Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors

Vida Kianzad, *Student Member*, *IEEE*, and Shuvra S. Bhattacharyya, *Fellow*, *IEEE*

Abstract—Multiprocessor mapping and scheduling algorithms have been extensively studied over the past few decades and have been tackled from different perspectives. In the late 1980's, the two-step decomposition of scheduling—into clustering and cluster-scheduling—was introduced. Ever since, several clustering and merging algorithms have been proposed and individually reported to be efficient. However, it is not clear how effective they are and how well they compare against single-step scheduling algorithms or other multistep algorithms. In this paper, we explore the effectiveness of the two-phase decomposition of scheduling and describe efficient and novel techniques that aggressively streamline interprocessor communications and can be tuned to exploit the significantly longer compilation time that is available to embedded system designers. We evaluate a number of leading clustering and merging algorithms using a set of benchmarks with diverse structures. We present an experimental setup for comparing the single-step against the two-step scheduling approach. We determine the importance of different steps in scheduling and the effect of different steps on overall schedule performance and show that the decomposition of the scheduling process indeed improves the overall performance. We also show that the quality of the solutions depends on the quality of the clusters generated in the clustering step. Based on the results, we also discuss why the parallel time metric in the clustering step may not provide an accurate measure for the final performance of cluster-scheduling.

Index Terms—Interprocessor communication, multiprocessor systems, scheduling, task partitioning.

1 Introduction

This research addresses the two-phase method of scheduling that was introduced by Sarkar [27] in which task clustering is performed in advance of the actual task to processor mapping and scheduling process and as a compile-time preprocessing step. This method, while simple, is a remarkably capable strategy for mapping task graphs onto embedded multiprocessor systems that aggressively streamlines interprocessor communication. In most of the follow-up work, the focus has been on developing simple and fast algorithms (e.g., mostly constructive algorithms that choose a lower complexity approach over a potentially more thorough one with a higher complexity, and that do not revisit their choices) for each step [14], [21], [26], [31] and relatively little work has been done on developing algorithms at the other end of the complexity/solution-quality trade-off (i.e., algorithms such as genetic algorithms that are more time consuming but have the potential to compute higher quality solutions). To the authors' best knowledge, there has also been little work on evaluating the idea of decomposition or comparing scheduling algorithms that are composed of clustering and merging (i.e., two-step scheduling algorithms) against each other or against one-step scheduling algorithms.

Embedded multiprocessor systems are typically designed as final implementations for *dedicated functions*; modifications to embedded system implementations are rare, and this allows embedded system design tools to employ more thorough, time-intensive optimization techniques. In contrast, multiprocessor mapping strategies for general purpose systems are typically designed with low to moderate complexity as a constraint [23]. Based on this observation, we took a new look at the two-step decomposition of scheduling in the context of embedded systems and developed an efficient evolutionary-based clustering algorithm (called CFA) that was shown to outperform the other leading clustering algorithms [10]. We also introduced a randomization approach to be applied to deterministic algorithms so they can exploit increases in additional computational resources (compile time tolerance) to explore larger segments of the solution space. This approach also provides a method for comparing the guided-random search algorithms against deterministic algorithms in a fair setup and is employed in this paper.

Most existing merging techniques do not consider the timing and ordering information generated in the clustering step. In this work, we have modified the ready-list scheduling algorithm to schedule groups of tasks or clusters instead of individual tasks. Our algorithm utilizes the information from the clustering step and uses the tasks' starting times that are determined during the clustering step to assign priorities to the clusters. We call the employed merging algorithm the Clustered Ready List Scheduling Algorithm (CRLA).

Our contribution in this paper is as follows: We first evaluate a number of leading clustering algorithms such as CFA [10], Sarkar's Internalization Algorithm (SIA) [27], and Yang and Gerasoulis's Dominant Sequence Clustering

- *V. Kianzad is with the Harvard Medical School, 330 Brookline Avenue, BIDMC SL-B05, Boston, MA 02215. E-mail: vida@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, MD 20742. E-mail: ssb@eng.umd.edu.*

Manuscript received 15 Dec. 2003; revised 27 Dec. 2004; accepted 6 June 2005; published online 25 May 2006.
Recommended for acceptance by U. Ramachandran.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0235-1203.

(DSC) algorithm [31] in conjunction with a cluster-scheduling or merging algorithm called CRLA and show that the choice of clustering algorithm can significantly change the overall performance of the scheduling. We address the potential inefficiency implied in using the two phases of clustering and merging with no interaction between the phases and introduce a solution that, while taking advantage of this decomposition, increases the overall performance of the resulting mappings. We present a general framework for the performance comparison of guided random-search algorithms against deterministic algorithms and an experimental setup for the comparison of one-step against two-step scheduling algorithms. This framework helps to determine the importance of different steps in the scheduling problem and the effect of different approaches in the overall performance of the scheduling. We show that decomposition of the scheduling process improves the overall performance and that the quality of the solutions depends on the quality of the clusters generated in the clustering step. We also discuss why the parallel execution time metric is not a sufficient measure for performance comparison of clustering algorithms.

This paper is organized as follows: In Section 2, we present the background and definitions used in this paper. In Section 3, we state the problem and our proposed framework. Experimental results are given in Section 4 and we conclude in Section 5 with a summary of the paper.

2 Background and Problem Statement

We represent the applications that are to be mapped into parallel implementations in terms of the *task graph model*. A task graph is a directed acyclic graph (DAG) $G = (V, E)$, where

- V is the set of task nodes, which are in one-to-one correspondence with the computational tasks in the application ($V = \{v_1, v_2, \ldots, v_{|V|}\}$),
- E is the set of communication edges (each member is an ordered pair of tasks),
- $t : V \rightarrow \aleph$ denotes a function that assigns an execution time to each member of V, and
- $C : V \times V \rightarrow \aleph$ denotes a function that gives the cost (latency) of each communication edge. That is, $C(v_i, v_j)$ is the cost of transferring data between v_i and v_j if they are assigned to different processors and $C(v_i, v_j) = 0$ if v_i and v_j are assigned to the same processor.

2.1 Scheduling and Clustering

The concept of *clustering* has been broadly applied to numerous applications and research problems such as parallel processing, load balancing, and partitioning [27], [20], [22]. Clustering is also often used as a front-end to multiprocessor system synthesis tool and as a compile-time preprocessing step in mapping parallel programs onto multiprocessor architectures. In this research, we are only interested in the latter context, where, given a task graph and an infinite number of fully connected processors, the objective of clustering is to assign tasks to processors. In this context, clustering is used as the first step to scheduling parallel architectures and is used to group basic tasks into subsets that are to be executed on the same processor. Once the clusters of tasks are formed, the task execution ordering of each processor will be determined and tasks will run sequentially on each processor with zero intracluster overhead. The justification for clustering is that if two tasks are clustered together and are assigned to the same processor when an unbounded number of processors are available, then they should be assigned to the same processor when the number of processors is finite [27].

In general, regardless of the employed communication network model, in the presence of heavy interprocessor communication, clustering tends to adjust the communication and computational time by changing the granularity of the program and forming coarser grain graphs. A perfect clustering algorithm is considered to have a decoupling effect on the graph, i.e., it should cluster tasks that are heavily dependent (data dependency is relative to the amount of data they exchange or the communication cost) together and form composite nodes that can be treated as nodes in another task graph. After performing clustering and forming the new graph with composite task nodes, there has to be a scheduling algorithm to map the new and simpler graph to the final target architecture. To satisfy this, clustering and list scheduling (and a variety of other scheduling techniques) are used in a complementary fashion in general. Consequently, clustering is typically first applied to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments.

The clustering goal (as well as the overall goal for this decomposition scheme) is to minimize the parallel execution time while mapping the application to a given target architecture. The **parallel execution time** (or simply parallel time) is defined by the following expression:

$$\tau_P = max(tlevel(v_x) + blevel(v_x) \mid v_x \in V), \qquad (1)$$

where $tlevel(v_x)$ ($blevel(v_x)$) is the length of the longest path between node v_x and the source (sink) node in the *scheduled graph*, including all of the communication and computation costs in that path, but excluding $t(v_x)$ from $tlevel(v_x)$. Here, by the scheduled graph, we mean the task graph with all known information about clustering and task execution ordering modeled using additional zero-cost edges. In particular, if v_1 and v_2 are clustered together, and v_2 is scheduled to execute immediately after v_1, then the edge (v_1, v_2) is inserted in the scheduled graph.

In the context of embedded system implementation, one limitation shared by many existing clustering and scheduling algorithms is that they have been designed for general purpose computation. In the general-purpose domain, there are many categories of applications for which short compile time is of major concern. In such scenarios, it is highly desirable to ensure that an application can be mapped to an architecture within a matter of seconds. Thus, some prominent examples of existing clustering algorithms such as DSC [31], Linear clustering [11], and SIA [27] have also been designed with low computational complexity as a major goal. However, in embedded application domains, such as signal/image/video processing, the quality of the synthesized solution is by far the most dominant consideration, and designers of such systems can often tolerate compile times on the order of hours or even days—if the synthesis results are markedly better than those offered by low complexity techniques. We have explored a number of approaches for exploiting this increased compile-time-tolerance. These will be presented in Sections 3.1 and 3.2.

In this paper, we assume a clique topology for the interconnection network where any number of processors can perform interprocessor communication simultaneously. We also assume dedicated communication hardware that allows communication and computation to be performed concurrently and we also allow communication overlap for tasks residing in one cluster.

2.2 Existing Approaches

IPC-conscious scheduling algorithms have received high attention in the literature and a great number of them are based on the framework of clustering algorithms [27], [31], [14], [17]. This group of algorithms, which are the main interest of this paper, have been considered as scheduling heuristics that directly emphasize reducing the effect of IPC to minimize the parallel execution time.

As introduced in Section 2.1, Sarkar's clustering algorithm has a relatively low complexity. This algorithm is an edge-zeroing refinement technique that builds the clustering step-by-step by examining each edge and clustering it only if the parallel time is not increased. Due to its local and greedy choices, this algorithm is prone to becoming trapped in a poor search space. DSC builds the solution incrementally as well. It makes changes with regard to the global impact on the parallel execution time, but only accounts for the local effects of these changes. This can lead to the accumulation of suboptimal decisions, especially for large task graphs with high communication costs, and graphs with multiple critical paths. Nevertheless, this algorithm has been shown to be capable of producing very good solutions, and it is especially impressive given its low complexity.

In comparison to the high volume of research work on the clustering phase, there has been little research on the cluster-scheduling or merging phase [16]. Among a few merging algorithms are Sarkar's task assignment algorithm [27] and Yang's Ready Critical Path (RCP) algorithm [29]. Sarkar's merging algorithm is a modified version of list scheduling with tasks being prioritized based on their ranks in a topological sort ordering. This algorithm has a relatively high time complexity. Yang's merging algorithm is part of the scheduling tool PYRROS [30] and is a low complexity algorithm based on the load-balancing concept. Since merging is the process of scheduling and mapping the clustered graph onto the target embedded multiprocessor system, it is expected to be as efficient as a scheduling algorithm that works on a nonclustered graph. Both of these algorithms lack this motive by oversimplifying assumptions such as assigning an ordering-based priority and not utilizing the (timing) information provided in the clustering step. A recent work on physical mapping of task graphs into parallel architectures with arbitrary interconnection topology can be found in [12]. A technique similar to Sarkar's has been used by Lewis and El-Rewini as well in [18]. GLB and LLB [26] are two cluster-scheduling algorithms that are based on the load-balancing idea. Although both algorithms utilize timing information, they are not efficient in the presence of heavy communication costs in the task graph. GLB also makes local decisions with respect to cluster assignments which results in poor overall performance.

Due to the deterministic nature of SIA and DSC, neither can exploit the increased compile time tolerance in embedded system implementation. There has been some research on scheduling heuristics in the context of compile-time efficiency [19], [14]; however, none studies the implications from the compile time tolerance point of view. Additionally, since they concentrate on deterministic algorithms, they do not exploit compile time budgets that are larger than the amounts of time required by their respective approaches.

There has been some probabilistic search implementation of scheduling heuristics in the literature, mainly in the forms of genetic algorithms (GA). The genetic algorithms attempt to avoid getting trapped in local minima. Hou et al. [8], Wang and Korfhage [32], Kwok and Ahmad [15], Zomaya et al. [35], and Correa et al. [3] have proposed different genetic algorithms in the scheduling context. Hou et al. and Correa et al. use similar integer string representations of solutions. Wang and Korfhage use a two-dimensional matrix scheme to encode the solution. Kwok and Ahmad also use integer string representations, and Zomaya et al. use a matrix of integer substrings. An aspect that all of these algorithms have in common is a relatively complex solution representation in the underlying GA formulation. Each of these algorithms must check at each step for the validity of the associated candidate solution and any time basic genetic operators (crossover and mutation) that are applied, a correction function needs to be invoked to eliminate illegal solutions. This overhead also occurs while initializing the first population of solutions. These algorithms also need to significantly modify the basic crossover and mutation procedures to be adapted to their proposed encoding scheme. We show that in the context of the clustering/merging decomposition, these complications can be avoided in the clustering phase, and more streamlined solution encodings can be used for clustering.

Correa et al. address compile time consumption in the context of their GA approach. In particular, they run the lower-complexity search algorithms as many times as the number of generations of the more complex GA, and compare the resulting compile times and parallel execution times (schedule makespans). However, this measurement provides only a rough approximation of compile time efficiency. A more accurate measurement can be developed in terms of fixed compile-time budgets (instead of fixed numbers of generations). This will be discussed further in Section 3.2.

As for the complete two-phase implementation, there is also a limited body of research work providing a framework for comparing the existing approaches. Liou and Palis address this issue in their paper [21]. They first apply three average-performing merging algorithms to their clustering algorithm and, next, run the three merging algorithms without applying the clustering algorithm and conclude that clustering is an essential step. They build their conclusion based on problem and algorithmic-specific assumptions. We believe that reaching such a conclusion may need a more thorough approach and a specialized framework and set of experiments. Hence, their comparison and conclusions cannot be generalized to our context in this paper. Dikaiakos et al. also propose a framework in [4] that compares various combinations of clustering and merging. In [26], Radulescu et al., to evaluate the performance of their merging algorithm (LLB), use DSC as the base for clustering algorithms and compare the performance of DSC and four merging algorithms (Sarkar's, Yang's, GLB, and LLB) against the one-step MCP algorithm. They show that their algorithm outperforms other merging algorithms used with DSC, while it is not always as efficient as MCP. In their comparison, they do not take the effect of clustering algorithms into account and only emphasize merging algorithms.

In Section 4, we show that the clustering performance does not necessarily provide an accurate answer to the overall performance of the two-step scheduling and, hence, cluster comparison does not provide important information with regard to the scheduling performance. Hence, a more accurate comparison approach should compare the two-step against the one-step scheduling algorithms. In this research, we will give a framework for such comparisons that take the compile-time budget into account as well.

3 The Proposed Mapping Algorithm and Solution Description

3.1 CFA: Clusterization Function Algorithm

In this section, we present a new framework for applying GAs to multiprocessor scheduling problems. For such problems, any valid and legal schedule should satisfy the precedence constraints among the tasks and every task should be present and appear only once in the schedule. Hence, the representation of a schedule for GAs must accommodate these conditions. Most of the proposed GA methods satisfy these conditions by representing the schedule as several lists of ordered task nodes where each list corresponds to the task nodes run on a processor. These representations are typically *sequence*-based [5]. Observing the complexity of these representations and the fact that conventional operators that perform well on bit-string encoded solutions do not work on solutions represented in the forms of sequences opens up the possibility of gaining a high quality solution by designing a well-defined representation. Hence, our solution only encodes the mapping-related information and represents it as a single subset of graph edges β, with no notion of an ordering among the elements of β. This representation can be used with a wide variety of scheduling and clustering problems. Our technique is also the *first* clustering algorithm that is based on the framework of genetic algorithms.

Our representation of clustering exploits the view of a clustering as a subset of edges in the task graph. Gerasoulis and Yang have suggested an analogous view of clustering in their characterization of certain clustering algorithms as being *edge-zeroing* algorithms [6]. In this paper, we apply this subset-based view of clustering to develop an efficient genetic algorithm formulation. For the purpose of a genetic algorithm, the representation of graph clusterings as subsets of edges is attractive since *subsets have natural and efficient mappings into the framework of genetic algorithms*.

Derived from the *schema* theory (a schema denotes a similarity template that represents a subset of $\{0,1\}^l$), canonical GAs (which use binary representations of each solution as fixed-length strings over the set $\{0,1\}$ and efficiently handle optimization problems of the form $f:\{0,1\} \rightarrow \Re$) provide near-optimal sampling strategies over subsequent generations [1]. Furthermore, binary encodings in which the semantic interpretations of different bit positions exhibit high symmetry (e.g., in our case, each bit corresponds to the existence or absence of an edge within a cluster) allow us to leverage extensive prior research on genetic operators for symmetric encodings rather than forcing us to develop specialized, less-thoroughly tested operators to handle the underlying nonsymmetric, nontraditional, and sequence-based representation. Hence, our binary encoding scheme is favored both by schema theory and significant prior work on genetic operators. Furthermore, by providing no constraints on genetic operators, our encoding scheme preserves the natural behavior of GAs. Finally, conventional GAs assume that symbols or bits within an individual representation can be independently modified and rearranged; however, a scheduling solution must contain exactly one instance of each task and the sequence of tasks should not violate the precedence constraints. Thus, any deletion, duplication, or moving of tasks constitutes an error. Traditional crossover and mutation operators are generally capable of producing infeasible or illegal solutions. Under such a scenario, the GA must either discard or repair the nonviable solution. Repair mechanisms transform infeasible individuals into feasible ones. They may not always be successful. Our proposed approach never generates an invalid solution and, thus, saves repair-related compilation time that would otherwise have been wasted in locating, removing, or correcting such solutions. Our encoding of clustering is based on the following definition:

Definition 1. *Suppose that β_i is a subset of task graph edges. Then, $f_{\beta_i}: E \rightarrow \{0,1\}$ denotes the* ***clusterization function*** *associated with β_i. This function is defined by:*

$$f_{\beta_i}(e) = \begin{cases} 0 & \text{if}(e \in \beta_i) \\ 1 & \text{otherwise,} \end{cases} \tag{2}$$

where E is the set of communication edges and e denotes an arbitrary edge of this set. When using a clusterization function to represent a clustering solution, the edge subset β_i is taken to be the set of edges that are contained in one cluster. To form the clusters, we use the information given in β (zero and one edges) and put every pair of task nodes that are joined with zero edges together. The set β is defined as in (3):

$$\beta = \cup_{i=1}^{n} \beta_i. \tag{3}$$

An illustration is shown in Fig. 1. In Fig. 1a, all the edges of the graph are mapped to 1, which implies that the β_i

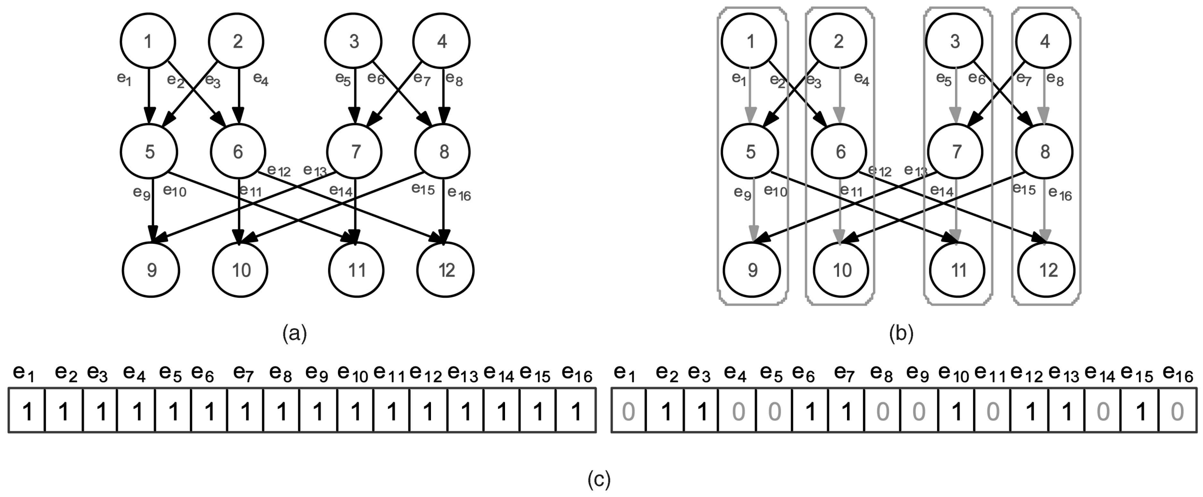
(a)
(b)
(c)
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15 e16
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15 e16
0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 0

Fig. 1. (a) An application graph representation of an FFT and the associated clusterization function f_{β_a}. (b) A clustering of the FFT application graph and f_{β_b}. (c) The resulting subset β_b of clustered edges, along with the empty subset β_a of clustered edges in the original (unclustered) graph.

subsets are empty. In Fig. 1b, edges are mapped to both 0s and 1s and four clusters have been formed. The associated β_i subsets of zero edges are given in Fig. 1c. The time complexity of forming the clusters is $O(|E|)$.

In this paper, the term *clustering* represents a clustered graph, where every pair of nodes in each cluster is connected by a path. A *clustered graph* in general can have tasks with no connections that are clustered together. In this research, however, we do not consider such clusters. We also use the term clustering and clustered graph interchangeably. Because it is based on clusterization functions to represent candidate solutions, we refer to our GA approach as the *clusterization function algorithm* (CFA). The CFA representation offers some useful properties that are described below (proofs of these properties are provided in the Appendix):

Property 1. *Given a clustering, there exists a clusterization function that generates it.*

Property 2. *Given a clusterization function, there is a unique clustering that is generated by it.*

There is also an implicit use of knowledge in CFA-based clustering. In most GA-based scheduling algorithms, the initial population is generated by randomly assigning tasks to different processors. The population evolves through the generations by means of genetic operators and the selection mechanism while the only knowledge about the problem that is taken into account in the algorithm is of a structural nature, through the verification of solution feasibility. In such GAs, the search is accomplished entirely at random considering only a subset of the search space. However, in CFA, the assignment of tasks to clusters or processors is based on the edge zeroing concept. In this context, clustering tasks nodes together is not entirely random. Two task nodes will only be mapped onto one cluster if there is an edge connecting them and they can not be clustered together if there is no edge connecting them because this clustering cannot improve the parallel time. Although GAs do not need any knowledge to guide their search, GAs that do have the advantage of being augmented by some knowledge about the problem they are solving have been shown to produce higher quality solutions and to be capable of searching the design space more thoroughly and efficiently [3]. The implementation details of CFA are provided in [9].

3.2 Randomized Clustering: RDSC, RSIA

Two of the well-known clustering algorithms discussed earlier in this paper, DSC and SIA, are deterministic heuristics, while our GA is a guided random search method where elements in a given set of solutions are probabilistically combined and modified to improve the fitness of populations. To be fair in the comparison of these algorithms, we have implemented a randomized version of each deterministic algorithm—each such randomized algorithm, like the GA, can exploit increases in additional computational resources (compile time tolerance) to explore larger segments of the solution space.

Since the major challenge in clustering algorithms is to find the most strategic edges to "zero" in order to minimize the parallel execution time of the scheduled task graph, we have incorporated randomization into the edge selection process when deriving randomized versions of DSC (RDSC) and SIA (RSIA). In the randomized version of SIA, we first sort all the edges based on the sorting criteria of the algorithm, i.e., the highest IPC cost edge has the highest priority. The first element of the sorted list, that is, the candidate edge to be zeroed (inserted in a cluster), then is selected with probability p where p is a parameter of the randomized algorithm (we call p the *randomization parameter*); if this element is not chosen, the second element is selected with probability p; and so on, until some element is chosen, or no element is returned after considering all the elements in the list. In this last case (no element is chosen), a random number is chosen from a uniform distribution over $\{0, 1, \ldots, |T| - 1\}$ (where T is the set of edges that have not yet been clustered).

In the randomized version of the DSC algorithm, at each clustering step, two node priority lists are maintained: a partial free task list and a free task list, both sorted in descending order of their task priorities (the priority for each task in the free list is the sum of the task's $tlevel$ and $blevel$. The priority value of a partial free task is defined based on the $tlevel$, IPC, and computational cost [31]). The criterion for accepting a zeroing is that the value of $tlevel(v_x)$ of the highest priority free list does not increase by such zeroing. Similar to RSIA, we first sort based on the sorting criteria of the algorithm, the first element of each sorted list then is selected with probability p, and so on. Further details on this general approach to incorporating randomization into greedy, priority-based algorithms can be found in [34].

When $p = 0$, clustering is always randomly performed by sampling a uniform distribution over the current set of edges, and when $p = 1$, the randomized technique reduces to the corresponding deterministic algorithm. Each randomized algorithm version begins by first applying the underlying (original) deterministic algorithm, and then repeatedly computing additional solutions with a "degree of randomness" determined by p. The best solution computed within the allotted (prespecified) compile-time tolerance is returned. Our randomized algorithms, by way of running the corresponding deterministic algorithms first, maintain the performance bounds of the deterministic algorithms. A careful analysis of the (potentially better) performance bounds of the randomized algorithms is an interesting direction for the future study. Experimentally, we have found the best randomization parameters for RSIA and RDSC to be 0.10 and 0.65, respectively.

3.3 Merging

Merging is the final phase of scheduling and is the process of mapping a set of clusters (as opposed to task nodes) to the parallel embedded multiprocessor system where a finite number of processors is available. This process should also maintain the minimum achievable parallel time while satisfying the resource constraints and must be designed to be as efficient as scheduling algorithms. As mentioned earlier for the merging algorithm, we have modified the ready-list scheduling heuristic so it can be applied to a cluster of nodes (CRLA). This algorithm is indeed very similar to the Sarkar's task assignment algorithm except for the priority metric: Studying the existing merging techniques, we observed that if the scheduling strategy used in the merging phase is not as efficient as the one used in the clustering phase, the superiority of the clustering algorithm can be negatively effected. To solve this problem, we implemented a merging algorithm (clustered ready-list scheduling algorithm or CRLA) such that it can use the timing information produced by the clustering phase. We observed that if we form the priority list in order of increasing (LST, $TOPOLOGICAL\_SORT\_ORDERING$) of tasks (or $blevel$), tasks preserve their relative ordering that was computed in the clustering step. $LST(v_i)$ or the latest starting time of task v_i is defined as $LST(v_i) = LCT(v_i) - t(v_i)$. $LCT(v_i)$ or the latest completion time is the latest time at which task v_i can complete execution. Similar to Sarkar's task assignment algorithm, the same ordering is also maintained when tasks are sorted within clusters.

In CRLA, initially there are no tasks assigned to the n_P available processors. The algorithm starts with the clustered graph and maps it to the processor thorough $|V|$ iterations. In each stage, a task at the head of the priority list is selected and, along with other tasks in the same cluster, is assigned to one of the n_P processors that gives the minimum parallel time increase from the previous iteration. For cluster to processor assignment, we always assume all the processors are idle or available. The algorithm finishes when the number of clusters has been reduced to the actual number of physical processors. In the following section, we explain the implementation of the overall system.

3.4 Two-Phase Mapping

In order to implement the two-step scheduling techniques described earlier, we used the three addressed clustering algorithms: CFA, RDSC, and RSIA in conjunction with CRLA. Our experiments were set up in two different formats that are described in Sections 3.4.1 and 3.4.2.

3.4.1 First Approach

In the first step, the clustering algorithms, being characterized by their probabilistic search of the solution space, had to run iteratively for a given time budget. Through extensive experimentation with CFA using small and large size graphs, we found that running CFA for 3,000 iterations is the best setup for CFA. We then ran CFA for this number of iterations and recorded the running time of the algorithm as well as the resulting clustering and performance measures. We used the recorded running time of CFA for each input graph to determine the allotted running time for RDSC or RSIA on the same graph. This technique allows comparison under equal amounts of running time. After we found the results of each algorithm within the specified time budget, we used the clustering information as an input to the merging algorithm described in Section 3.3 and ran it once to find the final mapping to the actual target architecture. In most cases, the number of clusters in CFA's final result is more than the number in RSIA or RDSC. RSIA tends to find solutions with smaller numbers of clusters than the other two algorithms. To compare the performance of these algorithms, we set the number of actual processors to be less than the minimum achieved number of clusters. Throughout the experiments, we tested our algorithms for two, four, and eight processor architectures depending on the graph sizes.

3.4.2 Second Approach

Although CRLA employs the timing information provided in the clustering step, the overall performance is still sensitive to the employed scheduling or task ordering scheme in the clustering step. To overcome this deficiency, we modified the fitness function of CFA to be the merging algorithm. Hence, instead of evaluating each cluster based on its local effect (which would be the parallel time of the clustered graph mapped to an infinite processor architecture), we evaluate each cluster based on its effect on the final mapping. Except for this modification, the rest of the implementation details for CFA remain unchanged. RDSC and RSIA are not modified although the experimental setup is changed for them. Consequently, instead of running these two algorithms for as long as the time budget allows, locating the best clustering, and applying merging in one step, we run the overall two-step algorithm within the time

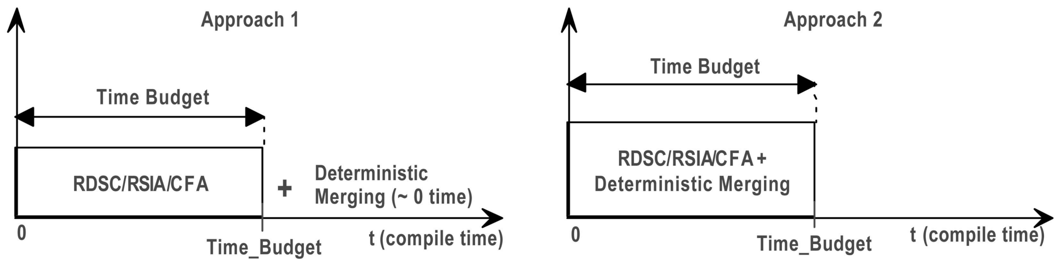

Fig. 2. The diagrammatic difference between the two different implementations of the two-step clustering and cluster-scheduling or merging techniques. Both find the solution at the given time budget.

budget. That is, we run RDSC (RSIA) once, apply the merging algorithm to the resulting clustering, store the results, and start over. At the end of each iteration, we compare the new result with the stored result and update the stored result if the new one shows a better performance. The difference between these two approaches is shown in Fig. 2. Experimental results for this approach are given in Section 4.

For the second proposed approach, the fitness evaluation may become time-consuming as the graph size increases. Fortunately, however, there is a large amount of parallelism in the overall fitness evaluation process. Therefore, for better scalability and faster runtime, one could develop a parallel model of the second framework. One such model (micrograin parallelism [15]) is the asynchronous master-slave parallelization model [7]. This model maintains a single local population while the evaluation of the individuals is performed in parallel. This approach requires only knowledge of the individual being evaluated (not the whole population), so the overheard is greatly reduced. Other parallelization techniques such as course-grained and fine-grained [15] can also be applied for performance improvements to both approaches, while the micrograin approach would be most beneficial for the second approach, which has a costly fitness function.

3.5 Comparison Method

In this section, we present a comparison framework that will help us answer some unanswered questions regarding the performance and effectiveness of multistep scheduling algorithms. We will determine 1) if a preprocessing step (clustering here) is advantageous to the multiprocessor scheduling, 2) the effect of each step (clustering and cluster scheduling) on the overall performance, and 3) the performance measures for each step.

To compare the performance of a two-step decomposition scheme against a one-step approach, since our algorithms are probabilistic (and time-tolerant) search algorithms, we need to compare them against a one-step scheduling algorithm with similar characteristics, i.e., capable of exploiting the increased compile time. Consequently, we first selected a one-step evolutionary based scheduling algorithm, called *combined genetic-list algorithm* (CGL) [3], that was shown to have outperformed the existing one-step evolutionary based scheduling algorithms. Next, we selected a well-known and efficient list scheduling algorithm (that could also be efficiently modified to be employed as a cluster-scheduling algorithm). The algorithm we selected is an important generalization of list-scheduling, which is called ready-list scheduling and has been formalized by Printz [25]. Ready-list scheduling maintains the list-scheduling convention that a schedule is constructed by repeatedly selecting and scheduling ready nodes, but eliminates the notion of a static priority list and a global time clock. In our implementation, we used the $blevel(v_x)$ metric to assign node priorities. We also used the insertion technique to further improve the scheduling performance. With the same technique described in Section 3.2, we also applied randomization to the process of constructing the priority list of nodes and implemented a *randomized ready-list* scheduling (RRL) technique that can exploit increases in additional computational resources. We then set up an experimental framework for comparing the performance of the two-step CFA (the best of the three clustering algorithms CFA, RDSC, and RSIA [10]) and CRLA against one-step CGL and RRL algorithms. We also compared DSC and CRLA against the RL algorithm (Step 3 in Fig. 3).

In the second part of these experiments, we study the effect of each step in overall scheduling performance. To find out if an efficient merging can make up for an average performing clustering, we applied CRLA to several clustering heuristics: First, we compared the performance of the two well-known clustering algorithms (DSC and SIA) against the randomized versions of these algorithms (RDSC and RSIA) with CRLA as the merging algorithm. Next, we compared the performance of CFA and CRLA against RDSC and RSIA. By keeping the merging algorithm unchanged in these sets of experiments, we are able to study the effect of a good merging algorithm when employed with clustering techniques that exhibit a range of performance levels.

To find out the effect of a good clustering while combined with an average-performing merging algorithm, we modified CRLA to use different metrics such as topological ordering and static level to prioritize the tasks and compared the performance of CFA and CRLA against CFA and the modified-CRLA. We repeated this comparison for RDSC and RSIA. In each set of these experiments, we kept the clustering algorithm fixed so we can study the effect of a good clustering when used with different merging algorithms. The outline of this experimental set up is presented in Fig. 3.

Step1. Select a well-known efficient single-phase scheduling algorithm.
(insertion-based Ready-List Scheduling (RL) with blevel metric)

Step2. Modify the scheduling algorithm to get
a) An algorithm that accepts clusters of nodes as input (Clustered Ready-List Scheduling (CRLA)),
b) An algorithm that can exploit the increased compile time.(Randomized Ready-List Scheduling (RRL))

Step 3. Compare the performance of a one-phase scheduling algorithm vs. a two phase scheduling algorithm

a) CFA + CRLA vs. RRL b) CFA + CRLA vs. CGL c) DSC + CRLA vs RL

Step 4. Compare the importance of clustering phase vs. merging phase

a) CFA + CRLA vs. RDSC + CRLA b) CFA + CRLA vs. RSIA + CRLA
c) DSC + CRLA vs. RDSC + CRLA d) SIA + CRLA vs. RSIA + CRLA
e) CFA + CRLA vs. CFA + CRLA (using different metrics)
f) RDSC + CRLA vs. RDSC + CRLA (using different metrics)
g) RSIA + CRLA vs. RSIA + CRLA (using different metrics)

Fig. 3. Experimental setup for comparing the effectiveness of a one-step versus the two-step scheduling method.

4 Performance Evaluation and Comparison

In this section, we present the performance results and comparisons of clustering and merging algorithms described in Section 3. All algorithms were implemented on an Intel Pentium III processor with a 1.1 GHz CPU speed. All the heuristics have been tested with three sets of input graphs that use similar structures and sizes as used in the literature. These three sets consist of 1) *Reference Graphs (RG)* that are task graphs that have been previously used by different researchers and addressed in the literature, 2) *Application Graphs (AG)* that involve numerical computations (number of tasks varies from 10 to 2,000 tasks) and digital signal processing (DSP), and 3) *Random Graphs (RANG)* that are generated using Sih's random benchmark graph generator [28]. Sih's generator attempts to construct synthetic benchmarks that are similar in structure to task graphs of real applications. Due to limited space, in this paper, we only present the detailed results of the FFT application from the AG set and a subset of RANG set graphs. More details on test graph sets and complete results are provided in [9].

To make a more accurate comparison, we have used the Normalized Parallel Time (NPT) that is defined as:

$$NPT = \tau_P / \left(\sum_{v_i \in CP} t(v_i) \right), \tag{4}$$

where τ_P is the parallel time. The sum of the execution times on the Critical Path (CP) represents a lower bound on the parallel time. Running times of the algorithms are not useful measures in our case because we run all the algorithms under an equal time-budget.

4.1 Results for the Application Graphs (AG) Set

The results of the performance comparisons of one-step scheduling algorithms versus two-step scheduling algorithms for a subset of the AG set (FFT set) are given in Fig. 4. The number of nodes varies from 100 to 2,500 nodes depending on the matrix size N. The CCR values represent the *communication to computation ratio* that is the average communication cost to the average computation cost. The first six graphs show the performance of the CFA and CRLA against RRL and CGL algorithm for two, four, and eight processor architectures.

A quantitative comparison of these algorithms is given in Table 1 and Table 2. The experimental results of studying the effect of clustering on the AG set are given in Fig. 5 and Fig. 6. We observed that CFA performs its best in the presence of heavy interprocessor communication cost (e.g., CCR = 10). In such situations, exploiting parallelism in the graph is particularly difficult, and most other algorithms perform relatively inefficiently and tend to greedily cluster edges to avoid IPC (over 97 percent of the time, CFA outperformed other algorithms under high communication costs). The trend in multiprocessor technology is toward increasing costs of interprocessor communication relative to processing costs (task execution times) [2], and we see that CFA is particularly well suited toward handling this trend (when used prior to the scheduling process).

4.2 Results for the Random Graphs (RANG) Set

In this section, we have shown the experimental results (in terms of average NPT or ANPT) for setI of the RANG task graphs. Fig. 7 shows the results of comparing RRL and CGL against the two step CFA and CRLA and RL scheduling against the two-step DSC algorithm and CRLA. The experimental results of studying the effect of clustering are given in Fig. 8 and Fig. 9. In general, it can be seen that, as the number of processors increases, the difference between the algorithms performance becomes more apparent. This is because, when the number of processors is small, the merging algorithm has limited choices for the mapping of clusters and, hence, most tasks end up running on the same processor regardless of their initial clustering.

A quantitative comparison of these scheduling algorithms is also given in Table 1 and Table 2. It can be seen that, given two equally good one-step and two-step scheduling algorithms, the two-step algorithm gains better performance compared to the single-step algorithm. DSC is a relatively good clustering algorithm but is not as efficient as CFA or RDSC. However, it can be observed that, when used against a one-step scheduling algorithm, it still can

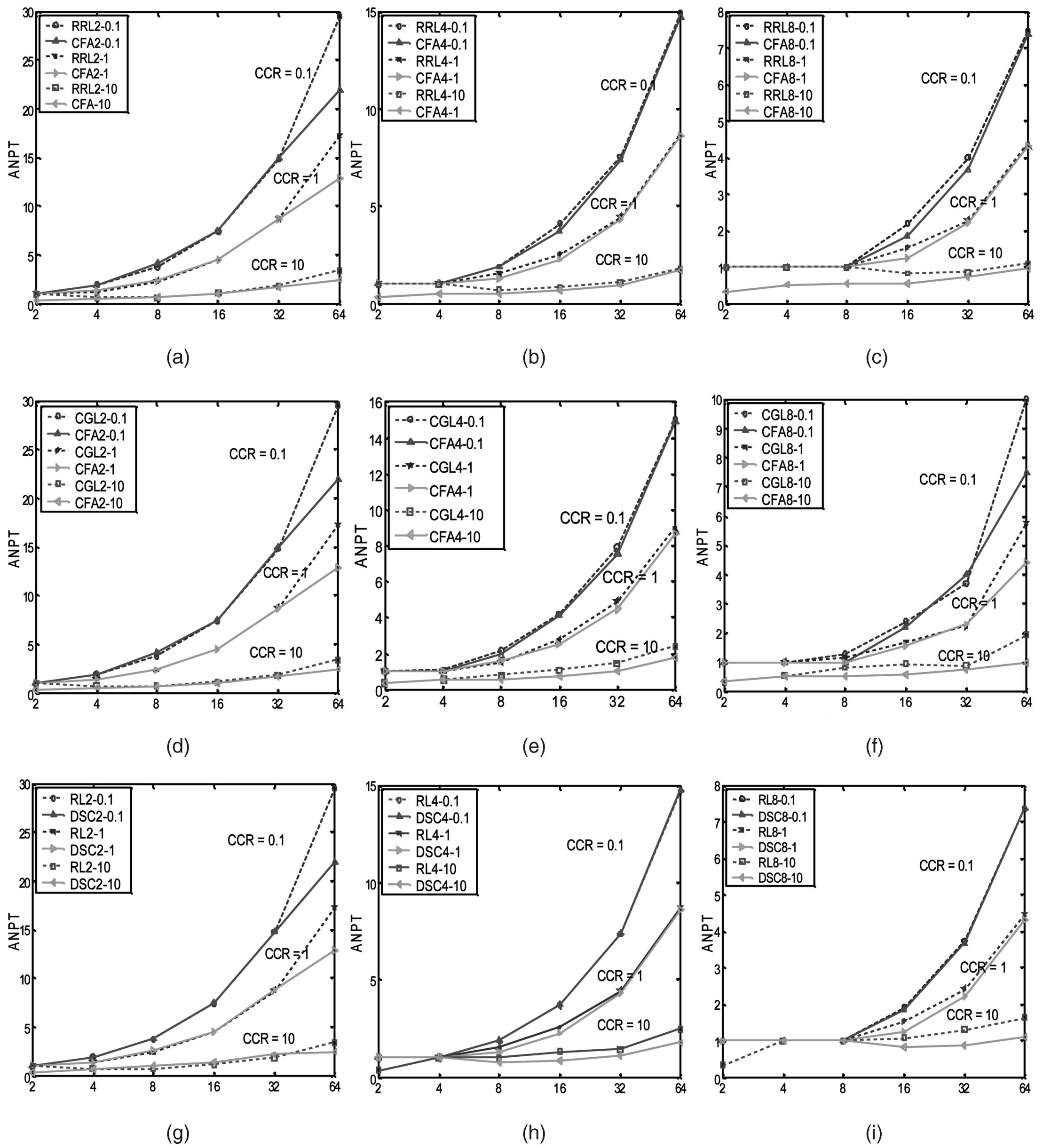

Fig. 4. Effect of one-phase versus two-phase scheduling for a subset of AG set. RRL versus CFA + CRLA on (a) 2-processor, (b) 4-processor, and (c) 8-processor architectures. CGL versus CFA + CRLA on (d) 2-processor, (e) 4-processor, and (f) 8-processor architectures. RL versus DSC + CRLA on (g) 2-processor, (h) 4-processor, and (i) 8-processor architectures.

offer better solutions (up to 14 percent improvement). It can also be observed that, the better the quality of the clustering algorithms, the better the overall performance of the scheduling algorithms. In this case, CFA clustering is better than RDSC and RSIA and RDSC are RSIA and better than their deterministic versions.

We have not presented the results of applying different metrics graphically; however, a summary of the results is as follows: For both test graph sets when tested with different merging algorithms (we used CRLA with three different priority metrics: topological sort ordering, static level, and a randomly sorted priority list), each clustering algorithm did best with the original CRLA (using *blevel* metric), moderately worse with static level, and worst with random level. As shown in the literature, the performance of the list scheduling algorithm highly depends on the priority metrics used and we observed that this was also the case for the original CRLA. Employing the information provided in clustering in the original CRLA was also another strength for the algorithm. We also implemented an evolutionary-based merging algorithm, however, we did not get significant improvement in the results. We conclude that, as long as the merging algorithm utilizes the clustering information and does not restrict the processor selection to the idle processors at the time of assignment (local decision or greedy choice), it can efficiently schedule the clusters without further need for complex assignment schemes or evolutionary algorithms.

We also observed that, in several cases where the results of clustering (parallel time) were equal, CFA could outperform RDSC and RSIA after merging (this trend was not observed for RDSC versus DSC and RSIA versus SIA). We

TABLE 1
Performance Comparison of CFA, RRL, CGL, RDSC, and RSIA

Algo.	RRL(%)				CGL(%)				RDSC(%)				RSIA(%)			
CFA+ CRLA	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.	>	=	<	% imp.
RG	78	15	7	6.0	84.5	11.5	4	17.6	84	12	4	5.0	84	16	0	8.0
AG	46	27	27	10.5	64	28	8	11.1	44	56	0	11.4	50	50	0	3.0
RANG	82	10	8	9.0	100	0	0	18.0	82.3	12.7	5	5.0	92.7	7.3	0	6.0
Avg.	***69***	***17***	***14***	***8.5***	***83***	***13***	***4***	***15.6***	***70***	***27***	***3***	***7.1***	***75.6***	***24.4***	***0***	***5.7***

TABLE 2
Performance Comparison of DSC and RL

Algo.	RL(%) - RG set				RL(%) - AG set				RL(%) - RANG set				*Avg.*
-	>	=	<	%imp.	>	=	<	%imp.	>	=	<	%imp.	***%imp.***
DSC+CRLA	70.9	11	18.1	10.0	47	36	17	14.0	10	80	10	14.0	***9.3***

also noted that there are occasional cases that two clustering results with different parallel times provide similar answers in the final mapping. There are also cases where a worse clustering algorithm (worse parallel time) finds better final results.

To find the reason for the first behavior, we studied the clustering results of each algorithm separately. CFA tends to use the most number of clusters when clustering tasks: There are several cases where two clusters could be merged with no effect on the parallel time. CFA keeps them as separate clusters. However, both RSIA and RDSC accept such clustering, i.e., when the result of clustering doesn't change the parallel time, and they tend to cluster as much as possible in the clustering step. Providing more clusters and clustering only those tasks with high data dependency gives more flexibility to the merging algorithm for mapping the results of CFA. This characteristic of CFA is the main reason that, even in the case of similar parallel time for clustering results, CFA is still capable of getting better overall performance.

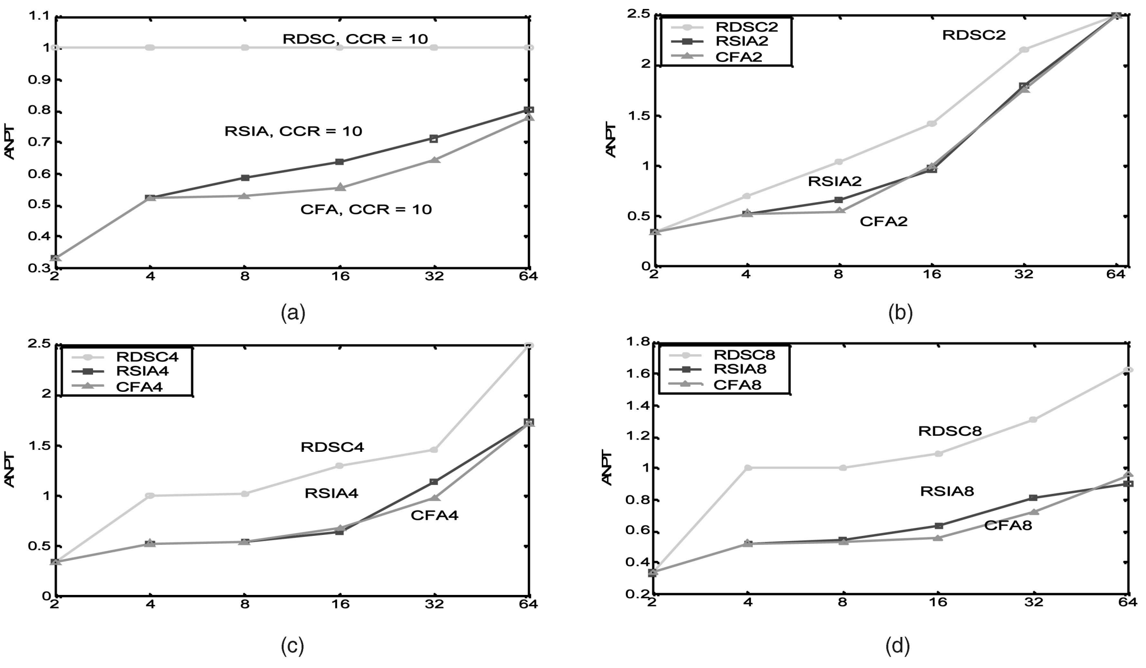

Fig. 5. Average normalized parallel time from applying RDSC, RSIA, and CFA to a subset of AG set (for CCR = 10), (a) results of clustering algorithms. Results of mapping the clustered graphs onto (b) a 2-processor, (c) a 4-processor, and (d) an 8-processor architecture.

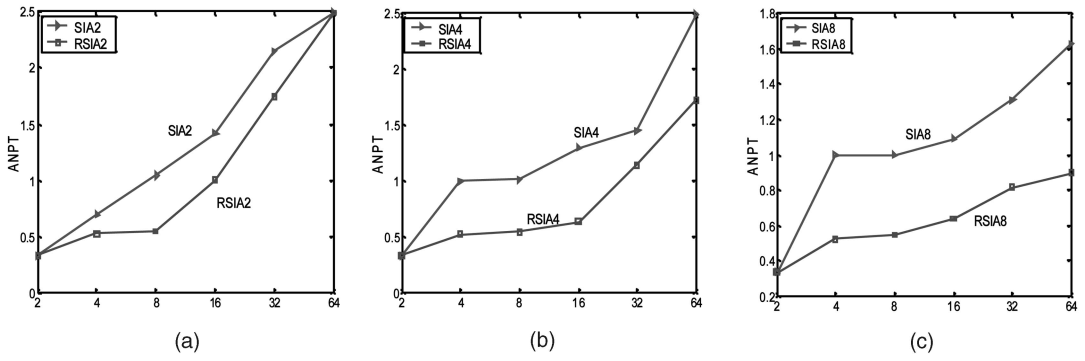

Fig. 6. Effect of clustering: Performance comparison of SIA and RSIA on a subset of AG graphs mapped to (a) 2-processor, (b) 4-processor, and (c) 8-processor architectures using the CRLA algorithm.

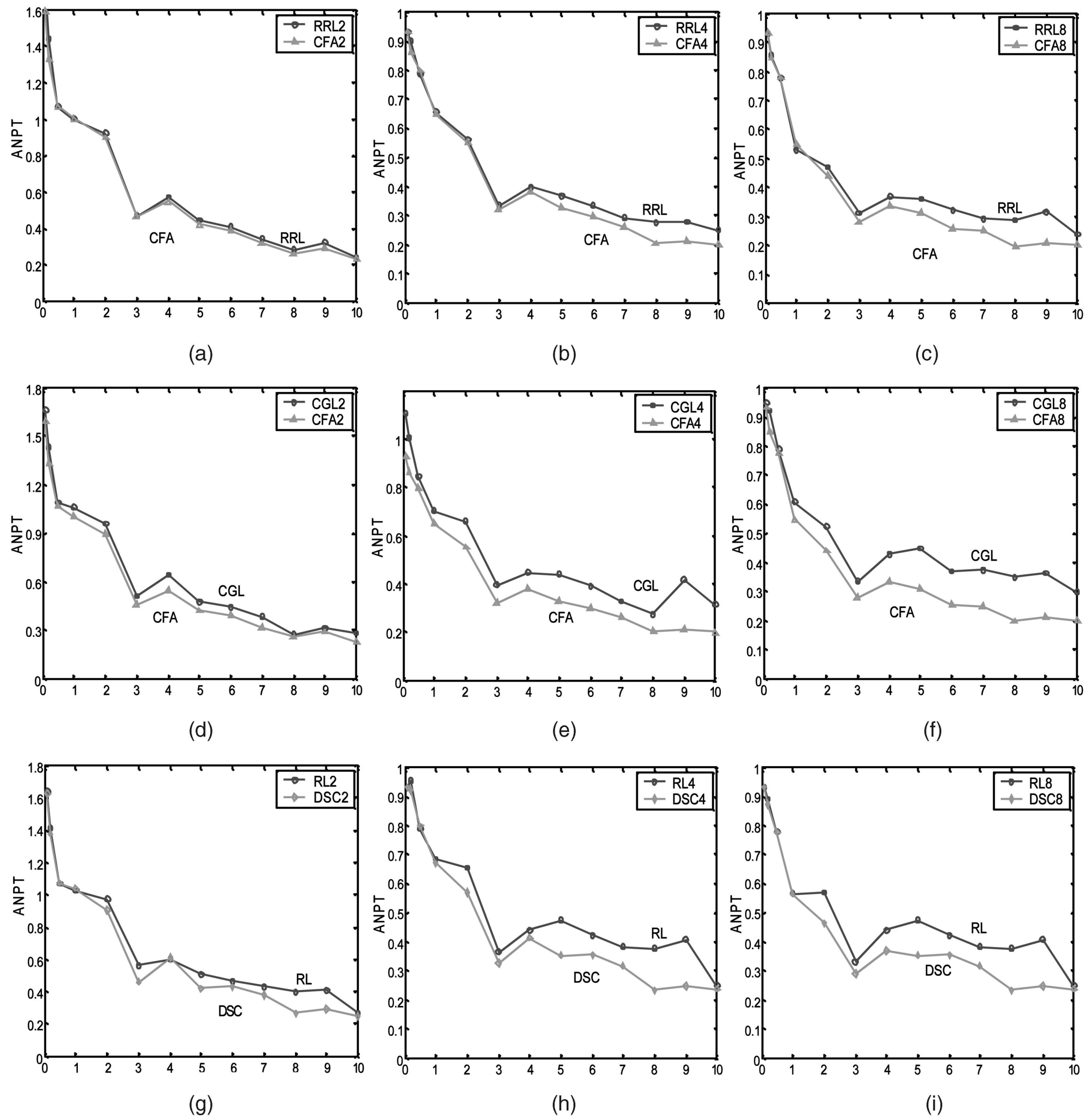

Fig. 7. Effect of one-phase versus two-phase scheduling for the RANG setI. RRL versus CFA + CRLA on (a) 2-processor, (b) 4-processor, and (c) 8-processor architectures. CGL versus CFA + CRLA on (d) 2-processor, (e) 4-processor, and (f) 8-processor architectures. RL versus DSC + CRLA on (g) 2-processor, (h) 4-processor, and (i) 8-processor architectures.

For the second behavior, we believe that the reason is behind the scheduling scheme (or task ordering) used in the clustering step. CFA uses an insertion based task scheduling and ordering, which is not the case for the other clustering algorithms. Hence, there are cases where similar clusterings of tasks end up providing different parallel times. This behavior was only observed for two cases. For a worse algorithm performing better at the end (only

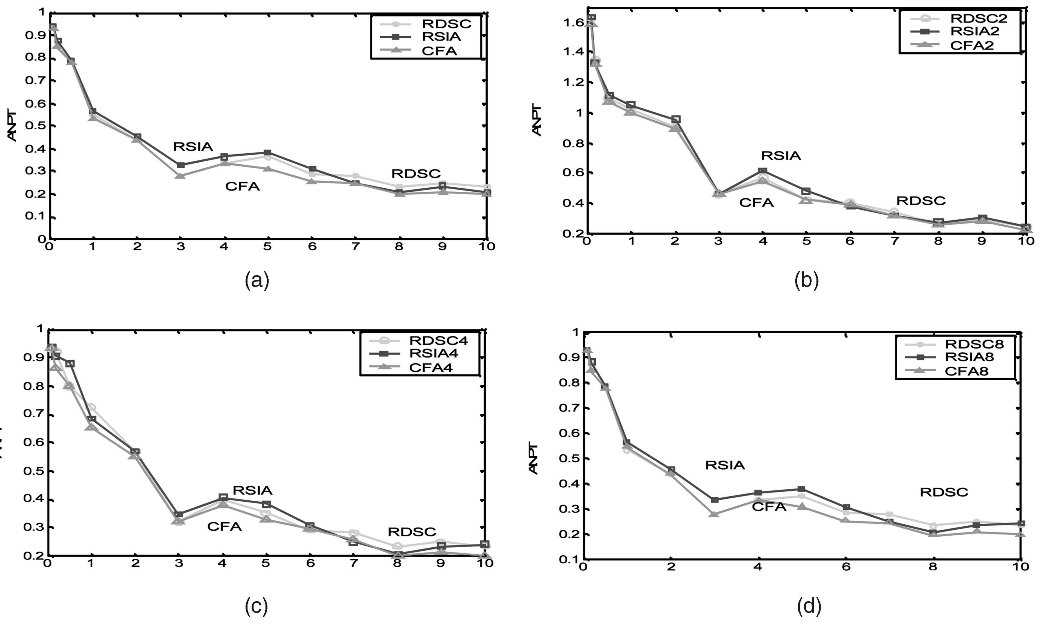

Fig. 8. Average normalized parallel time from applying RDSC, RSIA, and CFA to RANG setI, (a) results of clustering algorithms. Results of mapping the clustered graphs onto (b) a 2-processor, (c) a 4-processor, and (d) an 8-processor architecture.

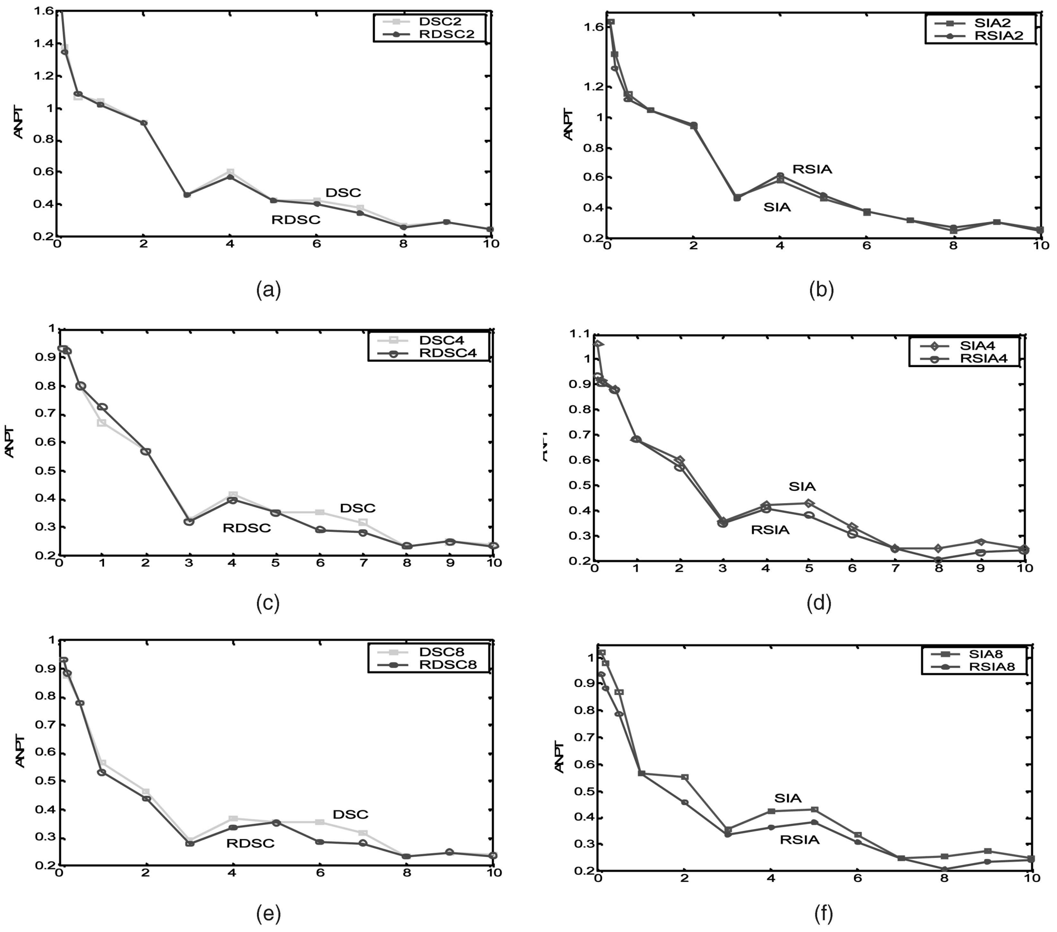

Fig. 9. Effect of clustering: Performance comparison of DSC, RDSC, SIA, and RSIA on RANG setI graphs mapped to (a), (d) 2-processor, (b), (e) 4-processor, and (c), (f) 8-processor architecture using the CRLA algorithm.

observed in the case of RSIA and SIA), the explanation is similar to that for the first behavior. A clustering algorithm should be designed to adjust the communication and computation time by changing the granularity of the program. Hence, when a clustering algorithm ignores this fact and groups tasks together as much as possible, many tasks with little data dependencies end up together, and while this approach may give a better parallel time for clustering, it will fail in the merging step due to its decreased flexibility.

Observing these behaviors, we believe that the performance of clustering algorithms should only be evaluated in conjunction with the cluster-scheduling step as the clustering results do not determine the final performance accurately.

5 Summary and Conclusions

In this paper, we presented an experimental setup for comparing one-step scheduling algorithms against two-step scheduling (clustering and cluster-scheduling) algorithms. We have taken advantage of the increased compile-time tolerance of embedded systems and have employed more thorough algorithms for this experimental setup. We have developed a novel and natural genetic algorithm formulation, called CFA, for multiprocessor clustering, as well as randomized versions, called RDSC and RSIA, of two well-known deterministic algorithms, DSC and SIA, respectively. The experimental results suggest that a preprocessing step that minimizes communication overhead can be very advantageous to multiprocessor scheduling and two-step algorithms provide better quality schedules. We also studied the effect of each step of the two-step scheduling algorithm in the overall performance and learned that the quality of clusters does have a significant effect on the overall mapping performance. We also showed that the performance of a poor-performing clustering algorithm cannot be improved with an efficient merging algorithm. A clustering is not efficient when it either combines tasks inappropriately or puts tasks that should be clustered together in different clusters. In the former case, merging cannot help much because merging does not change the initial clustering. In the latter case, merging can sometimes help by combining the associated clusters on the same processor. However, in this case, the results may not be as efficient as when the right tasks are mapped together initially. Hence, we conclude that the overall performance is directly dependent on the clustering step and this step should be as efficient as possible.

The merging step is important as well and should be implemented carefully to utilize information provided in clustering. A modified version of ready-list scheduling was shown to perform very well on the set of input clusters. We observed that, in several cases, the final performance is different than the performance of the clustering step (e.g., a worse clustering algorithm provided a better merging answer). This suggests that the clustering algorithm should be evaluated in conjunction with a merging algorithm as their performance may not determine the performance of the final answer. One better approach to compare the performance of the clustering algorithms may be to look at the number of clusters produced or cluster utilization in conjunction with parallel time. In most cases, the clustering algorithm with a smaller parallel time and more clusters resulted in better results in merging as well. A good clustering algorithm only clusters tasks with heavy data dependencies together and maps many end nodes (sinks) or tasks off the critical paths onto separate clusters giving the merging algorithms more flexibility to place the not-so-critically located tasks onto physical processors. We are currently working to generalize the merging step to be used for heterogeneous processors and interconnection constrained networks.

Appendix

Property 1. *Given a clustering, there exists a clusterization function that generates it.*

Proof. Our proof is derived from the function definition in (2). Given a clustering of a graph, we can construct the clusterization function f_β by examining the edge list. Starting from the head of the list, for each edge (or ordered pair of task nodes), if both head and tail of the edge belong to the same cluster $(\forall e_k | e_k = (v_i, v_j)((v_i \in c_x) \wedge (v_j \in c_x)))$, then the associated edge cost would be zero and, according to (2), $f(e_k) = 0$ (this edge also belongs to β_x, i.e., $e_k \in \beta_x$). If the head and tail of the edge do not belong to the same cluster $((((v_i \in c_x) \wedge (v_j \notin c_x)) \vee ((v_i \notin c_x) \wedge (v_j \in c_x))))$, then $f(e_k) = 1$. Hence, by examining the edge list, we can construct the clusterization function and this concludes the proof. □

Property 2. *Given a clusterization function, there is a unique clustering that is generated by it.*

Proof. The given clusterization function, f_β can be represented in the form of a binary array with the length equal to $|E|$, where the $i\text{th}$ element of array is associated with the $i\text{th}$ edge e_i and the binary values determine whether the edge belongs to a cluster or not. By constructing the clusters from this array, we can prove the uniqueness of the clustering. We examine each element of the binary array and remove the associated edge in the graph if the binary value is 1. Once we have examined all the edges and removed the proper edges, the graph is partitioned to connected components where each connected component is a cluster of tasks. Each edge is either removed or exists in the final partitioned graph depending on its associated binary value. Hence, anytime we build the clustering or clustered graph using the same clusterization function, we will get the same connected components, partitions, or clusters, and, consequently, the clustering formed by a clusterization function is unique. □

Acknowledgments

This research was supported in part by the DARPA funded Optoelectronics Center for innovative Photonic Chipscale Technologies contract number MDA972-00-1-0023, through Brown University.

References

[1] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary Computation: Comments on the History and Current State," *IEEE Trans. Evolutionary Computation*, vol. 1, pp. 3-17, 1997.

[2] L. Benini and G. De Micheli, "Powering Networks on Chip," *Proc. Int'l System Synthesis Symp.*, Oct. 2001.

[3] R.C. Correa, A. Ferreira, and P. Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, pp. 825-837, 1999.

[4] M.D. Dikaiakos, A. Rogers, and K. Steiglitz, "A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors," *Proc. Sixth IEEE Symp. Parallel and Distributed Processing,* 1994.

[5] B.R. Fox and M.B. McMahon, "Genetic Operators for Sequencing Problems," *Foundations of Genetic Algorithms,* 1991.

[6] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Graphs on Multiprocessors," *J. Parallel and Distributed Computing,* vol. 16, pp. 276-291, 1992.

[7] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[8] E.S.H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, pp. 113-120, 1994.

[9] V. Kianzad and S.S. Bhattacharyya, "A Comparison of Clustering and Scheduling Techniques for Embedded Multiprocessor Systems," Technical Report UMIACS-TR-2003-114, Inst. for Advanced Computer Studies, Univ. of Maryland at College Park, Dec. 2003.

[10] V. Kianzad and S.S. Bhattacharyya, "Multiprocessor Clustering for Embedded Systems," *Proc. European Conf. Parallel Computing,* pp. 697-701, Aug. 2001.

[11] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing,* 1988.

[12] N. Koziris, M. Romesis, P. Tsanakas, and G. Papakonstantinou, "An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures," *Proc. Eighth Euromicro Workshop Parallel and Distributed Processing (PDP '00),* pp. 406-413, 2000.

[13] Y. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel and Distributed Computing,* vol. 59, no. 3, pp. 381-422, Dec. 1999.

[14] Y. 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, pp. 506-521, 1996.

[15] Y. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm," *J. Parallel and Distributed Computing,* 1997.

[16] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys,* vol. 31, no. 4, pp. 406-471, Dec. 1999.

[17] R. Lepère and D. Trystram, "A New Clustering Algorithm for Scheduling Task Graphs with Large Communication Delays," *Proc. Int'l Parallel and Distributed Processing Symp.,* 2002.

[18] T. Lewis and H. El-Rewini, "Parallax: A Tool for Parallel Program Scheduling," *IEEE Parallel and Distributed Technology,* vol. 1, no. 2, pp. 62-72, May 1993.

[19] G. Liao, G.R. Gao, E.R. Altman, and V.K. Agarwal, "A Comparative Study of DSP Multiprocessor List Scheduling Heuristics," *Proc. Hawaii Int'l Conf. System Sciences,* 1994.

[20] P. Lieverse, E.F. Deprettere, A.C.J. Kienhuis, and E.A. De Kock, "A Clustering Approach to Explore Grain-Sizes in the Definition of Processing Elements in Dataflow Architectures," *J. VLSI Signal Processing,* vol. 22, pp. 9-20, Aug. 1999.

[21] J.-C. Liou and M.A. Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," *Proc. 11th Int'l Parallel Processing Symp. (IPPS),* pp. 152-156, Apr. 1997.

[22] J.N. Morse, "Reducing the Size of the Nondominated Set: Pruning by Clustering," *Computers and Operations Research,* vol. 7, nos. 1-2, pp. 55-66, 1980.

[23] P. Marwedel and G. Goossens, *Code Generation for Embedded Processors.* Kluwer Academic, 1995.

[24] C.L. McCreary, A.A. Khan, J.J. Thompson, and M.E. McArdle, "A Comparison of Heuristics for Scheduling DAGS on Multiprocessors," *Proc. Int'l Parallel Processing Symp.,* pp. 446-451, 1994.

[25] H. Printz, "Automatic Mapping of Large Signal Processing Systems to a Parallel Machine," PhD thesis, School of Computer Science, Carnegie Mellon Univ., May 1991.

[26] A. Radulescu, A.J.C. van Gemund, and H.-X. Lin, "LLB: A Fast and Effective Scheduling Algorithm for Distributed-Memory Systems," *Proc. Int'l Parallel Processing and Symp. Parallel and Distributed Processing,* pp. 525-530, 1999.

[27] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[28] G.C. Sih, "Multiprocessor Scheduling to Account for Interprocessor Communication," PhD dissertation, ERL, Univ. of California, Berkeley, Apr. 1991.

[29] T. Yang, "Scheduling and Code Generation for Parallel Architectures," PhD thesis, Dept. of Computer Science, Rutgers Univ., May 1993.

[30] T. Yang and A. Gerasoulis, "PYRROS: States Scheduling and Code Generation for Message Passing Multiprocessors," *Proc. Sixth ACM Int'l Conf. Supercomputing,* 1992.

[31] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, pp. 951-967, 1994.

[32] P. Wang and W. Korfhage, "Process Scheduling Using Genetic Algorithms," *IEEE Symp. Parallel and Distributed Processing,* pp. 638-641, 1995.

[33] M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 3, pp. 330-343, July 1990.

[34] E. Zitzler, J. Teich, and S.S. Bhattacharyya, "Optimized Software Synthesis for DSP Using Randomization Techniques," technical report, Computer Eng. and Comm. Networks Laboratory, Swiss Federal Inst. of Technology, Zurich July 1999.

[35] A.Y. Zomaya, C. Ward, and B. Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, pp. 795-812, 1999.

Vida Kianzad received the BS degree in electrical engineering (with honors) from the University of Tehran and the PhD degree in computer engineering from the Department of Electrical and Computer Engineering at the University of Maryland, College Park, in 2006. She is currently a research fellow at Harvard Medical School. Her research interests include VLSI signal processing, optical imaging, CAD for embedded system and nano-tech, and hardware/software codesign. She is a student 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 a 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 professor in the Department of Computer Science. Dr. Bhattacharyya is coauthor or coeditor of four books and the author or coauthor of more than 100 refereed technical articles. His research interests include VLSI signal processing, embedded software, and hardware/software codesign. Dr. Bhattacharyya has held industrial positions as a researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and as a compiler developer at Kuck & Associates (Champaign, Illinois). He is a fellow of the IEEE and IEEE Computer Society.