded. We have presented the methodology as a three-phase strategy, in which task nodes are mapped to the processors in the first (static mapping) phase. In the second (selective duplication) phase, some critical nodes are identified and duplicated for possible rescheduling at runtime, depending on the code memory constraints of the processors. The third (online) phase is a runtime scheduling algorithm that performs list scheduling based on the actual dynamics of the schedule up to the current time. We show that this technique provides better schedule length (up to 20 percent) compared to previous techniques, which are predominantly static in nature, with low overhead and a complexity comparable with existing online techniques. The effects of model parameters like the number of processors, memory, and various task graph parameters on performance are investigated in this paper.

Index Terms—Multiprocessor scheduling, conditional task graphs, unpredictable tasks, static and online scheduling, hybrid scheduling, node duplication.

1 Introduction and Related Work

Multiprocessor-based embedded systems have become quite widespread in recent times. Embedded systems such as personal handheld devices, set-top boxes, and miniprinters consist of complex applications with real-time performance requirements. For such applications, a multiprocessor architecture is getting increasingly preferred over a single processor solution to achieve the required performance. Each processor in a multiprocessor system usually has its local memory for code storage and execution. A *macro data-flow graph* (usually represented as a *directed acyclic graph* (DAG)) is used to describe an application at a high level. Mapping and scheduling of macro data-flow graphs to such multiprocessor architectures is a fundamental and challenging problem that is being addressed by many researchers. Several methods have been proposed for specific applications and architecture models to minimize the application execution time.

The system model for the synthesis of a macro data-flow graph uses a high-level *task graph* and a *processor* model. The general problem of the mapping and scheduling of high-level task graphs to a given multiprocessor system to minimize schedule length is known to be NP-complete. Since optimal polynomial time solutions are known only for highly constrained and restricted task graphs and processor models, heuristic-based algorithms have appeared in the literature for various kinds of task graphs and processor models [1]. A variety of task graph models has been proposed, for example, arbitrary communication/computation execution cost [2], [3], conditional nodes [4], [5], [6], and probabilistic execution cost [7]. Similarly, some scheduling algorithms consider heterogeneous [8] and arbitrary connected processor models [9]. Embedded multiprocessor systems are usually limited by the number of processors, local storage capacity, and communication. Scheduling algorithms for embedded processors usually avoid code migration and allow limited code duplication. Among the above models, this work focuses on the scheduling problem of task graphs having conditional nodes and tasks with unpredictable execution behavior on a set of homogeneous processors.

A conditional task graph (CTG) model captures the control flow of an application, in addition to the data flow (captured in unconditional graph models). The outgoing edges associated with a conditional node depict the control behavior of task graph at that node. An example CTG is shown in Fig. 1, which contains one conditional node v_3. During the execution of the task graph, one of the conditional branches is selected for further execution, depending on the condition evaluated at that node. The majority of the proposed algorithms for CTGs belong to the class of static list scheduling algorithms. *Static* (compile-time) scheduling methods map and schedule the tasks *offline* and minimize worst-case execution time (WCET) and processor resources. For example, Eles et al.[4] extracted all possible conditional (N alternative) schedules and then used a table merging technique to determine the final schedule. Since this technique extracts all possible conditional execution paths and then merges them, the processing overhead is expected tobe large if the number of such conditional paths issignificant. Some researchers proposed stochastic heuristic-based schedulers. For example,

• *The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India—721302.*
E-mail: {pravanjan, rkumar, ppchak}@cse.iitkgp.ernet.in.

Manuscript received 10 June 2007; revised 24 Aug. 2007; accepted 29 Aug. 2007; published online 26 Sept. 2007.
Recommended for acceptance by D. Trystram.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-06-0187.
Digital Object Identifier no. 10.1109/TPDS.2007.70784.

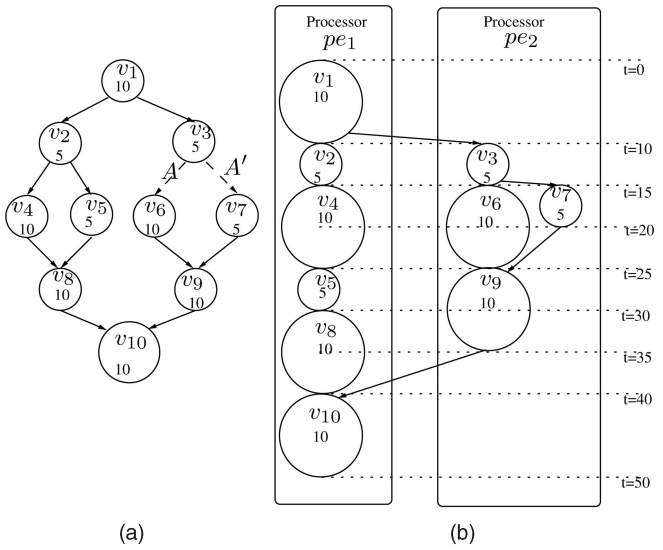

Fig. 1. An example. (a) An example task graph. (b) Task graph scheduled on two processors.

Grajcar [5] used a genetic algorithm combined with list scheduling. The static scheduling method proposed by Xie and Wolf [6] uses static mutual exclusion detection among nodes to effectively utilize the processor resources. This technique minimizes the worst-case schedule length. Resource reclaim during execution may be performed on individual processors for the improvement of schedule length. Resource reclaim here means that if a task is ready for execution in the scheduled processor before the start time assigned by the static scheduler, then it can start execution early. However, such an improvement is usually not significant, since static scheduling precommits the schedule based on a worst-case scenario.

Furthermore, it is not possible to predict the exact execution time of a macrotask within a task graph at runtime. Usually, static schedulers use the WCET of every task, which is known a priori. A WCET greater than multiple times that of the actual execution time has been reported in [10]. Since the exact execution time of any task can only be known at runtime, static scheduling again relies on resource reclaim within each processor to improve the schedule length.

The presence of conditional tasks and tasks with unpredictable execution behavior changes the dynamics of the task graph at runtime. Thus, the initial assumptions by static schedulers about the behavior of the tasks change during execution. Hence, few processors may get relatively more free time than others and have to wait for tasks to get finished on other processors to start execution again due to the precedence constraints. This leads to load imbalances in task distribution, which may not be handled by static schedulers.

Alternatively, *online* (dynamic) schedulers schedule tasks to processors using the information available at runtime. Feldmann et al. presented an efficient technique for the online scheduling of parallel jobs, assuming that the task execution time is not known a priori [11]. However, scheduling of parallel tasks often results in considerable overheads at runtime. To reduce the scheduling overhead, many low overhead techniques have been proposed for dependent tasks, for example, the scheduling algorithms proposed by Anderson et al. [12] and Gupta et al. [13]. Online schedulers can also utilize static information about the nature of the applications to effectively estimate future resource requirements. Such schedulers require that the collective code of all task nodes of the CTG resides on every processor so that they are schedulable on any processor at runtime. However, complete code mirroring on all processors may result in the capacity violation of local memory in embedded systems. The task partitioning problem with processor local memory constraint for independent tasks on multiprocessors was addressed by Fisher et al. [14].

In the setting of communication-intensive task graphs, the duplication of nodes on more than one processor (for a simultaneous execution of the same task) has been proved to be useful in reducing the wait time of ready tasks, for example, in the strategies proposed by Ahmad and Kwok [15] and Bansal et al. [16], [17]. Similarly, an optimal scheduling algorithm on an arbitrary number of processors for low-communication task graphs has been proposed by Park and Choe [18]. In this work, we have shown that even without communication, limited duplication can be useful for task graphs consisting of 1) conditional nodes and 2) tasks with unpredictable but bounded execution behavior. An online scheduler at runtime may then reschedule some of the nodes on different processors, mainly based on two types of dynamics: 1) the conditions evaluated and 2) the actual execution time of the finished nodes. Using limited duplication, task migration can be avoided completely. However, it leads to an important problem of selecting the critical tasks for duplication while meeting the code memory constraints, which is presented in this paper. The problem of limited duplication for minimizing communication, however, is a separate problem, which is not addressed in this work. Here, we assume the communication cost between tasks to be negligible.

We propose a hybrid scheduling methodology for dynamic task graphs (which are essentially CTGs with tasks of unpredictable execution behavior), which maps task nodes to processors in the static mapping stage and assigns start times to the nodes in the online scheduling stage. The strategy has three phases comprising task mapping, selective duplication, and an online scheduling algorithm. We compare the proposed strategy with two well-known static schedulers [3], [6] for multiprocessors under memory constraints. Experimental results indicate that the proposed methodology shows a significant average schedule length improvement over the static schedulers. The results also establish a relationship between the schedule length improvement and processor local memory constraints.

The rest of this paper is organized as follows: We explain the motivation of the work with an example in Section 2. Task and processor models, notations, and definitions are included in Section 3. Section 4 presents the algorithms for the three-phase strategy. Experimental results and comparison with static schedulers for a wide range of random task graphs are presented in Section 5. Section 6 concludes this work.

2 Motivational Example

A CTG represents the control flow and data flow of an application. For example, the FFT implementation in the Michigan (MI) benchmark [19] typically consists of two branch and join statements (such as an if/then/else case statement). Similarly, the Secure Hash Algorithm (SHA) contains 11 such statements. A decision or branch condition

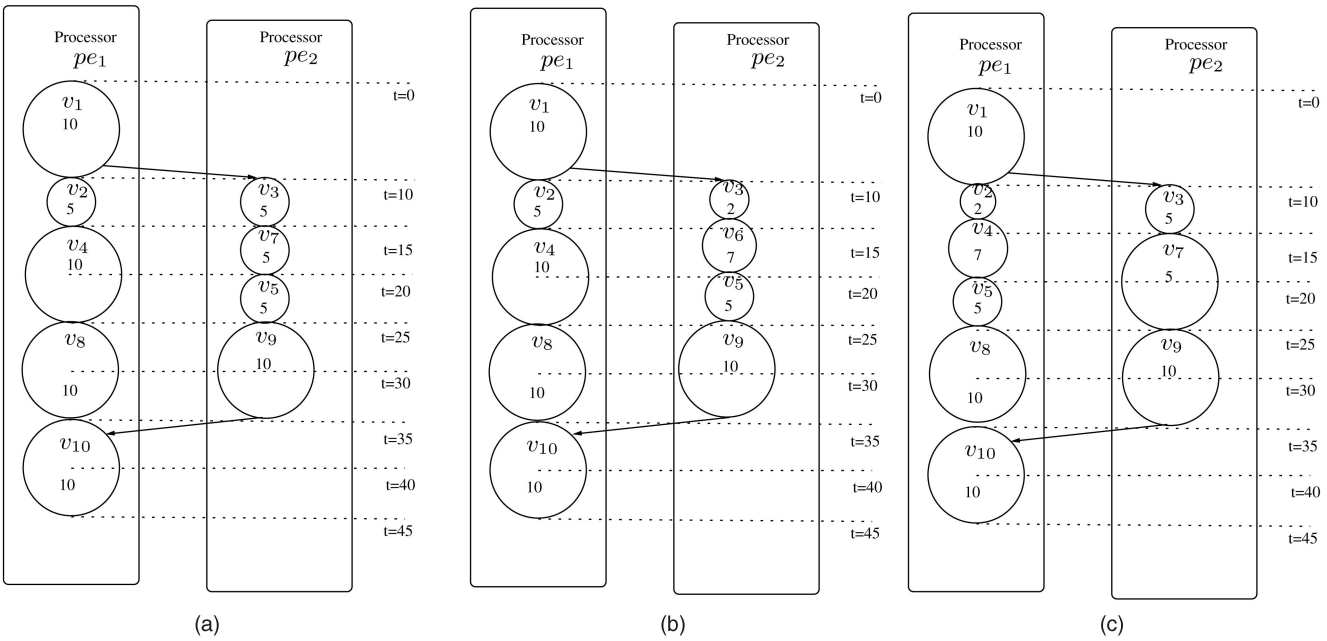

Fig. 2. Example of online schedules for Fig. 1. (a) Online schedule with condition A'. (b) Schedule with v_3 and v_6 finishing early. (c) Schedule with v_2 and v_4 finishing early.

inside an application forms a branch node in the corresponding task graph. The branch condition evaluation depends usually (not essentially) on the input data of the execution instance. Each task node inside a CTG depicts a certain functionality, and it may again contain inherent conditional paths, loops that make its execution nondeterministic. Parameters like cache hit and interrupts also contribute to the unpredictable execution behavior of the tasks at runtime. For example, the execution time of a task inside a SHA implementation may vary by several thousand cycles in multiple runs. Applications in current embedded systems are even more complex, and they are composed of several such blocks, which makes their execution even more control oriented and unpredictable. Hence, statically scheduling such task graphs to a multiprocessor results in an inefficient schedule at runtime, and fully dynamic scheduling requires complete code mirroring. Even with complete code mirroring without any static analysis, the runtime schedule may become inefficient. Traditionally, homogeneous processor models are more popular than heterogeneous models, because they are relatively easier to design and have shorter turnaround time.

The following example illustrates the improvement that can be achieved by a simple online remapping of nodes over a completely static schedule. We also discuss how a complete duplication of nodes may be avoided to achieve the above improvement.

Fig. 1a shows a 10-node task graph with one conditional node v_3. Let us initially assume that all tasks have a fixed execution time, which is shown within the respective nodes in Fig. 1a. The task graph is scheduled to a homogeneous two-processor system (pe_1 and pe_2) using static scheduling, as described in [6] (see Fig. 1b). It can be seen that for conditions A or A', the schedule length is 50.

Now, if we apply an Earliest Task First (ETF)-based online scheduling to the same task graph, we observe that the worst-case schedule length remains the same (for condition A), but the schedule length is improved by five units in the case of A' (Fig. 2a). It can also be seen that even with condition A, there is a scope of schedule length improvement if the nodes in processor (pe_1) finish their execution early. For example, Fig. 2b shows that if nodes v_3 and v_6 both execute at three units less than their WCET, the schedule length is decreased by five units. Therefore, a pure resource reclaim does not suffice to adjust to the needs of schedule correction for varying dynamics of the execution, because it does not remap or reschedule any task across the processors. In the online scheduler, initially, we assumed that any arbitrary node can be started in any processor. This requires the code of each task to reside on every processor. However, it can be seen that during an online scheduling, though a node can be started in any processor, it may not necessarily improve its start time or overall schedule length due to precedence constraints and graph structure. For example, tasks v_2 and v_3 can be started in either processor without any schedule length benefit. However, note that in the cases presented above (Figs. 2a and 2b), only the duplication of task v_5 in pe_2 is crucial in determining the schedule length. One may argue that mapping v_5 permanently to pe_2 will take care of the above cases, without the need of mapping into both. However, Fig. 2c shows that if tasks v_2 and v_4 finish their execution early instead of tasks v_3 and v_6, it is more judicious to start task v_5 on processor pe_1 instead of processor pe_2. The motivation of this work is to design a suitable initial mapping and to identify nodes such as v_5 to duplicate within the memory constraint and then apply an online scheduler to improve the average-case schedule length.

3 Models, Notations, and Definitions

The first two sections describe the task graph and processor models, which have been found offline and input to the proposed hybrid strategy. Section 3.3 describes the derived notations, which will be used in the subsequent sections. The following sections present the notations specific to the static mapping and online scheduling separately.

3.1 Task Graph Model

An application macro data-flow graph is represented as a special DAG $G(V_s, V_c, E_s, E_c)$, called a CTG. Each node in the DAG represents a computational block, which has to be executed without preemption. Nodes $V_s(G)$ and $V_c(G)$ represent the simple and conditional nodes of graph G. Each node v_i ($v_i \in V(G)$) can execute between its worst-case and best-case execution values, which are known a priori. The precedence relation between two nodes is expressed as a directed edge. For a simple node ($v_i \in V_s(G)$), all its child nodes can start their execution only after v_i completes its execution. Each conditional node ($v_i \in V_c(G)$) is associated with a condition, in which, when evaluated, one among the associated conditional edges is selected for further execution. $E_s(G)$ and $E_c(G)$ are the sets of simple and conditional edges, respectively, describing data-flow and control-flow relations among the tasks. The communication cost along these edges is assumed to be zero in this work:

1. $\mathbf{V(G)}$, $\mathbf{E(G)}$: the set of all nodes and edges, respectively, of graph G.
2. $\mathbf{C(G)}$: the set of all conditions. $C = \{c_1, c_2, c_3, \ldots c_{|C(G)|}\}$, where c_i is the ith condition. Each condition is associated with a corresponding node $v_i \in V(G)$, called a conditional node. The number of conditions ($|C(G)|$) is expressed as a fraction of the number of nodes ($|V(G)|$).
3. $\mathbf{WCET_{v_i}}$: the *WCET* of v_i.
4. $\mathbf{BCET_{v_i}}$: the *Best Case Execution Time* of v_i.
5. $\mathbf{cs_{v_i}}$: the code size of v_i.
6. **source, sink**: the start and end nodes, respectively, of a CTG. For example, in Fig. 1, v_1 and v_{10} are respectively the source and sink nodes.

3.2 Processor Model

The architectural model is represented by a set of processing elements. Symbolically, $PE = \{pe_1, pe_2, pe_3, \ldots pe_{|PE|}\}$, where pe_i is the ith processing element. Each processor has its own local code memory. A processor pe_i can execute a task v_j if its code has been mapped on it. All processors are assumed to be homogeneous:

1. $\mathbf{pe_i(v_j)}$: a Boolean expression, which is true when the code of v_j resides on pe_i, or else, this is false. This indicates whether the code of v_j is present in pe_i or not. Initially, $\forall pe_i \in PE$, $v_j \in V(G) pe_i(v_j)$ is false.
2. $\mathbf{M_{pe_i}}$: the available local code memory in pe_i. It is expressed as a fraction of the cumulative sum of the code size of nodes $\sum cs_{v_i}$.

3.3 Derived Task Graph Notations

1. $\mathbf{parent_{v_i}}$, $\mathbf{child_{v_i}}$: the set of immediate parent and children nodes of v_i, respectively. For example, in Fig. 1, $child_{v_3} = \{v_6, v_7\}$.
2. $\mathbf{ancestors_{v_i}}$: the set of all nodes on which v_i is dependent in the precedence relation of the CTG. For example, in Fig. 1, $ancestors_{v_6} = \{v_3, v_1\}$.
3. $\mathbf{descendants_{v_i}}$: the set of all nodes dependent on v_i in the precedence relation of the CTG. For example, in Fig. 1, $descendants_{v_6} = \{v_9, v_{10}\}$.
4. $\mathbf{Branch\_Info(v_i)}$: the branch information structure, which represents the branch condition on which v_i is executed. The detailed process of its creation is detailed in [6] and is given as a procedure in Algorithm 2 in this paper for completeness. The branch_info structure for a task v_i is given by the following values:
   - **level**: the number of branch tasks that have to be executed before reaching v_i.
   - **branch_label**: the ordered set of condition names that must have been executed if v_i is executed. $branch\_label[j]$ is the name of the jth-level branch for v_i.
   - **branch_condition**: the ordered set of condition values that must be true if v_i is executed. Note that for each $branch\_label[j]$, there will be a $branch\_condition[j]$, which denotes the branch taken from $branch\_label[j]$. For example, in Fig. 1, $branch\_label$ for both v_6 and v_7 is A, where as $branch\_conditions$ are A and A', respectively.
5. $\mathbf{mutex_{v_i}}$: the set of all nodes that are mutually exclusive with v_i in the CTG. For example, in Fig. 1, $mutex_{v_6} = \{v_7\}$. The mutual exclusion detection procedure is taken from [6] and has been added as a procedure in Algorithm 2. Given the branch information structure of two tasks, it defines three rules to determine if the two tasks are mutually exclusive.
6. **maxedge**: the maximum number of possible edges in an unconditional graph with $|V(G)|$ vertices, that is, ${}^{|V(G)|}C_2$. In a CTG, due to mutual exclusion among the nodes, the maximum possible edges will be less than ${}^{|V(G)|}C_2$. Thus, **maxedge** can be defined as the maximum number of edges possible in a CTG by adding edges without violating mutual exclusion.
7. **edgedensity**: $|E(G)|$, expressed as a fraction of **maxedge**.

3.4 Static Scheduling Notations

A node is said to be *scheduled* when its start time and processor mapping are allocated. In list-scheduling-based methods, task graphs are scheduled incrementally. When a subgraph of the task graph is scheduled, it is called a *partial schedule*:

1. $\mathbf{pe(v_i)}$: the processor on which v_i is scheduled.
2. $\mathbf{SU_{v_i}}$: the static urgency (SU) of a node v_i. It is the maximum length from the start of v_i to the sink. For example, in Fig. 1, two paths from v_3 to v_{10} are $v_3 \rightarrow v_6 \rightarrow v_9 \rightarrow v_{10}$ and $v_3 \rightarrow v_7 \rightarrow v_9 \rightarrow v_{10}$, and the maximum length is 35 due to the first path. Hence, $SU_{v_3} = 35$.
3. $\mathbf{ready\_time(v_i)}$: the time t at which v_i can start execution in a partial schedule, that is, t, at which $parent_{v_i}$ finishes execution.
4. $\mathbf{PE\_Available\_time(v_i, pe_j)}$: the time unit t at which v_i can start its execution on pe_j in a partial schedule.
5. $\mathbf{DU_{v_i,pe_j}}$: the dynamic urgency of a node v_i on processor pe_j in a partial schedule, which is computed by the expression $SU_{v_i} - Max(ready\_time(v_i), PE\_Available\_time(v_i, pe_j)) - WCET_{v_i}$. For details, see [6].
6. **EndSchedule**: a Boolean variable, for which a true value indicates the end of the schedule for a CTG. *EndSchedule* is true for $t > ST_{sink} + ex_{sink}$; otherwise,

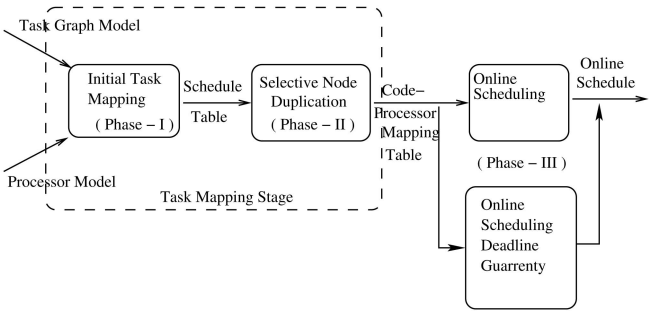

Fig. 3. Flow diagram of the proposed three-phase strategy.

this is false.

3.5 Online Scheduling Notations

At runtime, due to conditions evaluated at the branch nodes, a subset of the task graph is executed. Moreover, the actual execution time of a node can be found out only at runtime:

1. $\mathbf{t}$: represents the tth time unit in an online scheduling.
2. $\mathbf{ex_{v_i}}$: the actual execution time of the ith node in $V(G)$. ex_{v_i} takes a value between $BCET_{v_i}$ and $WCET_{v_i}$ at runtime. For example, in Fig. 1, $ex_{v_0} = 10$.
3. $\mathbf{parent^o_{v_i}}$, $\mathbf{child^o_{v_i}}$: the set of the immediate parent and children nodes of v_i at runtime, respectively. For example, in Fig. 1, $child_{v_3} = \{v_6, v_7\}$, whereas if v_3 evaluates to A during runtime, then $child^o_{v_3} = \{v_6\}$.
4. $\mathbf{ST^o_{v_i}}$: the execution start time of v_i at runtime.

4 Code Mapping and Online Scheduling

Fig. 3 describes the proposed three-phase strategy that consists of a task mapping and an online scheduling stage. The first two phases comprise the task mapping stage. The code segments of the tasks are allocated to the processors according to the schedule table generated after the second phase. The third, that is, the last, phase is the online scheduling stage, in which a start time is assigned to each mapped node at runtime. The static scheduling technique proposed by Xie and Wolf [6] is modified to incorporate code balancing for the first phase (Algorithm 1). The static scheduling assumes the WCET for each node while scheduling.

In the second phase (Algorithm 4), the static schedule produced in the first phase is analyzed, and a set of node processor mapping vectors are identified. A weight of each such vector is defined toward task mirroring on that processor. We then choose a subset of the vectors so as to maximize the total weight within each processor code memory constraint. The new node-processor mapping table generated after *selective node duplication* is used to map the code segments to the processors. The SU of the nodes and the mapping table are provided to the online scheduler. Based on the execution state (created by the conditional path taken and the actual execution time of finished nodes), node urgency, and mapping table considered, the online scheduler (Algorithm 5) activates each node in one of the processors on which the corresponding code segment is mapped. We also present a variation of the online scheduler, which guarantees a schedule length less than or equal to that obtained from the static scheduling in the first phase. The details of the algorithms are discussed next.

4.1 Task Mapping

4.1.1 Phase 1: Initial Task Mapping (Algorithm 1)

The objective of the Phase-1 static scheduling is to get a static schedule that minimizes the worst-case makespan and is compliant with the code memory constraint of each processor. This problem is NP-complete. We show this by citing two restricted (easier) problems that are NP-complete. It can be observed that by allowing sufficient memory on each processor (so that the memory constraints are nullified), finding out a schedule that minimizes the makespan is NP-complete [1], which is a more restricted problem. It can be seen that a variant of the above problem, that of partitioning the codes of a set of N tasks to a set of M memory-constrained processors (without considering the minimization of makespan) is a direct transformation of the bin packing problem and is hence NP-complete [14]. Due to the intractable nature of the problem, we present a code-size-aware list scheduling algorithm for CTGs for this initial task mapping stage.

The algorithm used in this stage is adapted from [6], which provides efficient resource sharing by detecting *static mutual exclusion* among the nodes. Since the actual execution time of the tasks are unknown, it assumes the WCET for each task. The scheduler performs a list scheduling on the task nodes based on their dynamic urgency DU_{v_i,pe_j}. The node selected is then scheduled to the earliest available processor, and in this step, the selected node may get scheduled on the same schedule slot as any scheduled node that is mutually exclusive with it. The branch labeling and mutual exclusion detection algorithm presented in [6] is given in Algorithm 2 for completeness. For the rest of the details of the algorithm, see [6]. The modification to the algorithm is the addition of a tie-breaking policy in the processor selection phase when multiple processors can execute the task at the same start time. This step helps distribute nodes evenly among the processors in a memory-constrained system, without affecting the static schedule performance. Tie breaking between processors is done by selecting the processor that has the maximum local memory available in the partial mapping. The steps of the algorithm are explained in brief as follows:

Algorithm 1: Phase 1 (Initial Task Mapping).
Procedure Initial Task Mapping
calculate SU_{v_i} $\forall v_i \in V(G)$ {Calculate the SU of all the nodes}
$AvailMem_{pe_i} \leftarrow M_{pe_i} \forall pe_i \in PE$
while *not EndSchedule* **do**
  for all $v_i \in ReadyList$ and $pe_j \in PE$ **do**
    Calculate DU_{v_i,pe_j} {Dynamic urgency as defined in Section 3.4}
  end for
  Select V_m such that $\forall v_i \in V_m \subset V$ and $pe_j \in PE$, DU_{v_i,pe_j} *is maximum* {Set of all nodes that produces the same dynamic urgency}
  Select v_m *from* V_m at random {Random node tie breaking}
  Select PE_M such that $\forall pe_m \in PE_M \subset PE$, DU_{v_m,pe_m} *is maximum* {Set of all processors that has the same dynamic urgency for the selected node pe_m}
  Select pe_m *from* PE_M such that $AvailMem_{pe_m}$

is maximum, random if not unique {Processor selection: select the processor with the highest available memory}
$pe_m \leftarrow v_m$ and update $AvailMem_{pe_m}$ {Schedule v_m on pe_m}
end while
End Procedure (Initial Task Mapping)

Procedure $PE\_Available\_time(v_i, pe_j)$ {This subroutine is used in DU_{v_i,pe_j} calculation}
if no task scheduled on pe_j Return 0
Let v_l be the latest allocated task on pe_j
while $v_l \in mutex_{v_i}$ **do**
$v_l =$ previous scheduled task on pe_j
end while
Return $ST_{v_l} + WCET_{v_l}$
End Procedure $(PE\_Available\_time(v_i, pe_j))$

Algorithm 2: Branch Labeling and Mutual Exclusion Detection.
Procedure Branch_Labeling (v_i)
if v_i is a branch join task **then**
delete $v_i.branch\_label[v_i.level]$ and $v_i.branch\_condition[v_i.level]$
$v_i.level - -$
end if
for all $v_j \in child_{v_i}$ **do**
if v_j is a branch fork task **then**
$k = v_i.level + 1$
$v_j.branch\_label = v_i.branch\_label$
$v_j.branch\_label[k] = v_i.condition\_name$
$v_j.branch\_condition[k] = branch\_condition$
else
$v_j.branch\_info = v_i.branch\_info$
end if
Branch_Labeling (v_j)
end for
End Procedure (Branch_Labeling (v_i))

Procedure (Given two tasks v_1 and v_2, the following rules are applied to detect if they are mutually exclusive)
1. If the level of one task is 0, then they are not mutually exclusive.
2. If $N = Min(v_1.level, v_2.level)$, then we compare the first N branch_label and branch_condition.
a) If $v_1.branch\_label[i] \neq v_2.branch\_label[i]$, then they are not mutually exclusive
b) If $v_1.branch\_label[i] = v_2.branch\_label[i]$ and $v_1.branch\_condition[i] \neq v_2.branch\_condition[i]$, then they are mutually exclusive
c) else compare $i + 1$ level, if $i > N$, then they are not mutually exclusive
End Procedure (Mutual Exclusion Detection)

4.1.2 Phase 2: Selective Code Duplication (Algorithm 4)

Principally, an online scheduler requires each node to be mapped to every processor so that it can start a node at any available processor at runtime. Thus, the code memory requirement on each processor is the sum of all the code sizes, that is, cs_{v_i}, which is not desirable and may violate the storage memory constraints. The selective duplication strategy tries meeting the memory constraints on each processor by suitably choosing nodes that are estimated to contribute most to the performance improvement during an online scheduling. It can be mentioned that the selective duplication algorithm presented here can be independently used with any valid schedule generated in Phase 1 by any scheduler. Therefore, it is not restricted to the particular scheduler used in Phase 1. The brief steps of the selective duplication algorithm are mentioned next and will be described later:

- Step 1. *Construction of a concurrency graph G_C and a schedule graph G_S* (Section 4.1.2). The selective duplication algorithm first constructs two types of graphs from the task graph and the schedule generated in Phase-I, namely, 1) the concurrency graph G_C and 2) the schedule graph G_S. These two graphs capture the possible concurrent execution in the CTG and dependency among the nodes in the schedule, respectively.
- Step 2. *Elimination of unimportant nodes from further consideration of duplication* (Section 4.1.2). Using the above two graphs, we identify certain nodes whose duplication in other processors are not needed. These nodes are eliminated from further consideration on their duplication on any of the other processors. We define three properties for the above elimination procedure.
- Step 3. *Weight calculation for each node processor mapping* (Section 4.1.2). Each qualified node is then considered for mapping on every processor. We estimate the maximum possible start time gain that a node v_i can gain if it is mapped on a new processor pe_j by using *the MinMax* algorithm (Section 4.1.2). A weight $W(v_i \rightarrow pe_j)$ is assigned for each new task processor mapping vector $(v_i \rightarrow pe_j)$ based on the start time gain and SU of the node.
- Step 4. *Duplication set selection* (Section 4.1.2). In this final step, a subset of the node processor mapping vectors is selected for each processor so as to maximize the sum of weights within the available memory on each processor.

Step 1: construction of a concurrency graph G_C and a schedule graph G_S. We define a concurrency graph G_C constructed from a CTG G with vertex set $V(G)$ such that for each edge $uv \in E(G_C)$, the corresponding two vertices u and v are not reachable from each other in G, and they are not statically mutually exclusive to each other. It can be seen that each edge in this concurrency graph denotes a possible concurrent execution among the corresponding nodes. For example, Fig. 4a represents the concurrency graph shown in Fig. 1a.

Next, we construct a schedule graph G_S from the static schedule generated in Phase 1. The purpose of the schedule graph is to capture the mapping information generated in Phase 1 in the form of a graph. We define a schedule graph constructed from a graph G (mapped and scheduled on P processors) as a graph G_S with vertex set $V(G)$ such that if $uv \in E(G)$ or $uv \in E(G_C)$ and $pe(u) = pe(v)$ and $ST_u < ST_v$, then $uv \in E(G_C)$.

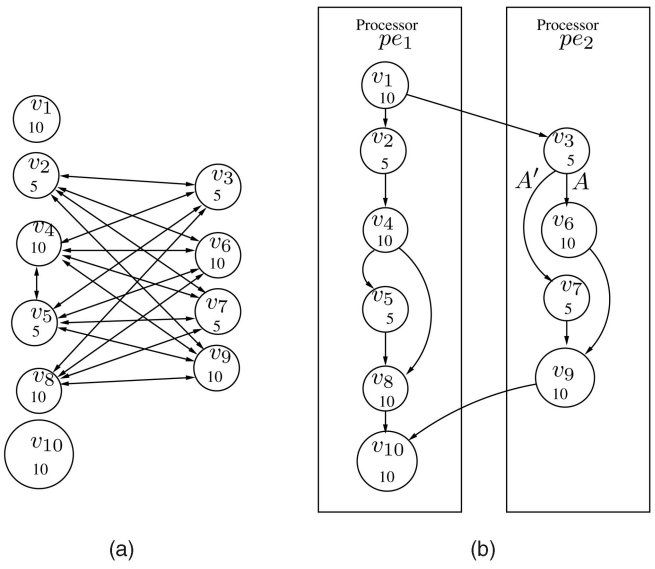

Fig. 4. Concurrency graph and schedule graph for Fig. 1.

It can be seen that G_S inherits all vertices and edges of the original G. Additionally, it contains edges between nodes that are concurrent to each other but are mapped on the same processor; that is, the start time of one node is affected by the finish time of the other node. For example, the schedule graph in Fig. 1a on to two processors (Fig. 1b) is shown in Fig. 4b. This schedule graph inherits all the edges of the graph shown in Fig. 1a. It also contains an edge between v_4 and v_5, because even though they are concurrent in G (Fig. 4a), they are mapped on the same processor pe_0. Hence, the start time of v_5 is affected by v_4. This behavior is added to the schedule graph as an edge, along with the dependency relation in G.

Step 2: elimination of unimportant nodes from further consideration of duplication. In this step, we identify and eliminate certain nodes from duplication based on some properties of the concurrency and schedule graphs derived from the initial schedule in Step 1:

Property 1. *If a task does not have any edge in the concurrency graph, then it need not be duplicated to other processors*. Note that the concurrency graph may be disconnected in nature. For example, in Fig. 4, v_1 and v_{10} are isolated nodes. Since there is no concurrent node with such an isolated node, duplicating this node on other processors does not provide any schedule length gain.

Property 2. *If a task v_i has at most $|PE|-1$ adjacent nodes in G_C and each adjacent node has a mapping in a different processor, then v_i need not be duplicated in any of the processors*. During an online list scheduling, all the nodes in the ready queue are concurrent to each other. The adjacent nodes for any node in G_C denote all possible nodes that can reside in the queue along with it. Let us consider a node v_i whose number of adjacent nodes is less than the number of processors ($|PE|-1$) and that all these nodes belong to different processors. Hence, when v_i becomes ready for execution, there can be only a maximum of $|PE|-1$ nodes in the queue. Since these nodes have mappings to different processors, irrespective of the duplication, they can be started at their earliest possible start time on the processors that they are initially mapped.

However, if we consider a task graph with a very low edge density mapped to a limited multiprocessor platform, *Properties 1* and *2* can eliminate only a few nodes. We therefore perform an approximate calculation to eliminate further nodes from duplication. Let us take a node v_i mapped on a processor $pe(v_i)$. Our aim here is to map v_i on a processor pe_j, where $pe_j \neq pe(v_i)$, and find out if the start time v_i can be improved in any execution behavior before $ready\_time(v_i)$, keeping the rest of the node processor mappings unchanged. If v_i can be started on pe_j earlier than $ready\_time(v_i)$, then there must be a node executing in parallel to v_i in $pe(v_i)$; otherwise, v_i could have been started in (pe_j) at the same time, eliminating the requirement of duplicating and executing v_i on a secondary processor pe_j. Thus, Property 3 is defined as follows:

Property 3. *If a node v_i does not have any adjacent node v_j in G_C such that the adjacent node is mapped to the same processor $pe(v_i)$, with $ST_{v_j} < ST_{v_i}$, v_i is eliminated from the duplication list.*

It can be noted that *Property 3* is stronger than both *Properties 1* and *2*. Hence, for implementation, *Property 3* alone is sufficient.

Step 3: weight calculation for each node processor mapping. From Property 3, we can conclude that each node selected for duplication has a concurrent node placed earlier than its schedule on the same processor. In the schedule graph, we have already represented this by adding an edge. If $C_{pe(v_i),v_i}$ is the set of such parent nodes of any node v_i, $Parent_{v_i}(G) \cup C_{pe(v_i),v_i} = Parent_{v_i}(G_S)$. Let us consider duplicating v_i on processor pe_j. Our aim is to find the *maximum* possible start time gain by node v_i when it is scheduled on pe_j, instead of $pe(v_i)$, under any possible execution state of the nodes occurring before v_i. Note that the execution of v_i can only be delayed in $pe(v_i)$ than $ready\_time(v_i)$ if any of the nodes in $C_{pe(v_i),v_i}$ finishes after $ready\_time(v_i)$. Hence, we generate a scenario where this delay becomes maximum when v_i is started in $pe(v_i)$. To achieve this, we assign an execution value of v_k to $WCET_{v_k}$ if $v_k \in C_{pe(v_i),v_i}$. We also assign v_l to $WCET_{v_l}$ for each $v_l \in Parent_{v_k}$. The rest of the nodes are assigned to their respective $BCET$ values. This assignment tries maximizing the delay of start of the execution of v_i in $pe(v_i)$. For this assignment, we calculate the maximum starting time Max_{v_i} of a node v_i by using the *MinMax* algorithm on G_S, as described in Section 4.1.2.

If we move the execution of v_i to pe_j, a new schedule graph $G_{S'}$) is formed. The dependency edges between $C_{pe(v_i),v_i}$ and v_i no longer exist. However, there is a new set formed C_{pe_j,v_i}, consisting of the concurrent nodes of v_i mapped earlier in pe_j. In the new schedule graph, $Parent_{v_i}(G) \cup C_{pe_j,v_i} = Parent_{v_i}(G_{S'})$.

For the above scenario, we find out the minimum possible start time (Min_{v_i}) by applying the *MinMax* algorithm (Algorithm 3) on $G_{S'}$, as described in Section 4.1.2. Hence, the maximum start time of the gain of v_i by executing on pe_j, instead of $pe(v_i)$, is

$$Gain(v_i \rightarrow pe_j) = Max_{v_i}(G_S) - Min_{v_i}(G_{S'}). \quad (1)$$

Note that this gain cannot be directly related to schedule the length gain of the whole task graph. Here, we propose to take the SU of a task to be the factor while prioritizing

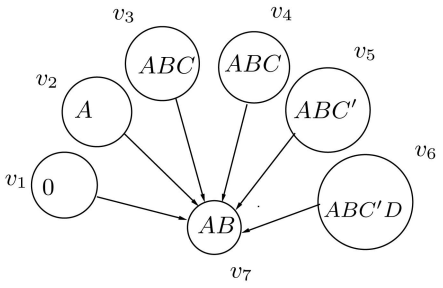

Fig. 5. MinMax example.

node duplication along with the gain. We define the weight of each task as

$$W_{v_i \to pe_j} = Gain(v_i \to pe_j) \times \frac{SU_{v_i}}{SU_{v_{source}}}. \quad (2)$$

The MinMax algorithm for a schedule graph. The *MinMax* algorithm calculates the minimum Min_{v_i} and the maximum Max_{v_i} possible start time for nodes in a schedule graph. The *MinMax* calculation for any node v_i ($v_i \in V(G_S)$) can be recursively defined as follows:

$$Max(v_i) = Maximum(Max_v + ex_v, \forall v \in Parent_{v_i}(G_S)), \quad (3)$$

$$\begin{aligned} Min_{v_i} = & Maximum(Min(H), \forall H \subset Parent_{v_i}(G_S) \\ & and\ occur_{v_i, LabelUnion(H)} = true), \\ where\ \ Min(H) = & Minimum(Min_v + ex_v, \forall v \in H). \end{aligned} \quad (4)$$

$LabelUnion(v_1, \ldots, v_n)$ is the union of the branch conditions of $v_1, \ldots, v_n$ (see Algorithm 2 and [6] for details on branch labeling in a CTG). $occur_{v_i, C_j}$ is true if $C_j \subset v_i.branch\_condition$; otherwise, it is false. If $occur_{v_i, C_j}$ is TRUE, it means that if node v_i executes, then all nodes of branch condition C_j must execute as well.

Fig. 5 illustrates the *MinMax* equation for a schedule graph with an example. The figure shows the parent nodes of v_7 ($v_1 - v_6$) in the schedule graph. The label inside each node denotes its branch labeling. Suppose that we want to find out Min_{v_7} and Max_{v_7} for node v_7. It can be seen that one or more nodes among $v_1, \ldots, v_6$ coexist with v_7 under different conditions. Hence, the maximum start time of v_7 under any condition is the maximum finish time among all of the parent nodes $(v_1, \ldots, v_6)$:

$$Max_{v_7} = Maximum(Max_v + ex_v, \forall v \in \{v1 - v6\}).$$

However, a similar equation for finding Min_{v_7} does not hold. Here, we have to find the minimum start time of v_7 under all possible conditions under which v_7 occurs. Nodes v_1 (condition 0) and v_2 (condition A) always coexist with v_7 (condition AB); hence, the minimum start time Min_{v_7} of v_7 is at least the maximum of the minimum finish time of the above nodes. It can be seen that when D' is evaluated, v_6 does not execute with v_7. Thus, we can eliminate it from the Min_{v_7} calculation completely. However, it can be observed that nodes v_3 and v_4 always coexist with each other and with v_7 whenever C is TRUE. Node v_5 exists with v_7 only when C' is TRUE, and for that instance, nodes v_3 and v_4 do not execute. This means that at least v_3, v_4, or v_5 will coexist with v_7. Hence, intuitively

$$\begin{aligned} Min_{v_7} = & Maximum(Min_{v_1} + ex_{v_1}, Min_{v_2} + ex_{v_2}, \\ & Minimum(Maximum(Min_{v_3} + ex_{v_3}, Min_{v_4} + ex_{v_4}), \\ & Min_{v_5} + ex_{v_5})). \end{aligned}$$

The same relation between v_7 and $(v1 - v6)$ can be obtained from (4). Here, for instance, $LabelUnion(v_3, v_5) = ABC \cup ABC' = AB$, which coexist with v_7.

Algorithm 3 describes the Min and Max start time calculation from a schedule graph G_S. The Min or Max function for any node v_i is expressed as a recursive function of its parents in G_S.

Algorithm 3: Calculate Min_{v_i} and Max_{v_i} for node $v_i \in G_S$.
The CTG G and Schedule Graph G_S are input to the scheduler.
$Min_{v_{source}} = 0$, and $Max_{v_{source}} = 0$
$Max_{v_i} \leftarrow Maximum(Min_v + ex_v, \forall v \in Parent_{v_i}(G_S))$
For a node $v_i \in G_S$, create two empty sets, $S1 = S2 = \emptyset$
for all $v \in Parent_{v_i}(G_S)$ **do**
  if $v.branch\_condition \subset v_i.branch\_condition$ {v always coexists with v_i} **then**
    $S1 \leftarrow S1 \cup \{v\}$
  else
    $S2 \leftarrow S2 \cup \{v\}$ {v may coexist with v_i under certain conditions}
  end if
end for
for all $v \in S2$ **do**
  Let $S_v \leftarrow \{v\} \cup mutex_{v_i} \cap S2$
  if $LabelUnion(S_v) \subset v_i.branch\_condition$ {one node within S_v must coexist with v_i} **then**
    $S_1 \leftarrow S_1 \cup \{v_m\}$, where $Min(v_m) + ex_{v_m}$ is minimum, and $v_m \in S_v$
  end if
end for
$Min_{v_i} \leftarrow Maximum(Min_v + ex_v, \forall v \in S_1)$

Step 4: duplication set selection. After Step 3, we obtain a set of node processor mapping vectors (like $v_i \to pe_j$) and their weight toward the duplication. We group together the set of node processor duplication vectors for each processor. Let this set be Dup_{pe_j} for a processor pe_j. The available code memory $AvailMem_{pe_j}$ for pe_j can be found out from Algorithm 1. Our aim here is to determine a subset of vectors $SubDup_{pe_j}$ from Dup_{pe_j} such that $\sum W_{v_i \to pe_j}$ is maximized and $\sum cs_{v_i} < AvailMem_{pe_j}$. The above formulation falls directly into the category of a 0-1 knapsack problem, where the following hold:

- The volume of the knapsack $U = AvailMem_{pe_j}$.
- The volume of each item $v_i \to pe_j \in Dup_{pe_j}$ is cs_{v_i}.
- The value of each item $v_i \to pe_j \in Dup_{pe_j}$ is $W_{v_i \to pe_j}$.

The solution to the knapsack gives a subset $SubDup_{pe_j}$, which is then mapped on the processor pe_j. The same procedure is carried out for all the processors. In this work, we have used a dynamic programming method to solve the knapsack, which is an exact algorithm. It can be noted here that the complexity of the dynamic programming is pseudopolynomial with respect to the inputs.

Hence, if the values of the number of nodes or code sizes are very high, an approximate knapsack solver like the one proposed by Sahni [20] can be used. This step completes the node processor mapping. It can be seen that the additional node processor mapping in the limited duplication does not disturb the initial mapping. A brief summary of the selective duplication is given in Algorithm 4.

Algorithm 4: Phase 2 (Selective Code Duplication).
The task graph and initial schedule are input to this algorithm
Construct a concurrency graph G_C and a schedule graph G_S {4.1.2}
Eliminate nodes using *Property 3* and a duplication node list Dup_G {4.1.2}
for all $v_i \in Dup_G$ **do**
  for all $pe_j \in PE \; and \; pe_j \neq pe(v_i)$ **do**
    Estimate $Gain(vi \rightarrow pe_j)$ (1) {Step 3 in Section 4.1.2}.
    Estimate weight $W_{v_i \rightarrow pe_j}$ ((2) in Section 4.1.2)
  end for
end for
for all $pe_j \in PE$ **do**
  $Dup_{pe_j} \leftarrow$ *set of all nodes to be duplicated on* pe_j
  Apply the 0-1 knapsack solver on Dup_{pe_j}, $AvailMem_{pe_j}$ and select $SubDup_{pe_j}$, which maximizes $W_{v_i \rightarrow pe_j}$
  $pe_j \leftarrow v_i \forall v_i \in SubDup_{pe_j}$ {Map each node in $SubDup_{pe_j}$ on pe_j}
end for

4.2 Phase 3: Online Scheduler

The global online scheduler maintains the application task graph and node-processor mapping information generated from the selective duplication process. It schedules the task nodes to processors at runtime. We also present a variation of the online scheduler that guarantees the deadline produced by the static scheduler during the initial task mapping (Algorithm 1).

4.2.1 Online Scheduler without Deadline Guarantee (Algorithm 5)

The scheduler keeps the statically processed *SU* information for each node and a $MaxHeap$, which keeps a ready node queue, with SU being the key. $FreeProcessorQ$ maintains the idle processor queue in the scheduler at any time t.

The scheduler is called at the initialization of the application execution, and it assigns the *source node* to a processor. When a processor finishes the assigned node execution at a certain time, it calls the global scheduler. At this call, the successor (child) nodes are added to the $MaxHeap$, and the processor freed is inserted into the $FreeProcessorQ$. The child of any node v_i depends upon the condition evaluated before the current time t. The scheduler then extracts the most urgent task from the $MaxHeap$ and tries scheduling it on a processor available in $FreeProcessorQ$. The extracted node can only be scheduled to a processor if its code resides in that processor. If no such mapping is found, then the scheduler extracts the next urgent task and attempts a similar scheduling. All the unscheduled nodes that are extracted by the scheduler are again inserted into the $MaxHeap$. The scheduling continues until the *sink node* finishes execution.

Algorithm 5: Phase 3 (Online Scheduling Strategy).
CTG $G(V_s, V_c, , E_s, E_c)$ and SU_{v_i} (from Algorithm 1) are the inputs.
$t = 0$
Insert $source$ into $MaxHeap$, Insert pe_i into $FreeProcessorQ \; \forall pe_i \in PE$.
while Not EndSchedule **do**
  for all v_i scheduled in pe_j and $t = ST^o_{v_i} + ex_{v_i}$ {v_i finishes execution} **do**
    Insert pe_j into $FreeProcessorQ$, Insert $child^o_{v_i}$ into $MaxHeap$ {Insert to the heap, with SU_{v_i} being the key}
  end for
  while *not empty* $FreeProcessorQ$ *and not empty* $MaxHeap$ **do**
    $v_t \leftarrow POP(MaxHeap)$
    if $\exists pe_t$ such that $pe_t \in FreeProcessorQ$ and $pe_t(v_t) = TRUE$ {Code of v_t must be present in the processor}
    then
      $ST^o_{v_t} \leftarrow t$, $pe_t \leftarrow sv_t$ {Schedule v_t on pe_t}
    else
      Insert v_t into $TempNodeQueue$ {$TempNodeQueue$ carries nodes that are popped from the MaxHeap but could not be scheduled to any processor}
    end if
  end while
  Insert v_i into $MaxHeap \; \forall v_i \in TempNodeQueue$ {Restores the unscheduled Ready nodes}
end while

4.2.2 Online Scheduling with Hard Deadline

The advantage of a static scheduling is that it provides a worst-case schedule length offline. We have observed that in the worst case, the schedule length of the online scheduler and static scheduler are almost similar. This is due to the fact that both are based on the list scheduling algorithm. However, in contrast to static scheduling, we cannot guarantee a schedule length in case of online scheduling. For hard real-time tasks, calculating an offline worst-case schedule length and always meeting this deadline are important criteria. Here, we propose a constraint that can be imposed on the above online scheduler to guarantee a schedule length that is at least as large as the schedule length generated from the initial static mapping. Note that each node has a primary schedule (the mapping and start time provided in Phase 1) and several secondary mappings (provided by the duplication algorithm). Hence, each processor has a canonical execution order based on the primary schedule of each node. The corresponding schedule graph G_S captures the schedule as a dependent task graph. Let us assume that the deadline that needs to be met is always greater than the worst-case schedule length in Phase 1. Hence, using the schedule graph G_S, we estimate the As Late As Possible (ALAP) time for each node. During an online scheduling, if a processor is about to schedule a secondary node, the online scheduler first ensures that the next primary node scheduled on the processor is meeting its ALAP time. Otherwise, it delays the

execution of the secondary node, which, in turn, in the worst case, may get scheduled on its primary processor. For example, in Fig. 4b, the ALAP time of node v_9 is 30. In Fig. 2b, at time $t = 19$, the online scheduler decides to schedule the secondary node v_5, because $t + WCET_{v_5} = 19 + 5 = 23$, which is less than 30. With the same argument, any node, including the $sink$ node in the worst case, gets scheduled on its primary processor at the start time generated from the initial schedule. Hence, this scheduler never violates the schedule length generated from the initial static mapping.

4.3 Complexity and Overhead

A complexity analysis for the selective duplication and the online scheduling algorithms is given as follows:

4.3.1 Selective Code Duplication

The complexity of calculating a mutual exclusion between two nodes is $O(|C|)$ [6]. Hence, the complexity of concurrency graph creation can be calculated as $O(|V|^2 \cdot |C|)$. If d_c is the maximum outdegree of any node in the concurrency graph G_C, then the complexity of schedule graph creation is $O(|V| \cdot d_c)$. The elimination procedure by *Property 3* is of the same complexity as schedule graph creation, that is, $O(|V| \cdot d_c)$. If d_s is the maximum indegree of any node in the schedule graph G_S, the complexity of the min calculation for a schedule graph becomes $O(|V| \cdot d_s^4 \cdot |C|^3)$. The complexity of the max calculation for G_S is $O(|V| \cdot d_s)$. For a node-processor gain calculation, we use the min calculation $|PE| - 1$ times and the max calculation once. Hence, the overall complexity of a selective duplication algorithm, except for Step 4, is

$$O(|PE| \cdot |V| \cdot d_s^4 \cdot |C|^3). \quad (5)$$

The complexity of calculating the duplication set for all processors with the 0-1 knapsack solver in Step 4 is

$$O(|PE| \cdot |V| \cdot AvailMem_{max}), \quad (6)$$

where $AvailMem_{max}$ is the maximum available memory in any processor.

We now discuss the complexity of the static phase, with some of the pure static scheduling techniques. Since the static scheduling problem is NP-complete, the complexity is exponential. Small-sized and medium-sized unconditional task graphs can be statically mapped with search-heuristic-based schedulers such as the algorithm proposed by Ahmad and Kwok [21]. However, a large number of pure static schedulers use list-scheduling-based schedulers, whose complexity is polynomial in nature. The complexity of the CTG scheduler in [6] is $O(|V||C|log(|V|)$. The complexity of the CTG scheduler by Eles et al. [4] is exponential with respect to the number of conditions in the task graph. The static task mapping proposed in this paper is dominated by the complexity of the knapsack solver (6), which may be exponential for an arbitrary-sized task graph and code size values, whereas the rest of the calculations can be seen as polynomial in nature. As mentioned earlier, an approximate algorithm like [20] can be used in such cases, which will restrict the calculations of the knapsack solver to a polynomial complexity $|PE| \cdot k \cdot |V|^{k+1}$, where k is a nonnegative integer that defines the order and the accuracy of the solution.

4.3.2 Online Scheduler

The online scheduler presented here may not be directly compared with the online schedulers like [12] and [13] in terms of performance due to major differences in the task graph and processor models. These methodologies also differ from each other in terms of the assumptions and their scopes. Nevertheless, we present a brief discussion on the overhead of the presented online scheduler in the context of overheads and complexity incurred in these techniques. The proposed methodology in [12] assumes an $N \times N$ grid of tasks, each of which is dependent on the prior execution of its two neighbor tasks to the left and above it. The overhead of such a graph scheduling has been found out to be $\Omega(loglog|V|)$ for $|PE|$ ($|PE| > 2$) processors. The above overhead is also a theoretical lower bound of the scheduling of the $N \times N$ grid graph. Similarly, an online priority list scheduler experiences $|V|log|V|$ worst-case overhead. Gupta et al. [13] used a representation, namely, a compact task graph model, for a low-overhead real-time scheduling. This algorithm can be applied in three levels, depending on the complexity tolerance desired. The first-level complexity of this technique is $O(N^2)$ for a limited number of processors. The overhead calculation of the proposed online scheduler in this paper is similar to an online priority list scheduler. Unlike the above techniques, the proposed online scheduler assumes limited duplication as opposed to a complete mapping, which increases the complexity of scheduling. We now derive and discuss the calculations.

Let s be the maximum size of the $MaxHeap$ maintained by the online scheduler. At each global scheduler call, there can be a maximum of s number of POP and $PUSH$ operations for task selection. With each POP operation, the code-processor mapping query is made on all the ready processors, and if a mapping is not found, then the node is again inserted into the $MaxHeap$. Hence, the worst-case overhead for every online scheduler call (signifying the application's worst-case stalling) is

$$(|PE| + 1) \cdot s \cdot log(s). \quad (7)$$

The worst-case overhead over the whole task graph execution is therefore

$$|V| \cdot (|PE| + 1) \cdot s \cdot log(s). \quad (8)$$

Note that the complexity can be further reduced by carefully designing the online scheduler. For example, by using a hash table for code-processor mapping information, the complexity reduces to

$$|V| \cdot s \cdot log(s). \quad (9)$$

The maximum size of the $MaxHeap$ s can be calculated as the maximum number of ready tasks that can come in parallel in the CTG. For a node v_i, no node in $mutex_{v_i}$, $ancestors_{v_i}$, and $descendants_{v_i}$ can coexist with v_i in $MaxHeap$. Hence

$$s = max \frac{|V - mutex_{v_i} - ancestors_{v_i} - descendants_{v_i}|}{|V|}. \quad (10)$$

The calculation of $MaxHeap$ also helps in deciding the queue length of the online scheduler beforehand for the targeted task graph. The worst-case scheduler overhead per call is dependent on the graph structure, as expected, but is

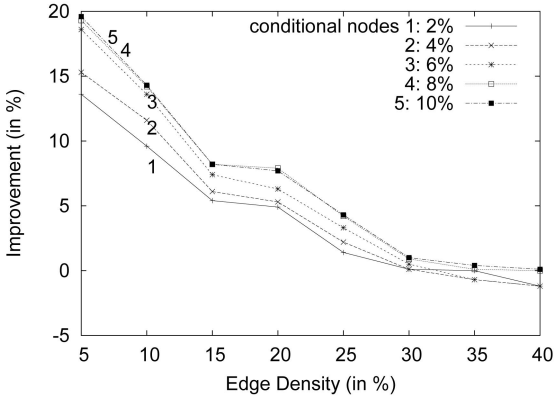

Fig. 6. Schedule length improvement (online over static scheduling, in percent) versus edge density (percentage of *max edge*) for a number of conditions (percentage of $|V|$), $|PE| = 3$, and $|V| = 200$.

independent of the number of conditions in the CTG. Hence, the strategy is suitable, even for control-intensive CTGs.

5 Experimental Results

The methodology proposed in this paper is designed for task graphs, which are dynamic in nature due to two factors, namely, 1) conditional nodes and 2) task nodes with unpredictable execution timings. We evaluate the proposed duplication and online scheduling algorithm for each of the above factors separately. First, we experiment on CTGs with a fixed execution time. Next, we evaluate our algorithm on unconditional task graphs with nodes of an unpredictable execution time.

5.1 Conditional Task Graph with Nodes of a Fixed Execution Time

5.1.1 Setup

We have modified the random task graph generation model defined by Johnsonbaugh and Kalin [22] and used it for generating the random CTGs. The parameters of the task graph generation process, namely, $|V|$, $|C|$, and *edge density*, were varied to generate a wide range of random CTGs. The number of nodes $|V|$ was varied from 100 to 500, and the *edge density* was from 5 percent to 50 percent. The number of conditions $|C|$ (expressed as a fraction of $|V|$) was varied from 2 percent to 10 percent. For each combination of parameters, 100 random CTGs were generated. Random execution cost and code size values were assigned between 5 to 15 units on each node. This way, we generated 12,500 random task graphs, producing different structures. The effect of different parameters were studied experimentally on the generated task graph set.

5.1.2 Experiments

The schedule length generated by the proposed strategy was compared to the well-known static scheduler proposed by Xie and Wolf [6]. In all cases, resource reclaim was performed on each processor while determining the schedule lengths for specific conditions. The average-case schedule improvement was calculated for the simulation of a maximum of 1,000 random instances among all the possible $2^{|C|}$ conditions for each task graph. If L_i is the schedule length produced by [6] with resource reclaim and L'_i is the schedule length produced by the proposed

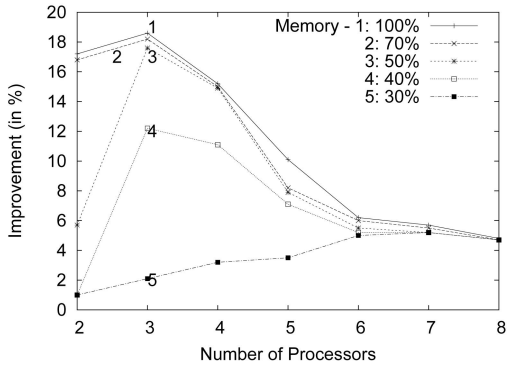

Fig. 7. Schedule length improvement (online over static scheduling, in percent) versus processors for different local memory sizes (percentage of $\sum cs_{v_i}$), $|V| = 200$, $|C| = 4$ percent, and edge density $= 5$ percent.

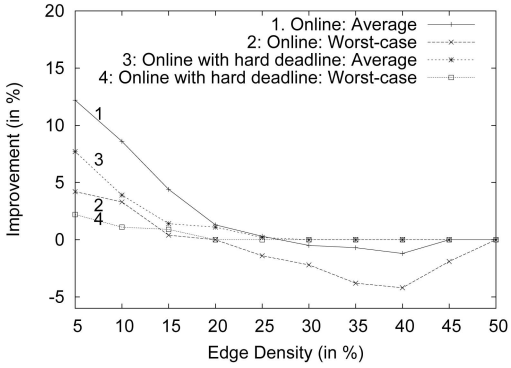

Fig. 8. Average/Worst-case improvement (online over static scheduling, in percent) versus edge density (percentage of *max edge*) for an online scheduler with and without hard deadline. $|PE| = 3$, $|V| = 200$, $|C| = 4$ percent, $MEM = 40$ percent.

strategy for the ith instance of a total of I instances of a task graph G execution, the average-case schedule improvement is calculated as

$$\%Av\_Improv_G = \frac{\sum_{i=1}^{I} \frac{(L_i - L'_i) \times 100}{L_i}}{I}. \quad (11)$$

The average improvement for a task graph set of N task graphs is calculated as

$$\%Improvement = \frac{\sum_{i=1}^{N} \%Av\_Improv_{G_i}}{N}. \quad (12)$$

In the first experiment, the processor local memory constraint was not applied, and the improvement was studied for a variation of the number of conditions in the task graph (Fig. 6). The processor local memory was then constrained up to 30 percent of the maximum memory required (Fig. 7). The number of processing elements $|PE|$ was then varied to study its effect on the schedule improvement for different memory constraints (Fig. 7). The worst-case improvement/degradation was also noted for the simulation of the above (Fig. 8).

5.1.3 Results

Fig. 6 shows the improvement on schedule length of the dynamic scheduler over the static scheduler for varying edge density and conditions, with no restriction on the processor local memory. It can be observed that with an

TABLE 1
Percent Improvement versus $|V|$ ($|PE| = 3$)

$\|V\|$	% Improvement		
	Max	Mean	Min
100	18.4	9.6	-1.2
200	19.9	9.5	-1.8
300	20.0	9.7	-1.5
400	20.8	9.9	-2.2
500	21.9	10.1	-2.1

increasing number of conditions, the average schedule length is improved. With a low edge density, the improvement is close to 20 percent for a 200-node 8 percent condition graph and a three-homogeneous-processor system, and it decreases with the edge density. As the edge density between the nodes increases, the nodes in the task graph tend to execute in a serialized order, hence restricting the scope of improvement in schedule length. We have observed that the improvement in schedule length scales with the task graph size. However, the average percentage of improvement (the fraction of the total schedule length) increases slowly or can be considered to be practically constant. This means that a higher saving in schedule length is achieved in a large-sized task graph, typically with a high schedule length but with a percentage of the average schedule length gain being the performance criteria, whereas all the graphs with 100-500 node sizes remain practically identical to each other (Fig. 6). Instead of presenting separate graphs like Fig. 6 for different graph sizes, the characteristics of those graphs (maximum, mean, and minimum improvement) are summarized in Table 1.

Fig. 7 shows the improvement of schedule length versus the number of processors and the local memory assigned to each processor. It can be observed that for a three-processor system, a considerable schedule length improvement can be achieved, even with a 40 percent memory in each processor, and it is comparable to an improvement with a 100 percent memory in each processor. As the local memory is increased in each processor, the duplication strategy can accommodate more nodes, and the dynamic scheduler performs better. As defined earlier, a 100 percent code duplication means complete code mirroring, and it provides scope for the maximum schedule length gain over the static scheduler. When the processor model assumes an infinite number of processors, all the tasks are scheduled at the earliest possible start time by both static and online scheduling strategies, providing an optimal schedule length. Hence, with an increase in the number of processors, the scope for the improvement of the online scheduler over the static scheduler reduces. The experiment clearly shows that the proposed methodology is best suited for architectural models with fewer number of processors, typically lying between 2 and 6. Architectures targeted for embedded applications like video encoders, MP3 decoders, and motion detectors usually contain a limited number of processors that typically fall in between the range.

If the memory assigned to each local processor is much less, the selective duplication stage yields very few duplications, and the online scheduler flexibility of a processor selection becomes entirely restricted. Hence, the online scheduler without hard deadline may show a degradation in schedule length for some instances among the $2^{|C|}$ conditions, particularly for a high edge density, where the scope of improvement is limited. The worst-case improvement/degradation is noted along with the average-case improvement for all the generated conditions during the experiment. Fig. 8 shows the worst-case improvement of the dynamic scheduler over a static scheduler. The negative y-axis represents the degradation. Fig. 8 also presents the results for an online scheduler with the deadline guarantee scheduler discussed earlier. It can be observed that the 4 percent degradation observed earlier is not more than with the later scheduler. However, it can be seen that the imposed constraint in the deadline guarantee scheduler affects the average improvement negatively. For a 200-node 4 percent condition graph and a three-homogeneous processor system, the average improvement reduces from 12 percent to 8 percent, roughly with the imposed constraint.

5.2 Unconditional Task Graph with Unpredictable Execution Time

We now evaluate our strategy on unconditional task graphs with nodes of varying execution time.

5.2.1 Setup

The experimental setup was identical to that of CTGs, except that we kept the number of conditions of the task graph to be zero. The randomly generated execution time for each node earlier is considered as WCET in this setup. We define aparameter α $(0 < \alpha < 1)$ as $\alpha = \frac{BCET_{v_i}}{WCET_{v_i}} \forall v_i \in V(G)$.

For each value of α, we calculated the value of BCET for each node. Then, at runtime, we assigned a random value between BCET and WCET to each node. Our aim here was to make all the tasks equally dynamic. For each task graph, we did 1,000 different runs and found out the average schedule length by using (12).

5.2.2 Experiments

In Phase 2, we mentioned that the proposed Phase-1 scheduler is not mandatory for selective duplication. In these experiments, we used well-known the dynamic critical path (dcp) algorithm [3] in Phase 1. We compared the results with the dcp schedule length with resource reclaim at runtime. We compared our method to the schedule length generated from above. We performed similar experiments to the previous section, except that we varied α from 0.2 to 0.8 in place of the number of conditions.

5.2.3 Results

Fig. 9 shows the improvement on schedule length of the dynamic scheduler over the static scheduler for varying edge density and conditions, with no restriction on the processor local memory. It can be observed that with a decreasing α, the average schedule length is improved. With a higher α, the task graph behaves like an unconditional fixed-execution-time task graph; hence, the static scheduling performs equally well. When the value of α is low, the task graph becomes more dynamic and suits the proposed method. The variation of the maximum, mean, and minimum average schedule length improvement with the number of nodes is tabulated in Table 2.

Fig. 10 shows the improvement of schedule length versus the number of processors and local memory assigned to

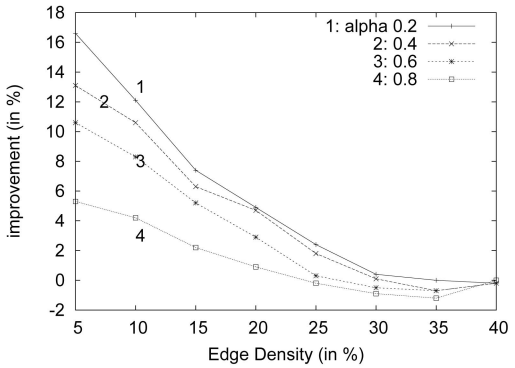

Fig. 9. Schedule length improvement (online over static scheduling, in percent) versus edge density (percentage of *max edge*) for different alpha α ($|PE| = 3$, and $|V| = 200$).

TABLE 2
Percent Improvement versus $|V|$ ($|PE| = 3$)

$\|V\|$	% Improvement		
	Max	Mean	Min
100	16.4	7.4	-1.9
200	16.6	7.7	-1.7
300	17.0	7.6	-2.0
400	17.7	7.9	-2.1
500	18.9	8.3	-1.9

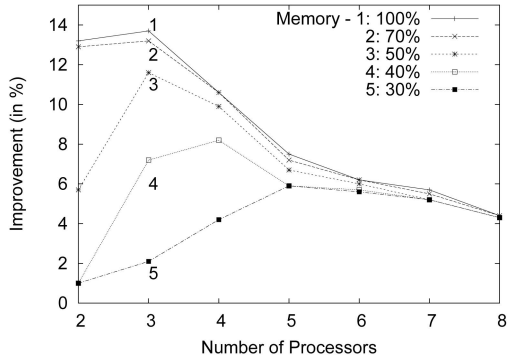

Fig. 10. Schedule length improvement (online over static scheduling, in percent) versus processors for different local memory sizes (percentage of $\sum cs_{v_i}$), $|V| = 200$, $\alpha = 0.5$, and edge density $= 5$ percent.

each processor for a task graph, with $\alpha = 0.5$. Finally, Fig. 11 presents the online scheduler's worst-case performance and the average/worst-case performance of the online scheduler with hard deadline.

5.3 Combined Results

The experiments performed above indicate that the proposed methodology is suitable for task graphs that have a higher number of conditions, a higher parallelism, and a significant nondeterminism in the execution time of its nodes. To study the effectiveness of the proposed technique, we have used the application task graphs extracted from the MI benchmark applications [19]. The task graphs are extracted manually by identifying the macro code blocks inside the application, and the parameters like WCET and BCET of each task is characterized by running the application multiple times on the SimpleScalar PISA Simulator [23]. The results are tabulated in Table 3. Here, the α column denotes the average α of all the tasks.

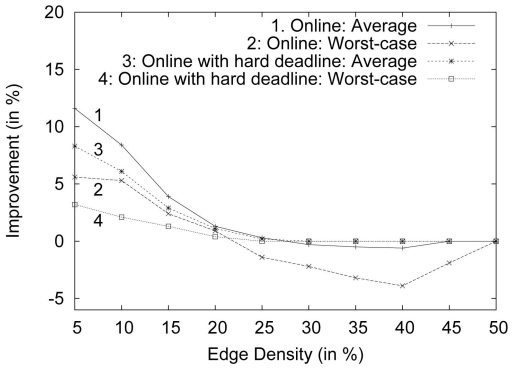

Fig. 11. Average/Worst-case improvement (online over static scheduling, in percent) versus edge density (percentage of *max edge*) for online scheduler with and without hard deadline. $|PE| = 3$, $|V| = 200$, $\alpha = 0.4$, and $MEM = 50]$; percent.

TABLE 3
Benchmark Applications

Name	$\|V\|$	$\|E\|$	$\|C\|$	α	% Improvement
adpcm	15	18%	0%	0.89	0
basicmath	77	30%	8%	0.78	8.6
crc32	10	45%	10%	0.91	3.5
dijkstra	84	12%	4%	0.74	7.9
fft	21	24%	10%	0.66	11.2
sha	115	8%	10%	0.62	14.3

TABLE 4
Overhead versus Task Graph Size. $|PE| = 4$, $|C| = 6$ Percent, Edge Density $= 5$ Percent

$\|V\|$	Computation Time (Static) (milli-seconds)	Cumulative overhead (Online) (milli-seconds)	Overhead per node (Online) (micro-seconds)
100	793.2	0.096	0.96
200	944.6	0.146	0.73
300	1070.8	0.669	2.23
400	2135.5	1.240	3.10
500	3181.1	2.820	5.64

5.4 Computation Time and Overhead Results

The computation time of the static phase and the overhead of the online scheduler are tabulated in Table 4. It can be observed that the static computation time (first two phases) increases with the number of nodes, as expected; however, it stays well within acceptable limits (in seconds) for the experiments performed. The overhead of the online scheduler was obtained by inserting time monitoring codes in the online scheduler. Variation of the overhead with respect to the size of the task graph is included in Table 4. In Table 4, we observe that the online scheduling overhead over the whole task graph increases with respect to the size of the task graph. However, the average time for scheduling a node (scheduling overhead) stays in microseconds. This is sufficiently low, since node execution times for macro data-flow graphs are typically in the range of seconds and go up to a few minutes.

6 Conclusions

This paper has presented a novel hybrid strategy for the mapping and scheduling of dynamic task graphs with conditional tasks and tasks with unpredictable execution behavior, which improves the average schedule length over

static schedulers. The strategy employs a static analysis approach that mirrors task nodes on a set of memory-constrained processors in a multiprocessor system based on a statically generated schedule such that the memory constraint on each processor is met. An online scheduling strategy then assigns the start times for the mapped nodes at runtime. Experimental results show that the proposed solution provides better average schedule length at runtime over the static schedulers for task graphs containing conditional nodes and nodes with unpredictable execution behavior. For memory-constrained multiprocessor platforms, the trade-off between the schedule length improvement and the processor local memory capacity has been investigated. We have also presented the effect of other model parameters on schedule length improvement. The algorithm presented in this paper may be suitably extended for heterogeneous processor models or may be integrated with the existing duplication strategies for highly communicating task graphs.

ACKNOWLEDGMENTS

The authors thank the reviewers for their valuable comments.

REFERENCES

[1] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys,* vol. 31, no. 4, pp. 406-471, 1999.
[2] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing,* vol. 18, no. 2, pp. 244-257, 1989.
[3] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, 1996.
[4] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems," *Proc. Conf. Design Automation and Test in Europe (DATE '98),* pp. 132-139, 1998.
[5] M. Grajcar, "Conditional Scheduling for Embedded Systems Using Genetic List Scheduling," *Proc. 13th Int'l Symp. System Synthesis (ISSS '00),* pp. 123-128, 2000.
[6] Y. Xie and W. Wolf, "Allocation and Scheduling of Conditional Task Graph in Hardware/Software Co-Synthesis," *Proc. Conf. Design Automation and Test in Europe (DATE '01),* pp. 620-625, 2001.
[7] K.M. Chandy and P.F. Reynolds, "Scheduling Partially Ordered Tasks with Probabilistic Execution Times," *Proc. Fifth ACM Symp. Operating Systems Principles (SOSP '75),* pp. 169-177, 1975.
[8] D.L. Rhodes and W. Wolf, "Co-Synthesis of Heterogeneous Multiprocessor Systems Using Arbitrated Communication," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design,* pp. 339-342, 1999.
[9] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *J. Parallel and Distributed Computing,* vol. 9, no. 2, pp. 138-153, 1990.
[10] R. Ernst and W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design,* pp. 598-604, 1997.
[11] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng, "Optimal Online Scheduling of Parallel Jobs with Dependencies," *Proc. 25th Ann. ACM Symp. Theory of Computing (STOC '93),* pp. 642-651, 1993.
[12] R. Anderson, P. Beame, and W. Ruzzo, "Low Overhead Parallel Schedules for Task Graphs," *Proc. Second Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA '90),* pp. 66-75, 1990.
[13] R. Gupta, D. Mosse, and R. Suchoza, "Real-Time Scheduling Using Compact Task Graphs," *Proc. 16th Int'l Conf. Distributed Computing Systems (ICDCS '96),* p. 55, 1996.
[14] N. Fisher, J. Anderson, and S. Baruah, "Task Partitioning upon Memory-Constrained Multiprocessors," *Proc. 11th IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA '05),* pp. 416-421, 2005.
[15] 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, 1998.
[16] S. Bansal, P. Kumar, and K. Singh, "An Improved Duplication Strategy for Scheduling Precedence-Constrained Graphs in Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 6, pp. 533-544, 2003.
[17] S. Bansal, P. Kumar, and K. Singh, "Dealing with Heterogeneity through Limited Duplication for Scheduling Precedence-Constrained Task Graphs," *J. Parallel and Distributed Computing,* vol. 65, no. 6, pp. 479-491, 2005.
[18] C.I. Park and T.Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," *IEEE Trans. Computers,* vol. 51, no. 4, pp. 444-448, 2002.
[19] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "MiBench: A Free Commercially Representative Embedded Benchmark Suite," *Proc. Fourth Ann. IEEE Workshop Workload Characterization (WWC),* http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf, Dec. 2001.
[20] S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem," *J. ACM,* vol. 22, no. 1, pp. 115-124, 1975.
[21] I. Ahmad and Y.-K. Kwok, "Optimal and Near-Optimal Allocation of Precedence-Constrained Tasks to Parallel Processors: Defying the High Complexity Using Effective Search Techniques," *Proc. Int'l Conf. Parallel Processing (ICPP '98),* pp. 424-431, 1998.
[22] R. Johnsonbaugh and M. Kalin, "A Graph Generation Software Package," *Proc. 22nd Technical Symp. Computer Science Education (SIGCSE '91),* pp. 151-154, 1991.
[23] T.M. Austin, E. Larson, and D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, no. 2, pp. 59-67, 2002.

Pravanjan Choudhury received the BTech degree in instrumentation engineering from the Indian Institute of Technology (IIT), Kharagpur, in 2002. He is currently working toward the PhD degree in computer science and engineering in the Department of Computer Science and Engineering, IIT, Kharagpur, where he has been a research consultant for the CAD Research Group since August 2003. He was with Himachal Futuristic Communication Limited R&D as a senior engineer. His research interests include multiprocessor systems, scheduling, CAD, and embedded systems.

Rajeev Kumar received the MTech degree in computer science and engineering from the University of Roorkee (now the Indian Institute of Technology (IIT), Roorkee) in 1992 and the PhD degree in computer science and engineering from the University of Sheffield in 1997. He is currently a professor of computer science and engineering in the Department of Computer Science and Engineering, IIT, Kharagpur. Prior to joining IIT, he was with the Birla Institute of Technology & Science, Pilani and the Defense Research and Development Organization. His research interests include multimedia and embedded systems, programming languages and software engineering, and multiobjective combinatorial optimization. He is a senior member of the IEEE.

P.P. Chakrabarti received the BTech and PhD degrees in computer science and engineering from the Indian Institute of Technology (IIT), Kharagpur, in 1985 in 1988, respectively. He joined the Department of Computer Science and Engineering, IIT, as a faculty member in 1988 and is currently a professor in the Computer Science and Engineering Department. He is also the dean of the Sponsored Research and Industrial Consultancy. His research interests include artificial intelligence, CAD for VLSI, and algorithm design. He is a senior member of the IEEE and a fellow of the Indian National Science Academy, the Indian Academy of Science, and the West Bengal Academy of Science and Technology. He received the President of India Gold Medal, the Swarnajayanti Fellowship, and the Shanti Swarup Bhatnagar Prize from the Government of India for his contributions.