Technical Report UMIACS-TR-2001-52, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2001.

Multiprocessor Clustering for Embedded System Implementation[1]

Vida Kianzad and Shuvra S. Bhattacharyya

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

June 2001

Abstract. In this paper, we address two key trends in the synthesis of implementations for embedded multiprocessors — (1) the increasing importance of managing interprocessor communication (IPC) in an efficient manner, and (2) the acceptance of significantly longer compilation time by embedded system designers. The former aspect is especially evident in the increasing interest among embedded system architects in innovative communication architectures, such as those involving optical interconnection technologies, and hybrid electro-optical structures [8, 19]. The latter aspect — increased compile-time tolerance — results because embedded multiprocessor systems are typically designed as final implementations for dedicated functions. While multiprocessor mapping strategies for general-purpose systems are usually designed with low to moderate complexity as a constraint, embedded system design tools are allowed to employ more thorough and time-consuming optimization techniques [13].

1. Introduction

In this paper, we develop novel partitioning and scheduling techniques that aggressively streamline interprocessor communication. In particular, we address two important trends in the synthesis of implementations for embedded multiprocessors — (1) the increasing importance of managing interprocessor communication (IPC) in an efficient manner, and (2) the acceptance of significantly longer compilation time by embedded system designers compared to designers of general purpose systems. The former aspect is especially relevant due to the increasing interest among embedded system architects in innovative communication architectures, such as those involving optical interconnection technologies, and hybrid electro-optical structures [8, 19]. Effective experimentation with unconventional architectures requires adequate design tools that can exploit such architectures. The latter aspect — increased compile time tolerance — results because embedded multiprocessor systems are typically designed as final implementations for *dedicated* functions; modifications to embedded system implementations are rare. This increased compile time tolerance allows embedded system design tools to employ more thorough, time-consuming optimization techniques [13]; in contrast, multiprocessor mapping strategies for general purpose systems are typically designed with low to moderate complexity as a constraint.

Our work builds on the two-phased decomposition of multiprocessor scheduling that was introduced by Sarkar [17], and explored subsequently by other researchers such as Yang and Gerasoulis [23] and Kwok and Ahmad [10]. In this decomposition, the application graph is first mapped to a *fully-connected* multiprocessor architecture that has an unbounded number of processors. We call such an architecture a fully-connected, infinite processor (**FCIP**) architecture. In a fully-connected network, any number of processors can perform interprocessor communication simultaneously. The objective in the mapping onto an FCIP is the same as the overall objective — minimization of net execution time. In the second phase of Sarkar's two-phase process, called *merging*, the schedule derived for an FCIP is

1.This research was supported by the Defense Advanced Research Projects Agency (Contract #MDA972-00-1-0023) through Brown University, and the U. S. National Science Foundation (Grant #9734275).

mapped onto the given resource-constrained architecture. Our use of Sarkar's decomposition scheme and the associated breakdown of scheduling into different phases is motivated by the idea of introducing modularity and hence more flexibility in allocating compile-time resources throughout the optimization process. In this paper, we focus on the first phase (*clustering*) of this decomposed problem — the scheduling of a given task graph onto an FCIP architecture. Algorithms to address second phase have been discussed by other authors (e.g., see [21]).

2. Previous Work

Algorithms for IPC-conscious scheduling have received increasingly high attention in the literature. A wide variety of these algorithms have been proposed as scheduling heuristics that directly emphasize reducing the effect of IPC to minimize the net execution time [17, 23, 10], and are based on the framework of clustering algorithms; this group of algorithms is the main focus of this paper. Among existing clustering approaches are Sarkar's Internalization Algorithm (SIA) [17] and the Dominant Sequence Clustering (DSC) algorithm of Yang and Gerasoulis [23].

Sarkar's clustering algorithm has relatively low complexity. It starts with a complete solution and iteratively makes local changes to it, and thus is likely to get stuck in local minima. DSC, on the other hand, builds the solution incrementally. It makes changes with regard to the global impact on the net execution time, but only accounts for the local effects of these changes, and 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.

However, being deterministic in nature, neither SIA nor DSC 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. Liao et al. [12] average the normalized compile time (compile time per application graph node per processor) for each heuristic (across seven basic list scheduling heuristics) for various random graphs, and describe the effect of graph size and number of processors on the total compile time consumed. Kwok and Ahmad also measure and compare the running times of different algorithms in [10]; however, they do not study the implications from the compile time tolerance point of view. Additionally, since both works 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 simulated annealing (SA) algorithms or genetic algorithms (GA). The simulated annealing algorithms attempt to avoid getting trapped in local minima and have been successfully used for scheduling problems [15]. GAs have the same characteristic as SAs regarding local minima and also have other advantages, which will be discussed in section 3.2. Hou et al. [7], Wang and Korfhage [22], Kwok and Ahmad [11], Zomaya et al. [25], and Correa et al. [2] have proposed different genetic algorithms in the scheduling context. Hou and Correa 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 at each step check for the validity of the associated candidate solution and any time basic genetic operators (crossover and mutation) 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 net execution times (schedule makespans). However, this measurement provides only a rough approximation of compile time effi-

ciency. 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 6.

3. Background

We represent the applications that are to be mapped into parallel implementations in terms of the widely-used *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.
- $C : V \times V \rightarrow \aleph$ denotes a function that gives the cost (latency) of each communication edge. That is, $C(v, v) \equiv 0$ for all V; $C(v_1, v_2) = C(v_2, v_1)$ for all v_1, v_2; and $C(v_1, v_2)$ is the cost of transferring data between v_1 and v_2 if they are assigned to different processors.

We assume that task graphs have unique source and sink nodes. Arbitrary DAGs can be converted to this form by appropriately connecting dummy source and sink vertices [3].

3.1 Clustering and Scheduling

The **net execution time** is defined by the following expression:

$$\tau_N = max(tlevel(v_x) + blevel(v_x) \,|\, v_x \in V), \quad (1)$$

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.

Although a number of innovative clustering and scheduling algorithms exist to date, none of these provide a definitive solution to the clustering problem. Some prominent examples of existing clustering algorithms are:
- Dominant sequence clustering (DSC) by Yang and Gerasoulis [23],
- Linear clustering by Kim and Browne [9], and
- Sarkar's internalization algorithm (SIA) [17].

In the context of embedded system implementation, one limitation shared by algorithms such as these 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, the clustering techniques of Sarkar, Kim, and especially, Yang have 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 run-time-tolerance, the first of which applies the concept of genetic algorithms to develop a novel approach for multiprocessor clustering and scheduling.

3.2 Genetic Algorithms

Clustering and scheduling problems, being intricate and combinatorial in nature, are best characterized by their very large, complex and multi-modal solution spaces. Genetic algorithms (GAs), inspired by observation of the natural process of evolution, are commonly considered to perform well on nonlinear and combinatorial problems [6]. A GA operates on a population of solutions rather than a single solution in the search space, and evaluates the fitness of candidate solutions to the problem to

guide its search, whereas heuristics often rely on very problem-specific knowledge and insights to get good results. The basic operations of a typical genetic algorithm are summarized below [6]:

1. Map the search space of all possible solutions of the problem onto a set of finite strings (chromosomes) over a finite alphabet.
2. Randomly select the initial population (first generation) of solutions.
3. Compute the *fitness* (measure of the quality) of each individual in the population.
4. Perform *crossover* between pairs of individuals to create new individuals and replace the randomly-selected individuals with these new individuals.
5. Randomly *mutate a* small part of the resulting population from last steps.
6. Repeat the optimization process starting at point 3 until the population converges, or until some other stopping criterion is met.

Due to the continued challenge of time-constrained scheduling problems and the promising performance of GAs on similar problems, scheduling problems have attracted a great deal of attention in the GA community. However, to our knowledge there are no GA approaches to task graph clustering in the literature. This paper develops an efficient GA approach to clustering task graphs. More details about our genetic representation and operator (crossover, mutation, etc.) implementation are discussed in the following section.

4. GA Implementation Details

This section briefly describes our proposed clustering methods in three parts: (1) the problem model (in particular, the underlying assumptions about tasks and processors); (2) solution encoding; and (3) fitness evaluation.

4.1 System and Task Model

We schedule parallel tasks onto a homogeneous multiprocessor system to minimize the net execution time, as defined in (1). Following the conventional clustering phase model, we assume an FCIP architecture as the implementation target. Task execution on each processor is non-preemptive. Applications are represented by task graphs, as described in Section 3.

4.2 Solution Encoding

We propose a new framework for applying GAs to scheduling problems. Whereas traditional solution methods are typically sequence based [4], our solution representation encodes scheduling-related information 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 representation of clustering exploits the view of a clustering as a subset of edges in the task graph. Gerasoulis and Yang have suggested this view of clustering in their characterization of certain clustering algorithms as being *edge-zeroing* algorithms [5]. One of our contributions in this paper is to apply this subset-based view of clustering to develop a natural, 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 $\{0, 1\}^n$), canonical GAs (which use binary representations of solution spaces) provide near-optimal sampling strategies [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 non symmetric representation. Accordingly, 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.

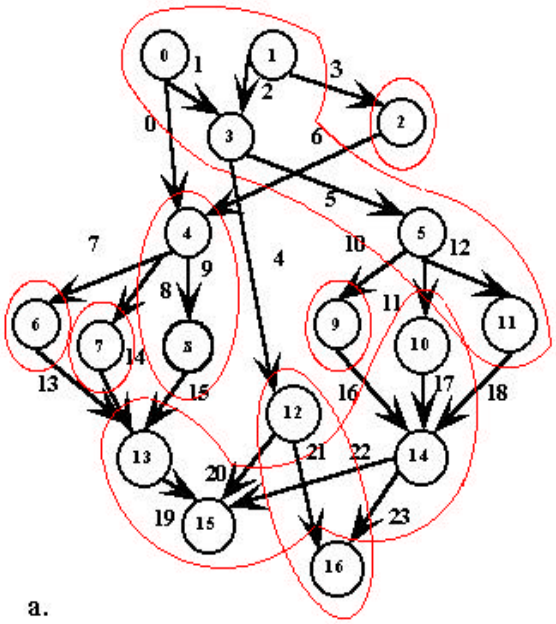
Our approach to encoding clustering solutions is based on the following definition.

Definition 1: Suppose that β is a subset of task graph edges. Then $f_\beta : E \rightarrow \{0, 1\}$ denotes the **clus-**

terization function associated with β. This function is defined by:

$$f(e_i) = \begin{cases} 0 \text{ if } (e_i \in \beta) \\ 1 \text{ otherwise} \end{cases}, \quad (2)$$

where E is the set of communication edges and e_i denotes the i th edge of task graph.When using a clusterization function to represent a clustering solution, the edge subset β is taken to be the set of edges that are contained in clusters. An illustration is shown in Figure 1. Because it is based on using



β = {1, 2, 5, 12} ∪ {9} ∪ {17, 19, 22} ∪ {21}
= {1, 2, 5, 9, 12, 17, 19, 21, 22}

Cluster	Nodes
c1	0, 1, 3, 5, 11
c2	2
c3	4, 8
c4	6
c5	7
c6	9
c7	10, 13, 14, 15
c8	12, 16

b.

e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	e_{13}	e_{14}	e_{15}	e_{16}	e_{17}	e_{18}	e_{19}	e_{20}	e_{21}	e_{22}	e_{23}
1	0	0	1	1	0	1	1	1	0	1	1	0	1	1	1	1	0	1	0	1	1	0	1

c.

Figure 1. (a) A clustering of a task graph; (b) the corresponding subset β of "zeroed" edges; (c) the corresponding clusterization function f_β.

clusterization functions to represent candidate solutions, we refer to our GA approach as the *clusterization function algorithm* (CFA).

In the CFA, the initial population is initialized with a random selection of clusterization functions (mappings from E into $\{0, 1\}$).

4.3 Fitness Evaluation

An outline of our proposed CFA approach is presented in Figure 2. As mentioned in section 3.2, a GA is guided in its search solely by its fitness feedback (lines 6 and 11 of the algorithm). The implemented search method in our research is based on *steady state genetic algorithms* [6] and fitness is calculated from the net execution time τ_N (from (1)), as follows:

$$Fitness = \tau_s / \tau_N, \quad (3)$$

where τ_s is the running time of the task graph when all tasks run sequentially. Thus, to evaluate the fitness of each individual in the population, we must first derive the unique clustering that is given by the associated clusterization function, and then schedule the associated clusters. Here, we have applied a modified version of list scheduling that abandons the restrictions imposed by a global scheduling clock, as proposed in the DLS algorithm [18]. Since processor assignment has been taken care of in the clus-

tering phase, the scheduler needs only to order tasks in each cluster and assign start times. The scheduler orders tasks based on the precedence constraints and the priority level [17] (the task with the highest *blevel* has the highest priority). Additionally, to reduce the processor idle times, a lower priority task can be scheduled ahead of a higher priority task if it fits within the idle time of the processor and also satisfies its precedence constraints when moved to this position.

The net execution time of the associated scheduled graph constitutes the fitness of each individual (member of the GA population), and the process of reproduction and evaluation continues as in Figure 2 while the time constraint is not violated.

5. Randomized Versions of Deterministic Clustering Algorithms

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 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 net execution time of the scheduled task graph, we have incorporated randomization into to the edge selection process when deriving randomized versions of DSC (RDSC) and SIA (RSIA). In the randomized version of each algorithm, we first sort all the edges based on the sorting criteria of the algorithm. The first element of the sorted list — the candidate to be zeroed — 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). Further details on this general approach to incorporating randomization into greedy, priority-based algorithms can be found in [24], which explores randomization techniques in the context of DSP memory management.

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 (pre-specified) compile-time tolerance (e.g., 10 minutes, 1 hour, etc.) is returned. Through extensive experiments, we have

1 **Algorithm** *CFA*
2 **Input**: A task graph specification of an application, with execution time and inter-processor communication estimates.
3 **Output**: An optimized clustering of the task graph onto multiple processors.
4
5 Generate initial population *k* using clusterization-function-based encodings
6 Evaluate fitness (clustering + scheduling)
7 **Repeat**
8 — Select *k* individuals according to their fitness values ("reproduction").
9 — Apply the crossover ("2-point crossover") operation *k*/2 times to generate *k* new "offspring" individuals ("recombination")
10 — Perform the mutation (randomly flip bits in the string with low probability) operation on selected individuals in the new population.
11 — Evaluate fitness (clustering + scheduling)
12 — Apply an elitist strategy (the top-ranked, fittest individual is never discarded) to the new population.
13 **Until** the time-constraint is met

Figure 2. A sketch of the proposed CFA algorithm for task graph clustering.

found the best randomization parameters for RSIA and RDSC to be 0.10 and 0.65, respectively.

6. Performance Evaluation and Comparison

In this section, we present an experimental comparison of DSC, SIA, RDSC, RSIA and CFA. We compare these heuristics using the widely used metric of speedup, which is defined by

$$Speedup = \tau_s / \tau_N , \tag{4}$$

where τ_N is the net execution time of the schedule produced by the given heuristic, and τ_s is the sequential (single-processor) running time, as defined in section 4.3.

To achieve valid comparisons, we also assume that the execution models, underlying architectures and objective functions for the heuristics are identical (these details were discussed in Section 3). The allotted running time for each input graph to RDSC or RSIA was determined from the CFA running time on the same graph for 3000 iterations (generations), which allows comparison under equal amounts of running time. All experiments were performed on an Intel Pentium III processor with a 1 GHz CPU speed.

All the heuristics have been tested with two sets of input graphs. The first set consists of 60 application graphs involving numerical computations (Laplace, Gaussian Elimination, etc., where the number of tasks varies from 10 to 50 tasks), and digital signal processing (DSP). The DSP-related task graphs include N-point Fast Fourier Transforms (FFTs), where N varies between 2 and 128; a collection of uniform and non-uniform multirate filter banks with varying structures and numbers of channels; and a compact disc to digital audio tape (cd2dat) sample-rate conversion application. Here, for each DSP application, we have varied the *communication to computation cost ratio* (CCR), which is defined by

$$CCR = \frac{\sum C(e)/|E|}{\sum t(x)/|V|} . \tag{5}$$

Specifically, we have varied the CCR between 0.1 to 10 when experimenting with each task graph.

The second set of input graphs consists of 140 random graphs in two sets: the first set (setI) consist of 6 subsets of graphs with CCRs of 0.1, 0.2, 0.5, 1, 2, 10. Each subset in turn is divided to small graphs (10 to 50 nodes) and large graphs (50 to 1000 nodes). The second set (setII) contains graphs with an average of 50 nodes and 100 edges and different CCRs (from 1 to 10).

To set the CFA algorithm parameters (size of population, number of generations, mutation probability, and crossover probability), we carried out a large number of experiments by varying these parameters and comparing the results. Based on the results of these experiments, we set the CFA genetic algorithm configuration to have a population size of 100, a number of generations equal to 3000, and mutation and crossover probabilities of 0.01 and 0.8, respectively.

The net execution times of application graphs and random graphs for the deterministic heuristics and CFA are shown in Figures 3 and 4, and the results for similar input graphs for RDSC, RSIA and CFA are given in Figures 5 and 6. It can be seen from the results that CFA consistently performs significantly better than the other approaches, and the benefit of the CFA approach increases with increasing CCR values. Overall, our experimental results show that CFA is preferable for compile time tolerances that accommodate the underlying GA configuration (less than 1 minute to 10 hours for the task graphs that we considered in our experiments). Our results are summarized further in the following section, along with our conclusions.

7. Summary and Conclusions

This paper has explored multiprocessor clustering techniques to exploit the increased compile time tolerance of the embedded systems domain, and achieve efficient mapping of applications onto multiprocessor architectures. 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

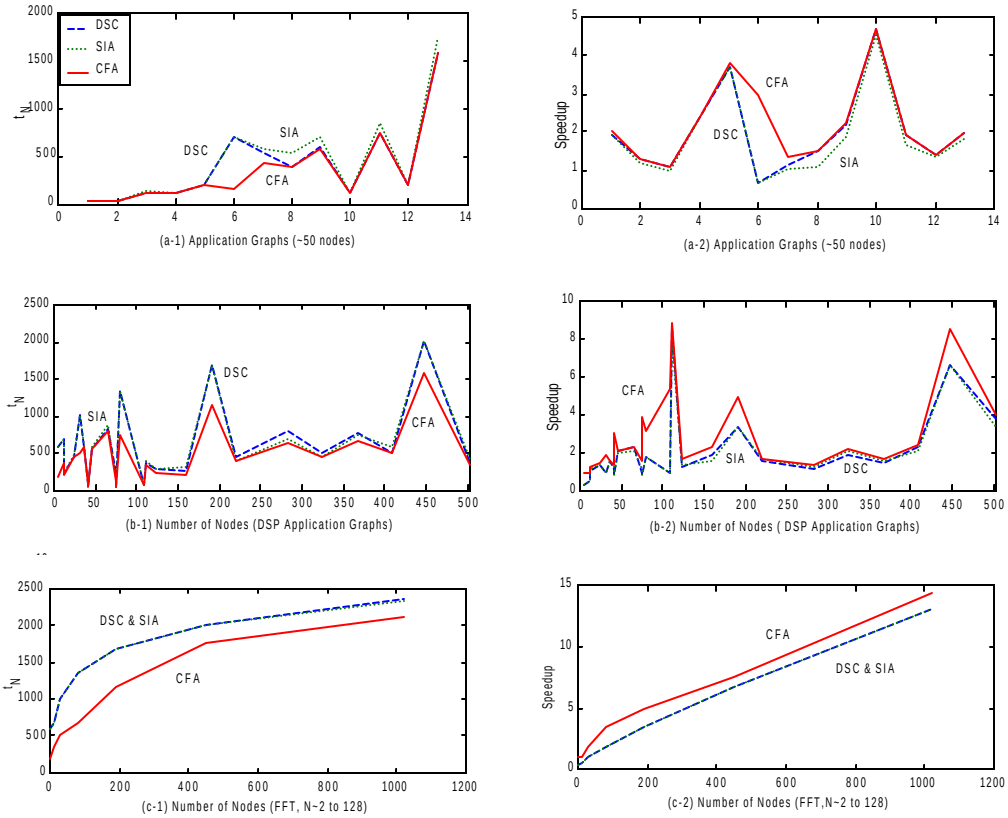


Figure 3. Performance comparison (net execution times and speedups) of the different heuristics for application graphs.

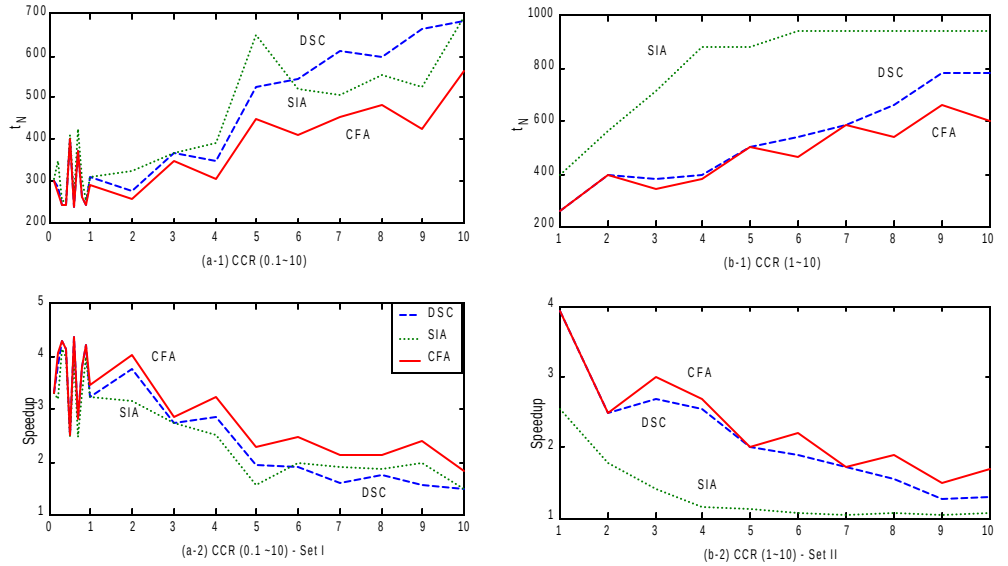


Figure 4. Performance comparison of different heuristics for random graphs.

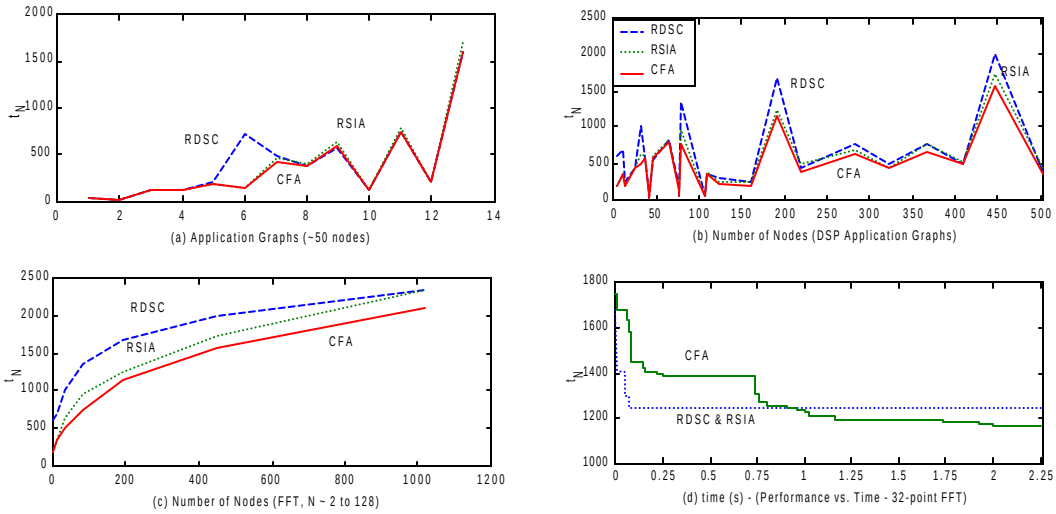


Figure 5. Performance comparison of CFA, RSIA and RDSC on application graphs.

two well-known deterministic algorithms, DSC [23] and SIA [17], respectively. RDSC and RSIA perform at least as well as DSC and SIA, but are able to exploit arbitrary increases in compile time tolerance due to their incorporation of probabilistic selection. Based on these developments, we have

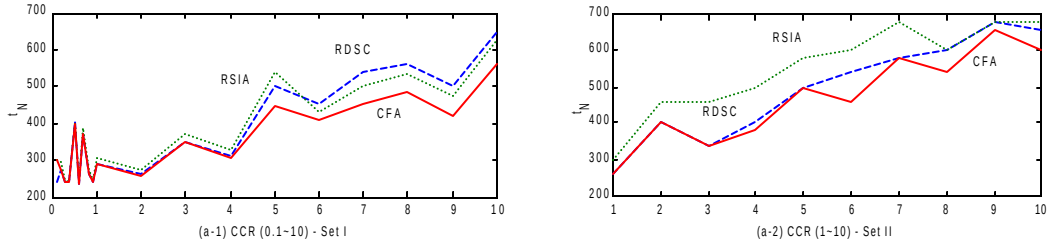


Figure 6. Performance comparison of CFA, RSIA and RDSC on random graphs.

performed an extensive experimental study that compares the alternative strategies under equal amounts of running time (compile time tolerance). Our experiments have demonstrated that the CFA algorithm significantly outperforms RDSC and RSIA, and that the improvement offered by CFA increases with increasing communication costs in the application relative to the amount of computation. Thus, CFA is especially useful when managing communication costs is important.

We have also observed that the performance of RDSC and RSIA varies significantly across various types of task graph structures. For example, RDSC appears to perform relatively poorly on task graphs that exhibit low parallelism and high interprocessor communication cost, or that contain multiple critical paths. Similarly, RSIA performs relatively poorly in the presence of uniform (homogeneous) communication costs across the task graph edge set. Presently, we are developing further experiments to quantify these distinctions. Another useful direction for further work is exploring the integration of merging algorithms into the CFA framework (e.g., in the fitness evaluation phase).

References

[1] T. Back, U. Hammel, and H.-P. Schwefel, “Evolutionary computation: comments on the history and current state,” *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 3-17, 1997.

[2] R.C. Correa, A. Ferreira, P. Rebreyend, “Scheduling Multiprocessor Tasks with Genetic Algorithms,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 0, 825-837, 1999.

[3] G. de Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[4] B. R. Fox and M. B. McMahon, "Genetic operators for sequencing problems," in *Foundations of Genetic Algorithms*, G. Rawlins, Ed.: Morgan Kaufmann Publishers Inc., 1991.

[5] A. Gerasoulis and T. Yang. "A comparison of clustering heuristics for scheduling directed graphs on multiprocessors." *Journal of Parallel and Distributed Computing*, Vol. 16, 276-291, 1992.

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

[7] E.S. H. Hou, N. Ansari and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 113-120, 1994.

[8] M. Ishikawa and N. McArdle. "Optically interconnected parallel computing systems." *IEEE Computer Magazine*, 61–68, February 1998.

[9] S. J. Kim and J. C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," in *Proc. of the Int. Conference on Parallel Processing*, 1-8, 1988.

[10] Y. Kwok and I. Ahmad, "Dynamic critical path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, 506-521, 1996.

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

[12] G. Liao, G. R. Gao, E. R. Altman, and V. K. Agarwal, "A comparative study of DSP multiprocessor list scheduling heuristics," in *Proc. of the Hawaii Int.Conference on System Sciences*, 1994.

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

[14] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAGS on multiprocessors," in *Proc. of the Int. Parallel Processing Symp.*, 1994, 446-451.

[15] A. K. Nand, D. Degroot, D.L.Stenger, "Scheduling directed task graphs on multiprocessor using simulated annealing," in *Proc. of the Int. Conference on Distributed Computer Systems*, 20-27, 1992.

[16] H. Printz, *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*. Ph.D. Thesis, school of computer Science, Carnegie Mellon University, May 1991.

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

[18] G. C. Sih and E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." *IEEE Transactions on Parallel and Distributed systems*, Vol. 4, No. 2, 1993.

[19] D. Spencer, J. Kepner, and D. Martinez, "Evaluation of advanced optoelectronic interconnect technology," MIT Lincoln Laboratory August 1999.

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

[21] T.Yang and A. Gerasoulis, "PYRROS: States scheduling and code generation for message passing multiprocessors," Proc. of 6th ACM Int. Conference on Supercomputing, 1992.

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

[23] T.Yang and A.Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 951-967, 1994.

[24] E. Zitzler, J. Teich, and S. S. Bhattacharyya. Optimized software synthesis for DSP using randomization techniques. Technical report, Computer Engineering and Communication Networks Laboratory, Swiss Federal Institute of Technology, Zurich, July 1999.

[25] A.Y. Zomaya, C. Ward, B. Macey, "Genetic scheduling for parallel processor systems: comparative studies and performance issues," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, 795-812, 1999.