Enhanced Genetic Algorithms for Task Scheduling in an Embedded Multiprocessor System-on-Chip System

*Yen-Hsiang Chang, Yi-Hsuan Lee, and Cheng Chen**
Department of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan, R.O.C.
Tel: (8863) 5712121 ext: 54734, Fax: (8863) 5724176
E-mail: {yhchang, yslee, cchen}@csie.nctu.edu.tw

Abstract

With the rapid evolution of submicron technology, manufacturers are integrating increasing numbers of components on one chip. Thus, the embedded multiprocessor *System-on-Chip* (*SoC*) system becomes one of the most attractive trends currently and in the future. By the limitation of portability and inexpensive packaging, SoC designers need an efficient scheduling technique to assign applications to the target realization while meeting both timing and power constraints. *Genetic Algorithm* (*GA*) is an appropriate scheduling method to solve this multi-objective problem. In general, GA can obtain the near-optimal solution but suffer from longer scheduling time. Hence, we propose two enhanced methods named *Constrained Genetic Algorithm* (*CGA*) and *Partitioned Genetic Algorithm* (*PGA*) to overcome this drawback. We also construct a simulation and evaluation environment to evaluate their performance. According to our experimental results, both CGA and PGA can not only obtain near-optimal solutions, but also dramatically decrease the scheduling time in comparison with standard GA.

Keywords: *Genetic Algorithm, Embedded system, Task scheduling, Power consumption*

1. Introduction

With the rapid evolution of submicron technology, manufacturers are integrating increasing numbers of components on one chip. Because several kinds of processor will be embedded and the heterogeneous architecture is adopted, the embedded multiprocessor *System-on-Chip* (*SoC*) system becomes one of the most attractive trends currently and in the future [1-2]. Fig. 1 contains a typical heterogeneous SoC architecture [3]. At the same time, the portability and the need for inexpensive packaging will limit power consumption to a few watts or less. Thus, SoC designers need a consistent system design technology that can cope with such characteristics. And this technology should also efficiently schedule these applications to the target realization while meeting all real-time and other constraints [4-5].

In order to consider more than one constraint during scheduling, *Genetic Algorithm* (*GA*) is an appropriate candidate. GA is a modern heuristic technique based on the principles of evolution and natural genetics, which can find a near-optimal solution usually [6]. However, its main drawback is to spend much time doing scheduling. Hence, in this paper, we propose two enhanced genetic algorithms to overcome this drawback, and construct a simulation and evaluation environment to evaluate their performance.

The first method is named *Constrained Genetic Algorithm* (*CGA*), which uses a restricted mechanism instead of randomness to generate the initial population with higher quality. According to our experimental results, CGA can exactly spend shorter time to find the

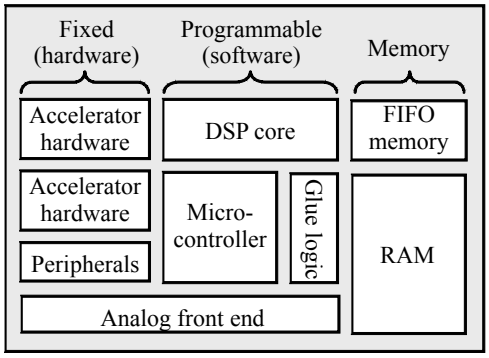


Fig. 1. Typical heterogeneous SoC architecture.

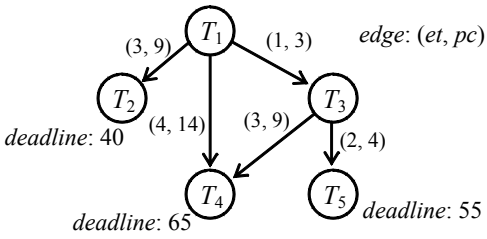
near-optimal solution as well as standard GA, because it doesn't waste time to evaluate many unfit individuals. Our second method is *Partitioned Genetic Algorithm* (*PGA*), which integrates the concept of *Divide-and-Conquer* mechanism to partition the entire problem and solve them individually. Like the essential advantage of *Divide-and-Conquer* mechanism, our simulation results illustrate that PGA can dramatically decrease the time doing scheduling in comparison with both standard GA and CGA.

The remaining of this paper is organized as follows. Section 2 introduces some fundamental background. The design issues and principles of our CGA and PGA are introduced in Section 3 and 4 respectively. In Section 5, we give some experimental results of our methods to demonstrate their figure and merits. Finally, some conclusions and future work are given in Section 6.

2. Fundamental Background

2.1 Problem Description

In order to formalize our scheduling problem, we first define an embedded multiprocessor SoC system and a parallel program. An embedded multiprocessor SoC system



(a)

	Time		Power	
	P_1	P_2	P_1	P_2
T_1	18	16	5	11
T_2	6	8	13	6
T_3	20	7	5	12
T_4	7	24	25	9
T_5	11	13	9	11

(b)

Fig. 2. (a) Task graph, (b) relation table.

is composed of a set of *m* heterogeneous computation components. They are connected by a common bus. Each component can execute at most one task at a time and task preemption is not allowed. The parallel program is described by an acyclic task graph as follows.

Definition 2.1 A task graph $TG = (T, E, et, pc)$ is an acyclic directed graph, where T and E are sets of execution tasks and dependence edges, and et and pc are functions from E to integer, representing execution time and power consumption of each edge respectively.

Because the embedded multiprocessor SoC system is heterogeneous, tasks executed by different computation components will cost different execution time and power consumption. In general, a component use less time executing a task usually costs much power consumption. We simply use a relation table to describe the execution time and power consumption of tasks in different components. Every *exit task*, which has no successors, is associated with a value representing its *deadline*. Fig. 2 is examples of task graph and relation table.

Given a parallel program to be executed on an embedded multiprocessor SoC system, the scheduling problem consists of finding a task schedule that minimize the power consumption of the parallel program under its timing constraints. A solution to a scheduling

problem is an assignment for each task of a starting time and a computation component. In general, optimizing allocation under time, power, and precedence constraints in a multiprocessor environment is an NP-hard problem [7-9].

2.2 Genetic Algorithm

A GA is a search algorithm that is based on the principles of evolution and natural genetics [6]. It combines the exploitation of past results with the exploration of new areas of the search space. By using *survival of the fittest* techniques combined with a structured yet randomized information exchange, a GA can mimic some of the innovative flair of human search [7].

A GA maintains a *population* of candidate solutions that evolves over time and ultimately converges. The individuals in the population are represented with a *chromosome*. Each individual has a numeric *fitness* value that measures how well this solution solves the problem. GA has a *selection* mechanism that chooses the fittest individuals of the current population to serve as parents of the next generation. GAs use two operators *crossover* and *mutation* to explore the search space. The crossover operator chooses randomly a pair of individuals among those selected previously, and exchanges some part of the information. The mutation operator takes an individual randomly and alters it. As natural genetics, the probability of applying mutation is very low while that of crossover is usually high [9].

The structure of the GA is a loop composed of a selection followed by a sequence of

T_i	T_1	T_2	T_3	T_4	T_5
$Comp(T_i)$	2	2	1	2	1

(a)

T_1	E_{12}	T_2	E_{13}	T_3	E_{14}	E_{34}	T_4	E_{35}	T_5
2	*N*	2	*B*	1	*N*	*B*	2	*N*	1

(b)

Fig. 3. (a) Schedule, (b) modified schedule.

crossovers and mutations. Moreover, the termination condition may be the number of iterations, execution time, results stability, etc. [7].

3. Constrained Genetic Algorithm (CGM)

Although GA usually can find a near-optimal solution, its main drawback is to spend much time doing scheduling. The aim of the *Constrained Genetic Algorithm* (CGA) presented in this section is to overcome this drawback.

3.1 Representation

A tricky question is how to represent a schedule in a way suitable for a heuristic algorithm. Fig. 3(a) is a schedule of task graph in Fig. 2(a). In Fig. 3(a), a pair T_i, $Comp(T_i)$ means that task T_i should be executed on component $Comp(T_i)$. The interpretation of the order in the schedule is that two tasks are explicitly ordered only if they are executed on the same component. Furthermore, tasks are ordered according to their precedence constrains defined in the task graph.

3.2 Initial Population

Most GAs generate individuals of the initial population randomly. This mechanism is intuitive, but qualities of these individuals may be uneven and need more generations to converge. In CGA, we determine a schedule with following rules:

I. Choose a task at random among those for which all predecessors are already scheduled.

II. Choose a component by the *roulette wheel* principle according to the power consumption.

These rules make a task have larger probability to be assigned to a component that causes less power consumption. Because this mechanism can prevent CGA from wasting time to evaluate many unfit individuals, it can achieve the quality of the initial population and let CGA converge much quickly.

3.3 Fitness Function

Before evaluating individuals, communication overheads should be considered. We insert edges into every schedule by using *As Late As Possible* (ALAP) mechanism. Fig. 3(b) is the modified schedule from Fig. 3(a). In Fig. 3(b), E_{ij} with marker *B* means that this communication overhead must be considered because T_i and T_j are executed by different components. Otherwise, marker *N* indicates this communication overhead is zero.

Next, the execution time and power consumption of an individual can be calculated straightforward. Power consumption of an individual is the summation of power cost by all tasks. As for the execution time, starting times of all tasks must be assigned at first. Each task can be started after all its predecessors are completed and the allocated component is free. We associate a flag *Start*(*T*) with every task *T*. If the starting time of *T* is determined by the maximal completing time of its predecessors, *Start*(*T*) is set to *P*. Otherwise, *Start*(*T*) is set to *C* representing its starting time is determined by the finish time of *Comp*(*T*). This flag will be

used later in Chapter 4. After assigning starting times to all tasks, the execution time of an individual can also be determined.

The aim for our scheduling problem is to minimize the power consumption under its timing constraints. A schedule is *legal* if and only if it doesn't violate any deadline constraint. We use formulas (3.1) and (3.2) to calculate fitness values.

$$F(s) = \begin{cases} MP - Power(s) & \text{for legal schedule} \quad (3.1) \\ [MP - Poser(s)] \times [MT - Time(s)] \times 0.001 & \text{for illegal schedule} \quad (3.2) \end{cases}$$

Power(*s*) and *Time*(*s*) are power consumption and execution time of schedule *s* respectively. *MP* and *MT* are two constants representing the maximal power and time, which can be calculated by executing all tasks sequentially at their worst components. The goal of formula (3.2) is giving illegal schedules a chance to be reproduced. This fitness function will assign a larger value for better schedule and make it survive in the next generation easily.

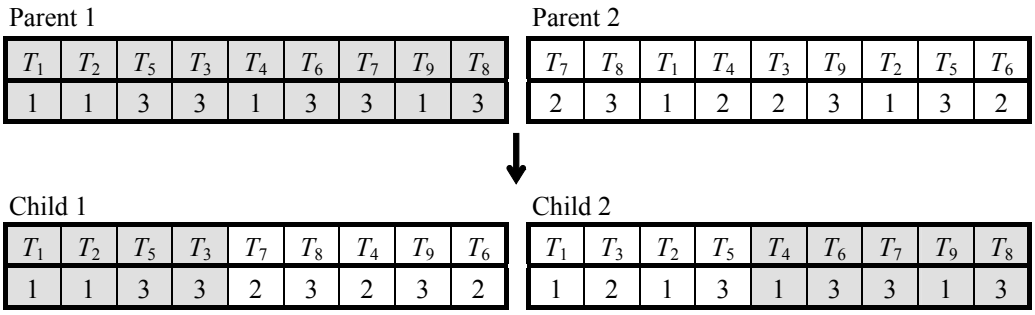
3.4 Selection

The selection is done using a biased roulette wheel principle. Thus, the better the fitness of an individual, the better the odds of it being selected.

3.5 Crossover

Crossover produces new individuals that have some portions of both parent's genetic material. Let s_1 and s_2 be two individuals that should generate two offsprings s_1' and s_2'. s_1' and s_2' are generated by the following rules:

I. Choose a task T_i randomly as the crossover point to separate s_1 into two parts.



T_1	T_2	T_5	T_3	T_4	T_6	T_7	T_9	T_8
1	1	3	3	1	3	3	1	3

T_7	T_8	T_1	T_4	T_3	T_9	T_2	T_5	T_6
2	3	1	2	2	3	1	3	2

T_1	T_2	T_5	T_3	T_7	T_8	T_4	T_9	T_6
1	1	3	3	2	3	2	3	2

T_1	T_3	T_2	T_5	T_4	T_6	T_7	T_9	T_8
1	2	1	3	1	3	3	1	3

Fig. 4. An example of crossover operation.

II. Keep the left part of s_1 to s_1' and right part of s_1 to s_2'.

III. Scan s_2. If T_i belongs to the left part of s_1, put T_i and $Comp(T_i)$ into s_2'. Otherwise put T_i and $Comp(T_i)$ into s_1'.

Fig. 4 shows an example of crossover operation.

3.6 Mutation

Mutation ensures that the probability of finding the optimal solution is never zero. It also acts as a safety net to recover good genetic material that may be lost through selection and crossover. We use two kinds of mutation operators. The first kind selects two tasks T_i and T_j randomly and swaps $Comp(T_i)$ and $Comp(T_j)$. The second kind selects a task T_i and alters $Comp(T_i)$ at random.

4. Partitioned Genetic Algorithm (PGA)

CGA can converge quickly compared with standard GAs, but it still needs considerable scheduling time when the number of tasks becomes large. Obviously, scheduling time of GA directly depends on the number of tasks being scheduled. Hence, we present a *Partitioned Genetic Algorithm* (PGA), which integrates the concept of *Divide-and-Conquer* mechanism

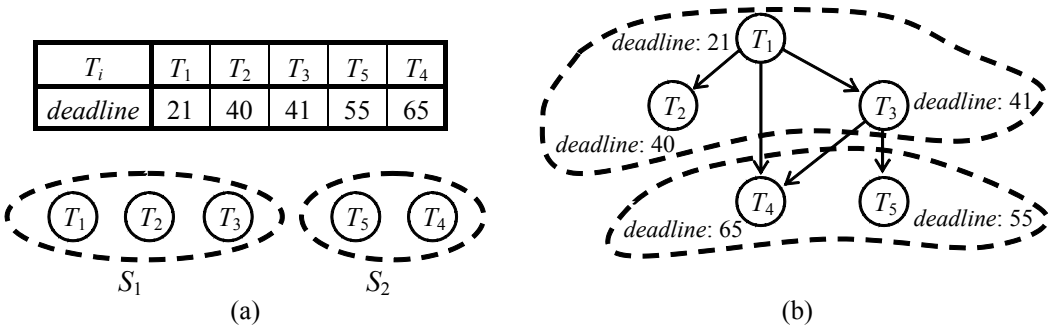


Fig. 5. An example of partitioning a task graph.

into CGA to decrease the number of tasks being scheduled at a time.

4.1 Deadline Partition Algorithm

The main steps of *Divide-and-Conquer* algorithm is to partition the problem into several parts, solve them individually, and merge them to form the final solution. In this subsection we present a *Deadline Partition* algorithm to partition the original task graph according to the deadline of every task. Unfortunately, by the definition of task graph, only exit tasks are associated with deadline values. Thus, before partitioning we simply use formula (4.1) to calculate deadline values of other tasks.

$$deadline(T_i) = \mathbf{MIN}\{deadline(T_j) - time(T_j)\} \qquad (4.1)$$

where T_j is a successor of T_i and $time(T_j)$ is the maximal execution time among all components

Steps of Deadline Partition algorithm are shown below:

I. Calculate the deadline value of each task by formula (4.1).

II. Sort tasks in increasing order according to their deadline values.

III. Partition tasks into subgroups evenly in sequence.

Fig. 5 is the result of partitioning task graph in Fig. 2(a) into two subgroups. Notice that any two subgroups cannot depend on each other; otherwise they both cannot be executed and

will cause a deadlock. Fortunately, the following Lemma shows that Deadline Partition algorithm can always generate legal partition result.

Lemma 4.1 Partition result generated by Deadline Partition algorithm is always legal.

Proof: Assume that $S_1 \ldots S_n$ are subgroups generated by Deadline Partition algorithm and tasks $T_i \in S_i$, $T_j \in S_j$, for $i < j$. Because Deadline Partition algorithm sorts and partitions tasks in sequence, it is obvious that $deadline(T_i) \leq deadline(T_j)$. From formula (4.1), T_i cannot be a successor of T_j. Thus, tasks in S_i will not depend on any task in S_j and the partition result is always legal.

4.2 Partitioned Genetic Algorithm (PGA)

After partitioning original task graph into subgroups, we apply CGA to schedule all subgroups individually in sequence. Finish time of every computation component in a subgroup is transferred to the next subgroup as the starting time of corresponded component. Because subgroups are scheduled one by one, we don't need an additional merge algorithm like standard *Divide-and-Conquer* mechanism. The final solution can be directly cascaded by local schedules generated by all subgroups.

4.3 Power Minimization Algorithm

In PGA, number of tasks scheduled by CGA each time is much less, so it can significantly decrease the scheduling time compared with original CGA. Nevertheless, PGA usually obtains inferior solution to CGA, or it cannot find a legal schedule even if the solution

S_1

T_i	T_1	T_2	T_3
$Comp(T_i)$	1	2	1

S_2

T_i	T_5	T_4
$Comp(T_i)$	1	1

(a)

S_1

T_i	EPD_{i1}	EPD_{i2}
T_1	0	-6
T_2	-7	0
T_3	0	-6

S_2

T_i	EPD_{i1}	EPD_{i2}
T_4	0	16
T_5	0	-3

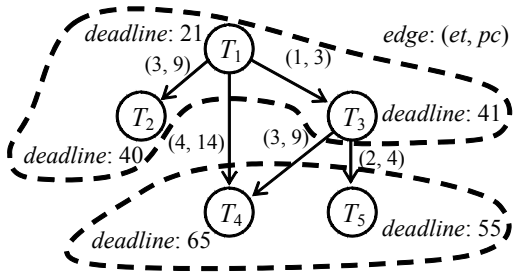
(b)

Fig. 6. (a) Schedules, (b) voluntary tables.

is existed essentially. This is because PGA only can obtain local optimal solutions, and the combination of local optimal solutions is usually not equal to the global optimal solution. In order to overcome this side effect, we present a *Power Minimization* algorithm to improve the solution generated by PGA.

Suppose that task T is executed by component i, and variable $EPD_{ij}(T)$ is defined as the difference of power consumptions while T is executed by component i and j. For each subgroup, we maintain a *voluntary table* to record *EPD* values of every task and component pair. Fig. 6 illustrates schedules of subgroups in Fig. 5 and their voluntary tables. In the voluntary table, a positive $EPD_{ij}(T)$ value indicates that if we reallocate T from component i to j can decrease the power consumption. This voluntary table is used as a referential material in Power Minimization algorithm.

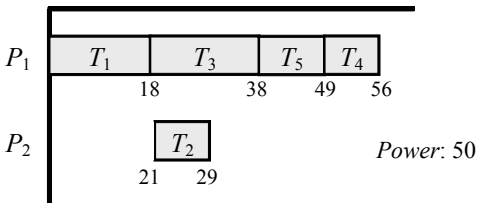
Before improvement we analyze two reasons that cause the side effect of PGA. The first is about the precedence constraints. Suppose tasks $T_i \in S_i$, $T_j \in S_j$, and T_i is a predecessor of T_j. When we schedule S_i, T_i is allocated to $Comp(T_i)$ which causes less power consumption but completes T_i just in time. In order to satisfy the deadline constraint of T_j, CGA will be forced to select $Comp(T_j)$ which costs much more power to execute T_j quickly. The second is caused



(a)

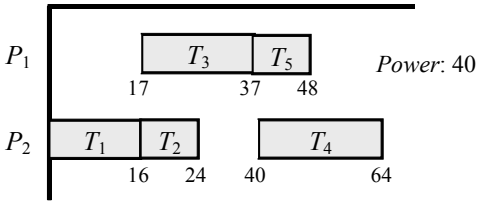
	Time		Power	
	P_1	P_2	P_1	P_2
T_1	18	16	5	11
T_2	6	8	13	6
T_3	20	7	5	12
T_4	7	24	25	9
T_5	11	13	9	11

(b)



(c)

S_1	T_i	EPD_{i1}	EPD_{i2}
	T_1	0	-6
	T_2	-7	0
	T_3	0	-6



(e)

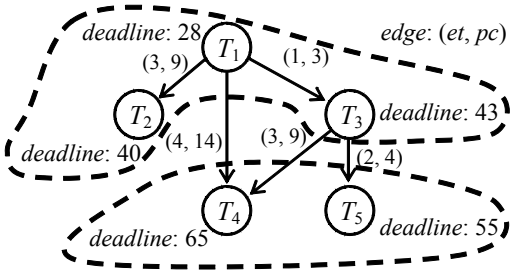
S_2	T_i	EPD_{i1}	EPD_{i2}
	T_4	0	16
	T_5	0	-3

(d)

Fig. 7. (a) Task graph, (b) relation table, (c) schedule generated by PGA, (d) voluntary tables, (e) schedule improved by Power Minimization algorithm.

by the finish time of the computation component. According to precedence constraints, we expect that T_j can be started from time t at component i to satisfy its deadline. But in fact component i finishes another task $T_k \in S_i$ later than time t, which causes T_j to violate its deadline constraint. It also forces T_j being allocated to component j that costs more power consumption. Both these two situations will let PGA generate worse solution than CGA.

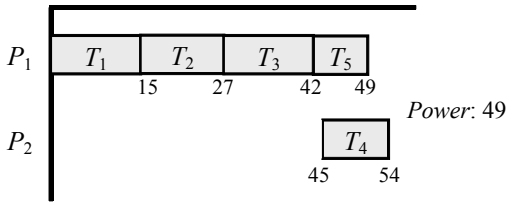
During scheduling process of CGA, every task T is assigned a flag *Start*(T) indicating the factor that determines its starting time. This flag also can distinguish the reason why the solution generated by PGA is inferior to CGA. After scheduling a subgroup, we construct its voluntary table and check every task in this subgroup to see if they can be reallocated to



(a)

	Time		Power	
	P_1	P_2	P_1	P_2
T_1	15	6	5	8
T_2	12	9	4	12
T_3	15	8	4	14
T_4	20	9	9	25
T_5	7	12	11	7

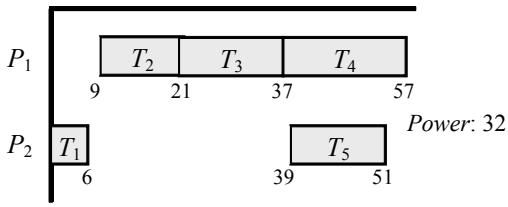
(b)



(c)

S_1

T_i	EPD_{i1}	EPD_{i2}
T_1	0	-3
T_2	0	-8
T_3	0	-10



(e)

S_2

T_i	EPD_{i1}	EPD_{i2}
T_4	16	0
T_5	0	4

(d)

Fig. 8. (a) Task graph, (b) relation table, (c) schedule generated by PGA, (d) voluntary tables, (e) schedule improved by Power Minimization algorithm.

decrease the power consumption. If *Start*(*T*) is *P*, predecessors of *T* are tried to be reallocated. Otherwise, we try to reallocate tasks executed by the same component with *T*. Based on the voluntary table, we only need to try components corresponded to positive *EPD* values. We still use task graph in Fig. 2(a) as an example. Fig. 7 and 8 are two examples that individuals generated by PGA are improved using Power Minimization algorithm. The entire scheduling flow of PGA is shown in Fig. 9.

5. Experimental Results

We construct a simulation and evaluation environment to evaluate proposed methods. Instead of randomly generating instances, we prefer to use four instances extracted from

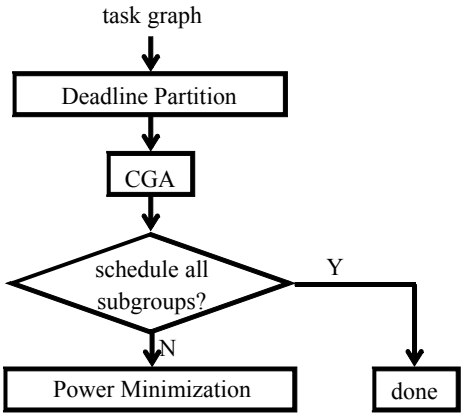
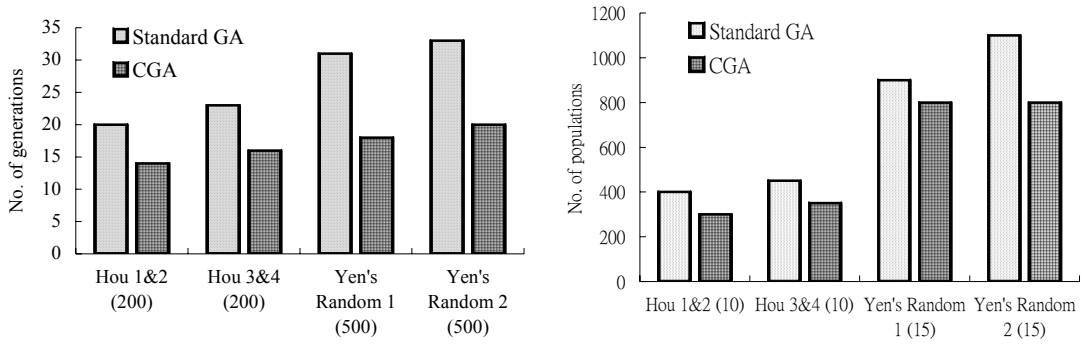


Fig. 9. Scheduling flow of PGA.

[10-11]. But number of tasks in these instances is less than 60, which is not large enough for our evaluation. Therefore, we use a tool *TGFF* (*Task Graph For Free*) developed by [12] to provide larger task graphs with hundred tasks. Meanwhile, we also choose method proposed in [4] as the standard GA to compare with, because its process is similar as CGA except for the generation of initial population.

In the following evaluation, we use the same system architecture and probabilities of crossover and mutation to compare the scheduling time spent by standard GA and CGA. Fig. 10(a) shows that with the same number of populations, CGA needs less number of generations than standard GA to obtain the optimal solution. On the other hand, as shown in Fig. 10(b), with the same number of generations CGA also needs less number of populations to converge. Both above two results indicate that CGA spends shorter scheduling time than standard GA to obtain the same solution. We also use task graphs with 100 and 200 tasks generated by TGFF to evaluate the scheduling time while the number of computation components increases. Fig. 11 indicates that CGA is superior to standard GA obviously.



* 2 computation components

* The number in the parenthesis on x-axis is the number of populations

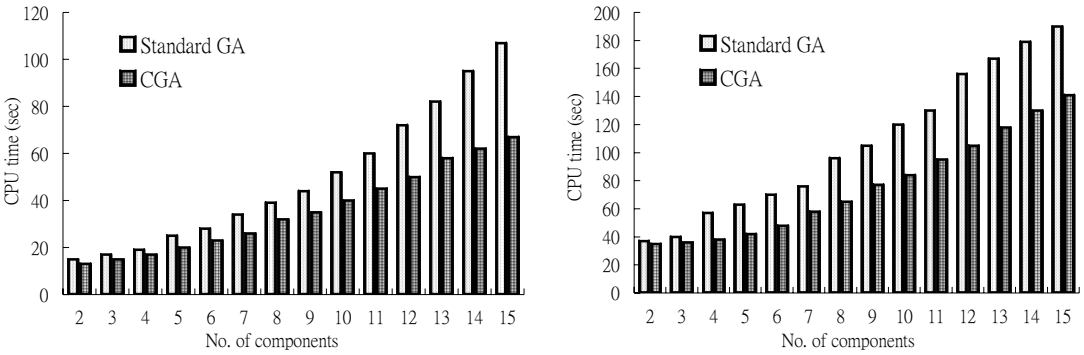
(a)

* 2 computation components

* The number in the parenthesis on x-axis is the number of generations

(b)

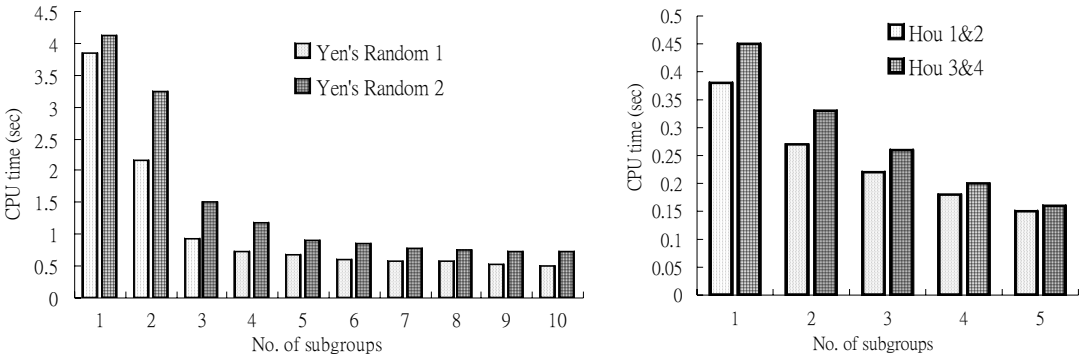
Fig. 10. Experimental results.



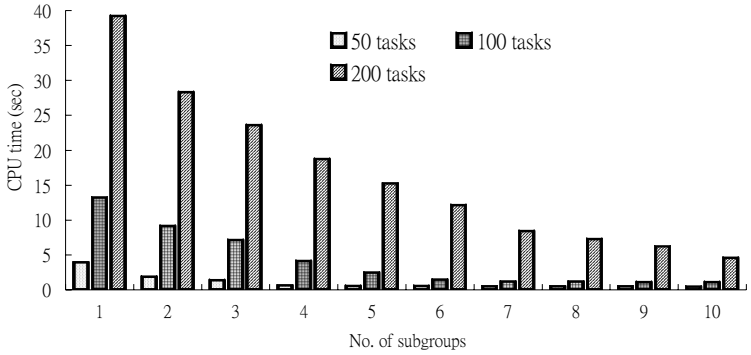
(a) (b)

Fig. 11. Experimental results.

From Fig. 12, we can see that PGA dramatically decreases scheduling time in both task graphs extracted from [10-11] and generated by TGFF. In these results, PGA with only one subgroup is essentially the same as CGA. Finally, Fig. 13 shows the results of PGA with and without applying Power Minimization algorithm. It is obvious that PGA may obtain inferior solution when the number of subgroups increases as mentioned in previous section. After applying Power Minimization algorithm, it can be decreased in some degree, although the final result is usually worse than CGA.

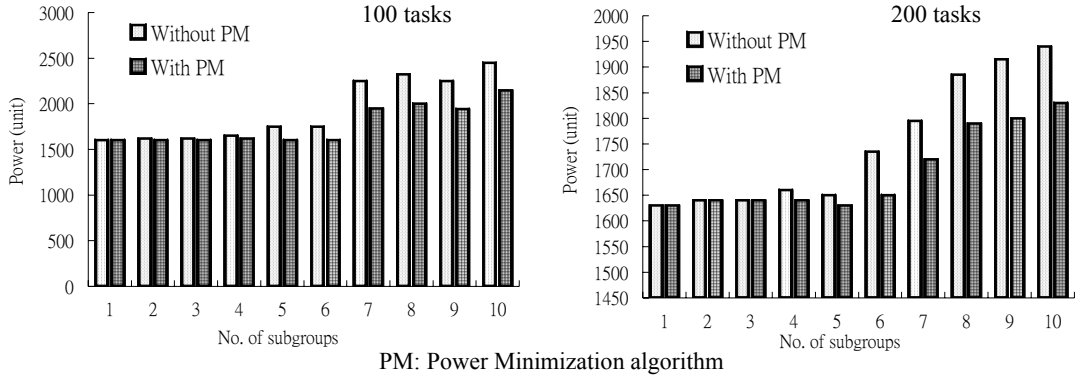


(a) (b)



(c)

Fig. 12. Experimental results.



PM: Power Minimization algorithm

(a) (b)

Fig. 13. Experimental results.

6. Conclusions and Future Work

In this paper we have proposed two enhanced genetic algorithms to schedule parallel program on embedded multiprocessor SoC system, and constructed a simulation and evaluation environment to evaluate them. We assume the embedded multiprocessor SoC

system is heterogeneous, and our scheduling goal is to find a task schedule that minimizes the power consumption under the predefined precedence and deadline constraints.

Because GA searches solutions randomly, it usually suffers from longer scheduling time. Therefore, we propose CGA to generate individuals in the initial population more restricted, which can make CGA converge much quickly. Besides, we also propose PGA that partitions the parallel problem into subgroups and schedules them by CGA individually, to further decrease entire scheduling time. According to our experimental results, both CGA and PGA can not only obtain near-optimal solutions as well as standard GA, but also spend less time doing scheduling.

In addition to previous features, there are still several promising issues in future researches. In standard *Divide-and-Conquer* mechanism, subgroups can be proceeded in parallel, and an additional merge algorithm are designed to cascade these individual solutions to form the final solution. But in our PGA, subgroups must be scheduled one by one in sequence, which cannot fully utilize the feature of *Divide-and-Conquer* mechanism. Therefore, it may be an interesting research topic to design another enhanced GA to eliminate this limitation. If the new algorithm can be successfully constructed, it is able to decrease the scheduling time more dramatically.

Reference

[1]. G. Silcott, J. Wilson, N. Peterson, W. Peisel, and K. L. Kroekar, “SoCs Drive New

Product Development", *Computer*, Vol. 32, Issue 6, pp. 61-66, June 1999.

[2]. H. De Man, "System-on-Chip Design: Impact on Education and Research", *IEEE Design & Test of Computers*, Vol. 16, Issue 3, pp. 11-19, July-Sep. 1999.

[3]. Peng Uang, Chun Wong, Paul Marchal, Francky Catthoor, Dirk Desmet, Diederik Verkest, and Rudy Lauwereins, "Energy-aware Runtime Scheduling for Embedded-multiprocessor SoCs", *IEEE Design & Test of Computers*, Vol. 18, Issue 5, pp. 46-58, Sep.-Oct. 2001.

[4]. Jaewon Oh, Hyokyung Bahn, Chisu Wu, and Kern Koh, "Pareto-based Soft Real-time Task Scheduling in Multiprocessor Systems", *Proc. of 7th Asia-Pacific Software Engineering Conference*, pp. 24-28, 2000.

[5]. R. P. Dick and N. K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for Hardware-software Cosynthesis of Distributed Embedded Systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 10, pp. 920-935, Oct. 1998.

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

[7]. Albert Y. Zomaya, Chris Ward, and Ben Macey, "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 8, pp. 795-812, Aug. 1999.

[8]. Ricardo C. Correa, Afonso Ferreira, and Pascal Rebreyend, “Scheduling Multiprocessor Tasks with Genetic Algorithms”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 8, pp. 825-837, Aug. 1999.

[9]. Man Lin and Laurence Tianruo Yang, “Hybrid Genetic Algorithms for Scheduling Partially Ordered Tasks in a Multiprocessor Environment”, *Proc. of 6th International Conference on Real-time Computing Systems and Applications*, pp. 382-387, 1999.

[10]. T. Y. Yen, **Hardware-Software Cosynthesis of Distributed Embedded Systems**, Ph. D Dissertation, Department of Electrical Engineering, Princeton University, Princeton, NJ, June 1996.

[11]. J. Hou and W. Wolf, “Process Partitioning for Distributed Embedded Systems”, *Proc. of International Workshop Hardware-software Codesign*, pp. 288-294, March 1996.

[12]. R. P. Dick, D. L. Rhodes, and W. Wolf, “TGFF: Task Graph For Free”, *Proc. of International Workshop Hardware-software Codesign*, pp. 97-101, 1998.