Design Automation for Embedded Systems, 6, 425–449, 2002.
© 2002 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Comparing Three Heuristic Search Methods for Functional Partitioning in Hardware–Software Codesign

THEERAYOD WIANGTONG tw1@ic.ac.uk
Department of Electrical & Electronic Engineering, Imperial College, Exhibition Road, London

PETER Y. K. CHEUNG p.cheung@ic.ac.uk
Department of Electrical & Electronic Engineering, Imperial College, Exhibition Road, London

WAYNE LUK wl@doc.ic.ac.uk
Department of Computing Imperial College, Queen's Gate, London

Abstract. This paper compares three heuristic search algorithms: genetic algorithm (GA), simulated annealing (SA) and tabu search (TS), for hardware–software partitioning. The algorithms operate on functional blocks for designs represented as directed acyclic graphs, with the objective of minimising processing time under various hardware area constraints. The comparison involves a model for calculating processing time based on a non-increasing first-fit algorithm to schedule tasks, given that shared resource conflicts do not occur. The results show that TS is superior to SA and GA in terms of both search time and quality of solutions. In addition, we have implemented an intensification strategy in TS called *penalty reward*, which can further improve the quality of results.

Keywords: Functional partitioning, genetic algorithm, hardware–software codesign, heuristic search algorithms, simulated annealing, tabu search.

1. Introduction

Task partitioning and task scheduling are required in many applications, for instance codesign systems, parallel processor systems, and reconfigurable systems. Sub-tasks extracted from the input description should be implemented in the right place (using the partitioner) at the right time (using the scheduler). It is well known that such scheduling and partitioning problems are NP-complete and are therefore intractable [1], [2]. Furthermore, in the case of reconfigurable hardware, such problems are considered as NP-hard [3]. Optimization techniques based on *heuristic methods* are generally employed to explore the search space so that feasible and near-optimal solutions can be obtained.

This paper focuses on algorithms involving heuristic search to solving hardware–software partitioning problems. Although heuristic methods do not guarantee the optimum point, they can produce acceptable solutions within a reasonable amount of time. Common heuristic-based algorithms include tabu search (TS), simulated annealing (SA), and genetic algorithm (GA). To the best of our knowledge, there has not been a systematic comparison of these three algorithms for hardware–software partitioning and scheduling. The novel contributions made in this paper are:

1. We compare TS with GA and SA, and find that for randomly generated task graphs as well as some task graphs normally encountered in real applications, TS is superior to GA and SA in terms of search time and quality of solutions.

2. We introduce an approach that uses processing time (makespan) as a system cost and also an architectural model for estimating processing time based on a non-increasing best-fit algorithm, in which hardware tasks are scheduled without resource conflicts.

3. We extend TS to include *penalty reward*, based on an intensification strategy, and prove that it can improve the quality of solutions obtained from search space.

The rest of the paper is organised as follows. Section 2 describes previous work on partitioning and scheduling. Section 3 provides an overview of the three heuristic algorithms used, and introduces the method implemented in TS to intensify searching in promising regions. Section 4 introduces the reference system architecture used in the comparison, and the model for estimating processing time as a measure of design quality. Section 5 consists of experimental results and discussions. In the last section, we draw some conclusions and propose future work.

2. Previous Work on Partitioning and Scheduling

Research in hardware–software codesign encompasses many interesting areas of research such as system specification and modeling, partitioning and scheduling, compilation, system co-verification, co-simulation, code generation for hardware and software, and hardware–software interfacing. However the common objective is to develop design methodologies and tools for systems containing both hardware and software.

A number of hardware–software codesign systems have been reported in the literature and they focus on different aspects in the design process. For instance, in *COSYMA* [14] hardware–software partitioning is based on simulated annealing. Tasks in the initial software-only solutions are mapped to hardware in order to provide the necessary acceleration so that system timing constraints are met. In contrast the *VULCAN* system is a hardware-oriented approach. It starts with a complete hardware solution and tries to reduce system cost by moving the non-time-critical parts to software. Other tools such as *Ptolemy* [26] and *Chinook* [27] emphasise different problems in codesign rather than partitioning or scheduling. *Ptolemy* concentrates on hardware–software co-simulation and code generation. *Chinook* focuses on the synthesis of interface between hardware and software.

Partitioning is a crucial step in codesign [9], [14], [16], [17] because it plays an important role in allocating tasks properly between hardware and software under system constraints. The results of partitioning directly affect system cost and performance. The approach used for partition is dependent on the type of scheduling employed. Two types of scheduling are common: pipelined scheduling and sequential scheduling. Partitioning for pipelined scheduling is very different from partitioning for sequential scheduling [11]. The objective of partitioning for sequential implementation is to minimize the time for one complete execution of the directed acyclic graph (DAG), while partitioning for pipelined scheduling intends to obtain the least number of pipelined stages and memory requirement, and also to satisfy constraints on pipeline stage delay time.

An example of pipelined partitioning is described in [12] where tasks are partitioned and pipelined to satisfy the system throughput requirement. As a result the pipeline stage delay becomes the system constraint. Partitioning and scheduling are characterized into functional blocks in the spatial domain. The *functional partitioning* approach has several advantages that make it a good technique for hardware–software codesign [9]. The most important feature is that both software solutions and hardware implementations [20] can be obtained simultaneously. Another pipelined partitioning method is reported in [11] where the partitioner uses a branch-and-bound approach to minimize the *initiation interval* in a pipelined implementation. (Initiation interval is the time difference between the start of two consecutive iterations after steady state is reached.)

The work reported in this paper involves partitioning in functional level for sequential scheduling rather than pipeline scheduling. Each precedence level, containing both software and hardware tasks, is sequentially executed from the top level down. In the same precedence level, however, hardware and software tasks can run concurrently.

There are many algorithms are available to solve the optimization of the partitioning and scheduling problem [10], [13], [14], [15]. These include dynamic programming, branch-and-bound, a list scheduling, integer linear programming, graph partitioning, simulated annealing and genetic algorithm. Some hybridization approaches combine several algorithms to solve the problem. Common to all these algorithms are the following objectives: fast execution; applicability to hardware–software partitioning; scalability with ease of inclusion of new constraints; consistently yielding good quality results.

Comparisons between these optimization algorithms have been reported recently. For example, there is a comparison between simulated annealing (SA) and genetic algorithm (GA) in [9], and GA was found to yield the best solutions. However, the system architecture used in that paper is simple and may not be applicable to complex systems. In contrast, our model can deal with resource conflicts. There is also a comparison between SA and GA in [6] but the context is to solve routing problem in communication networks.

While SA and GA are commonly used to overcome the intractability of partitioning and scheduling problems [22], the tabu Search (TS) method is less popular. TS was described and compared with SA in [22] and, surprisingly, TS achieved higher performance. However the focus of that work is to minimise the communication cost without taking into account resource conflicts. Also there is no attempt to utilise re-annealing strategy in SA and intensification strategy in TS.

While [19] covers a comparison of the three heuristic search methods (SA, GA and TS), its main emphasis is on hardware–software system architecture synthesis. Functional partitioning in [19] is obtained by an optimal branch-and-bound algorithm and only the component selection is optimized using heuristic methods.

The aim of this paper is to compare the GA, SA and TS algorithms for hardware–software partitioning. The algorithms will be used to minimize processing time subject to area constraints, precedence constraints and constraints due to shared resource conflicts.

3. Heuristic searches

The following section describes how the three optimization algorithms are adapted and modified to solve the partitioning problem. The fitness function and modifications in GA is described. The strategy used to intensify searching in local regions to improve the efficiency of TS, called *penalty reward*, is explained.

Briefly, GA is a general search algorithm [6]. It is versatile and effective for solving combinatorial optimization problems [8]. To apply GA to a particular problem, it is common to use a non-standard chromosome representation or a problem-specific genetic operator.

SA is a search algorithm that avoids becoming trapped in local optima by using Boltzmann distribution. The crucial factor in getting a good result comes from the "cooling schedule" [6], [7]. Localized simulated annealing (LSA) divides search space and provides a better control over temperature and annealing speed in different subspaces.

TS is a systematic approach to searching a solution space to avoid cyclic searching or being trapped in local optima. Reactive tabu search (RTS) is an improved version of tabu search. It uses a simple feedback scheme to determine the value of the *prohibition parameter* in TS [6]. In this paper, rather than only diversifying searching, we adapt penalty strategy to intensify searching in particular promising regions of the solution space. TS with this adaptation—called Tabu Search with Penalty Reward, TSPR—offers better results compared to those from the standard TS.

3.1. Genetic Algorithm

Genetic algorithm is based on Darwinian natural evaluation and selection. The algorithm has four main operations: evaluation, selection, crossover and mutation. Starting from an initial population, the degree of "*fitness*" for each member of the population is evaluated according to a *fitness function*. A set of parents is then selected from this population based on the fitness evaluation to breed a new generation of candidate solutions. The desirable features in each parent are encapsulated in a code associated with each parent, and this code is known as its "*chromosome*." Each chromosome is made out of a number of "*genes*," each capturing a desirable feature. Two children are reproduced from two parents by first randomly dividing each parents chromosome into two sets of genes, and then mixing them in a crossover fashion. This process, known as "*crossover*," helps to transmit the good features of parents into the next generation. Finally, *mutation* is used to avoid the searching process being trapped in local minima. This is done by occasionally (say with a probability of 0.01) modifying a gene (i.e., mutated). Figure 1 depicts the GA algorithm.

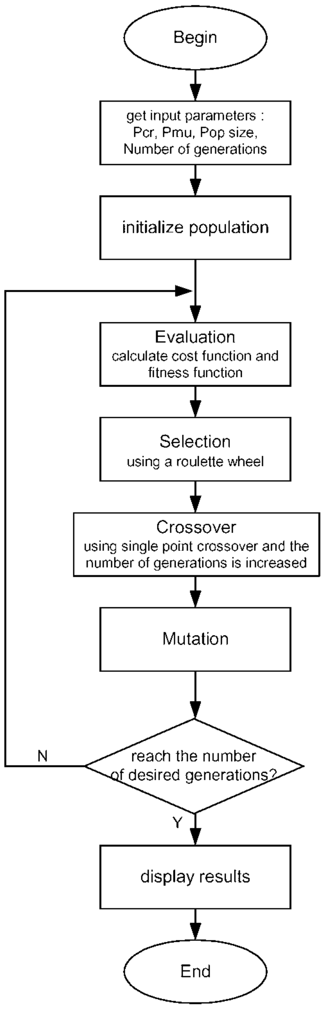

Begin
get input parameters : Pcr, Pmu, Pop size, Number of generations
initialize population
Evaluation calculate cost function and fitness function
Selection using a roulette wheel
Crossover using single point crossover and the number of generations is increased
Mutation
reach the number of desired generations?
N
Y
display results
End

Figure 1. GA algorithm.

The efficiency of GA in solving a given problem depends heavily on the representation of desirable characteristics in the chromosome, a good choice of the fitness function, and the appropriate use of the crossover mechanism. In our formulation, each individual in a population is characterised by a chromosome with the number of genes equal to the number of tasks or functional blocks. A gene is encoded as “0” if the task is implemented in software, and “1” if it is mapped to hardware.

To calculate the fitness of a member in a population, we first define the cost of the member as:

$$
\begin{aligned}
C_i &= K_1 A_i + \phi B_i \qquad && \textit{if } A_i > 0 \\
&= \phi B_i && \textit{if } A_i \leq 0
\end{aligned}
\tag{1}
$$

where C_i = cost of member i in the population

K_1, K_2, ϕ = constants

$A_i = (HwArea_i - A_{constraint})$ is the amount of hardware area exceeding the area constraint

B_i = processing time of member i in the population

Fitness F_i is defined as:

$$F_i = \max(C_i) - C_i \tag{2}$$

while the normalised fitness is defined as:

$$\hat{F}_i = \frac{F_i}{\sum\limits_{all\ i's} F_i} \tag{3}$$

In order to ensure that hardware area does not exceed the area constraint $A_{constraint}$, K_1 must be chosen to be significantly larger than ϕ. We also investigate the use of an alternative cost function defined as:

$$
\begin{aligned}
C_i &= A_i^{K_2} + \phi B_i \qquad && \textit{if } A_i > 0 \\
&= \phi B_i && \textit{if } A_i \leq 0
\end{aligned}
\tag{4}
$$

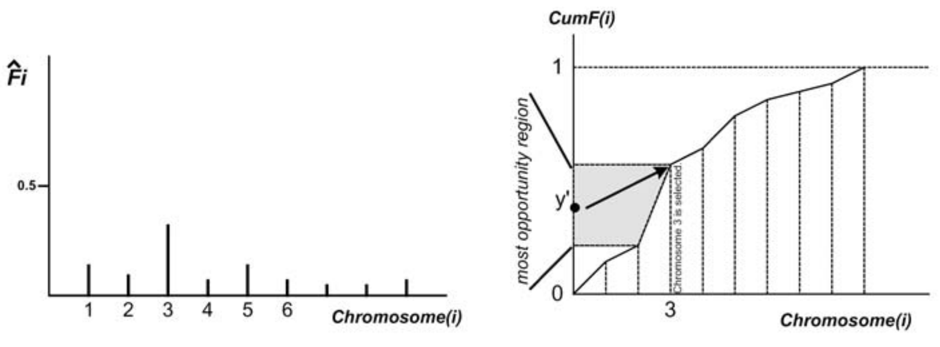
These cost functions are adapted from those found in [9]. The objective of the optimization procedure is to maximize the fitness values and to minimize the cost.

In order to perform selection, we first construct a cumulative fitness graph based on:

$$CumF_i = \sum_{1}^{i} \hat{F}_i \tag{5}$$

This is shown in Figure 2(b). A roulette wheel selection method is applied by generating a random value between 0 and 1. For example, if y' is generated, the chromosome 3 will be selected as a parent. It can be seen that the larger the fitness value of a chromosome, the higher the gradient on this curve, and the higher the probability of it being selected. Based on this selection procedure, a set of parent population is chosen for crossover reproduction. A chromosome can be selected more than once into the set of survivors.

A conventional single cut-point crossover scheme is employed to generate two children for every two surviving parents. In GA, because we want to maintain population size in



(a) Normalized fitness (b) Cumulated normalized fitness

Figure 2. Normalized and cumulated normalized fitness for roulette wheel selection.

each generation, some new children possibly exceed the area constraint due to a random crossover operation. However, these children have a high chance to be rejected in the selection procedure because of having a small value of fitness.

Finally, mutation is implemented by randomly flipping individual bits in the child chromosome with a small probability value (e.g., less than 0.01).

3.2. Simulated Annealing

Simulated Annealing (SA) was first introduced in 1953 by Metropolis *et al.* [6], [21]. Thirty years later, Kirtpatrick [25] suggested that this technique could be used to search feasible solutions in optimization problems. Since then the method has become very popular in solving many combinatorial optimization problems because of its simplicity, ease-of-use, robustness for a large number of problems, and the ability to avoid being trapped in local optima.

SA is adapted from the well-known neighborhood search (NS), a hill-climbing algorithm. Unlike NS which always chooses the best solution in the neighborhood and thus missing a better solution further away, SA accepts an inferior solution in the neighborhood according to a probability function. This probability is set high at the beginning of the optimization process, but is gradually reduced to zero. This is akin to the annealing process where the temperature of annealing is gradually reduced. The rate of the drop of the acceptance probability is governed by a function known as the *cooling schedule*. As the temperature cools to a predefined threshold, a solution is arrived.

It is well known that the choice of the cooling schedule is important [6]. It affects the efficiency of the algorithm and the quality of the solution. Too fast a cooling schedule can result in the algorithm stopping early at some local minimum; too slow a cooling schedule makes the algorithm jump more or less randomly for a long time before settling to a solution. Two popular cooling schedules used in this paper are:

1. Geometric schedule

$$T_{new} = \alpha T_{old} \tag{6}$$

α is a constant close to 1 (typically in the range of 0.9–0.99). k is the number of iterations given by $k = [log(T_f) - log(T_s)]/log(\alpha)$ where T_s = starting temperature, T_f = finish temperature.

2. Lundy and Mees schedule [21]

$$T_{new} = \frac{T_{old}}{(1 + \beta T_{old})} \tag{7}$$

β is a constant near to zero or equals to $(T_s - T_f)/(kT_sT_f)$ where k is the number of iterations.

The SA algorithm used to solve the partitioning problem in this study tries to minimize processing time while staying in the boundary of area constraints. Our implementation of SA has the following features:

1. It supports two popular cooling schedules described above and provides a comparison between them.
2. The cost function is chosen to minimize processing time without violating hardware area constraints.
3. New solutions are generated by randomly swapping a task between hardware and software in the task graph.

A re-annealing strategy is also used: when a move is rejected, the temperature is increased (for example, by the rate of the equation $T_{new} = T_{old}/(1 - \gamma T_{old})$). To avoid being trapped in an infinite loop, the increasing rate of temperature is made much lower than the decreasing rate (e.g., $\beta = 100\gamma$). The two cooling schedules, geometric and Lundy / Mees with re-annealing, are compared. The better one is selected for comparison with the GA and TS algorithms.

The flowchart of SA algorithm is depicted in Figure 3. An initial solution that obeys the area constraint is first generated. A neighborhood searching is performed by randomly changing one state bit in the solution, effectively swapping between hardware and software implementation of one of the tasks. A neighborhood solution is only considered if it obeys the area constraint. The cost difference between the current solution (S_{now}) and the neighboring solution (S_{neigh}) is calculated. In our formulation, the cost is a direct function of processing time. The neighborhood selection process already guarantees that the area constraint will be met.

The acceptance (or not) of a more costly neighboring solution is determined by the Boltzmann probability distribution function:

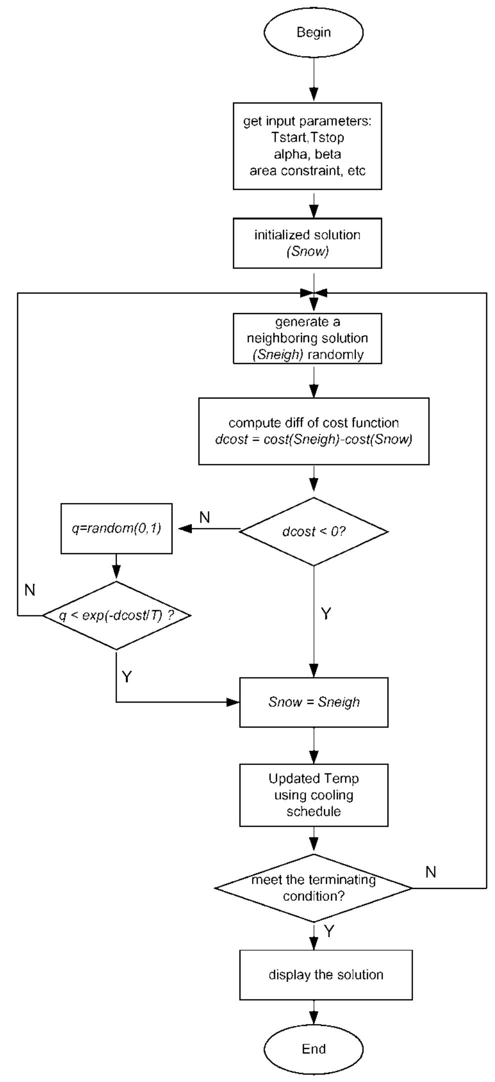

Begin
get input parameters: Tstart,Tstop alpha, beta area constraint, etc
initialized solution (Snow)
generate a neighboring solution (Sneigh) randomly
compute diff of cost function dcost = cost(Sneigh)-cost(Snow)
dcost < 0?
N
q=random(0,1)
q < exp(-dcost/T) ?
N
Y
Y
Snow = Sneigh
Updated Temp using cooling schedule
meet the terminating condition?
N
Y
display the solution
End

Figure 3. Simulated annealing.

$$q = \exp\left(-\frac{cost\ difference}{temperature}\right) \tag{8}$$

An accepted move replaces the current solution and then the temperature is reduced according to one of the two cooling schedules described earlier. The optimization loop terminates when the following conditions are met: 1) when the new temperature reaches T_{stop}; or 2) there is no improvement over the last 500 iterations.

3.3. Tabu Search with Penalty Reward

In contrast to simulated annealing (SA) which exploits random moves, tabu search (TS) exploits data structures of the search history as a condition of the next moves. More generally, TS is a search method designed to cross boundaries of feasibility normally treated as barriers, and it systematically imposes and releases constraints to allow the exploration of forbidden regions [6].

The major distinction of TS compared to the other search methods is that it uses a short-term memory, *recency-based memory*, to store recent search areas and a long-term memory, *frequency-based memory*, to store frequency of searching in each area. It is able to avoid searching the same neighborhood recorded in the memory (tabu status). However, after a given time has elapsed (depends on short-term memory size), the tabu status will be released and the area become eligible for searching again.

For the frequency-based memory, the frequency information will be used to penalize non-improving moves by adding a larger penalty to a neighbor that stays in a region of greater frequency counts. This frequency penalization can be used for both diversification and intensification strategies as explained in [21]. We introduce a way to implement these strategies in this paper, called *penalty reward*. We use two different values of penalty weights. The large value will be used to diversify searching while the small value will be used to intensify searching in promising local regions. The details will be explained later.

Sometimes, *aspiration criteria* are introduced in TS to determine when tabu restrictions can be overruled. The appropriate uses of the criteria might be important for improving the efficiency of TS. A simple type of aspiration criterion, as used in this paper, is that when a trial move yields a solution better than the best so far, the move will be picked although it is in tabu status.

Figure 5 shows the structures of recency-based memory and frequency-based memory used in this paper. The tabu list is kept in recency-based memory. After searching a new set of neighbors for the current solution, one of the neighbors is selected by using conditions shown in blocks A, B and C in Figure 4. Only the neighbors not in the tabu list are placed on the top of the FIFO memory, and such neighbors are said to have the maximum tabu degree. However, in the next iterations, they are shifted down to reduce the degree of tabu. Values of tabu degree between 7 to 20 appear to work well for a variety of problem classes, while values between $0.5\sqrt{N}$ and $2\sqrt{N}$ appear to work well for other problems where N is a measure of the problem dimension [21].

Frequency-based region is a long-term memory used to store the frequency of visiting to defined regions. For a problem consisting of N tasks each of which could be in hardware represented by "1" or in software represented by "0," the entire search space can be represented by an N-bit state value. Storing the frequency of visit for the entire space can be extremely memory intensive. Therefore we define a *region* as the k least significant bits of the search space as shown in Figure 5(b). Each visited neighborhood will have its corresponding region frequency count incremented. For example, if $k = 4$ as in Figure 5 (b), and the current visited neighbor is as shown, then frequency-based memory at address "0100" will be incremented by 1.

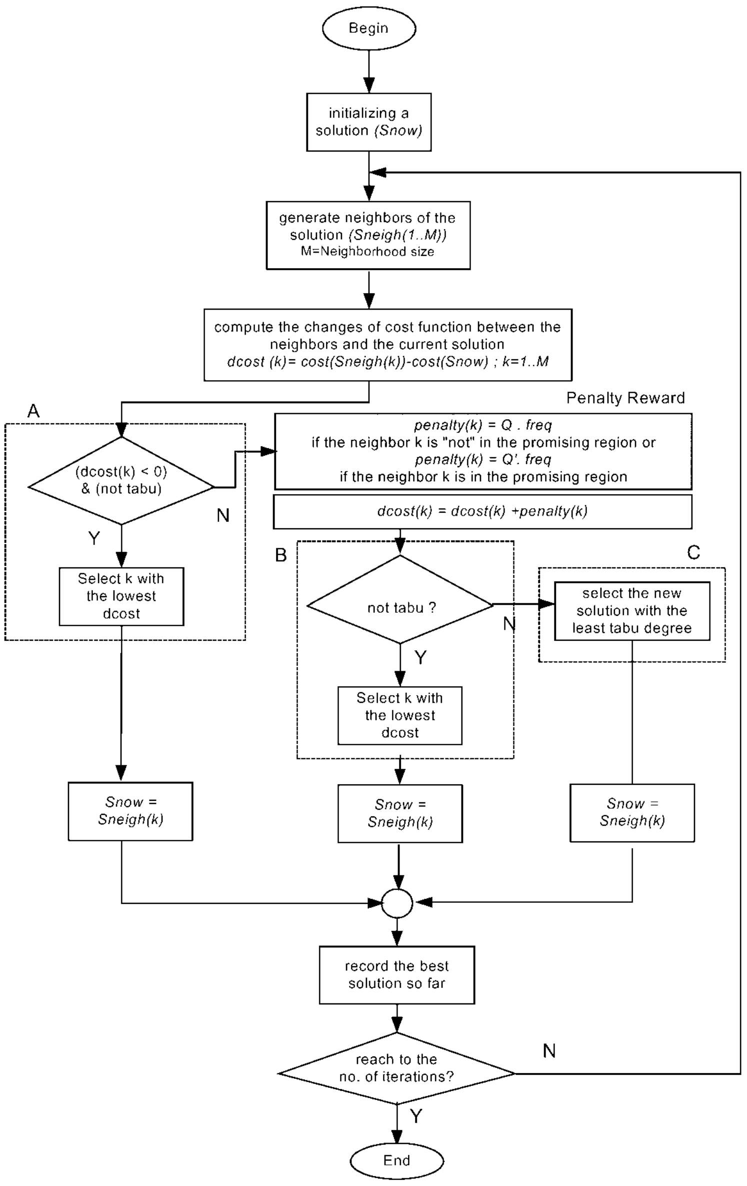

Begin
initializing a solution (Snow)
generate neighbors of the solution (Sneigh(1..M)) M=Neighborhood size
compute the changes of cost function between the neighbors and the current solution dcost (k)= cost(Sneigh(k))-cost(Snow) ; k=1..M
Penalty Reward
penalty(k) = Q . freq if the neighbor k is "not" in the promising region or penalty(k) = Q'. freq if the neighbor k is in the promising region
dcost(k) = dcost(k) +penalty(k)
A
(dcost(k) < 0) & (not tabu)
Y
N
Select k with the lowest dcost
B
not tabu ?
Y
N
Select k with the lowest dcost
C
select the new solution with the least tabu degree
Snow = Sneigh(k)
Snow = Sneigh(k)
Snow = Sneigh(k)
record the best solution so far
reach to the no. of iterations?
N
Y
End

Figure 4. Tabu search—blocks A, B and C are used to new neighboring solution.

Data in frequency-based memory are scaled by a positive constant Q which is then used as penalty values. This penalty is added to the cost difference $dcost(k) = cost(S_{neigh}(k)) - cost(S_{now})$, where $cost(\,)$ is the cost function, S_{now} is the current solution and $S_{neigh}(k)$ is the k^{th} neighbor of the current solution. The purpose of such penalty modification is to

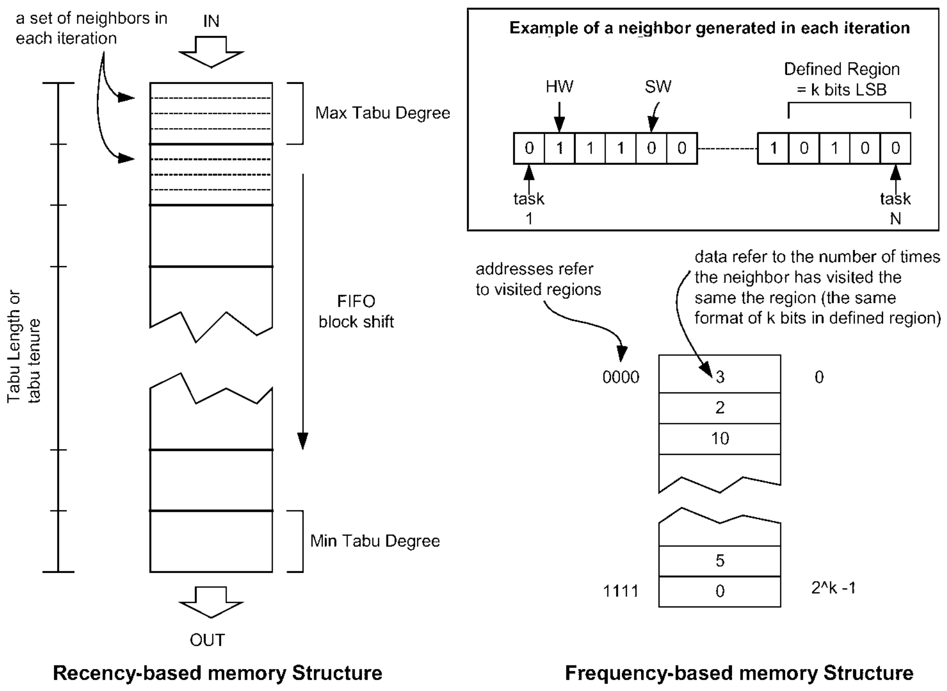

a set of neighbors in each iteration
IN
Max Tabu Degree
Tabu Length or tabu tenure
FIFO block shift
Min Tabu Degree
OUT
Recency-based memory Structure
Example of a neighbor generated in each iteration
HW
SW
Defined Region = k bits LSB
0 1 1 1 0 0
1 0 1 0 0
task 1
task N
addresses refer to visited regions
data refer to the number of times the neighbor has visited the same the region (the same format of k bits in defined region)
0000
3
2
10
5
0
1111
0
2^k -1
Frequency-based memory Structure

Figure 5. The structure of (a) Recency-based memory, (b) Region and frequency-based memory.

divert future search into regions which are rarely visited. We improve upon this idea of penalty [6] in the following way: the region of the best solution in the last K_q iterations is selected. For all neighbors in this region, instead of using Q to compute the penalty factor, a new reward constant $Q' < Q$ is used. The region that earns the *reward* (Q') is called the *promising region*. So neighbors belonging to the promising region in the next K_q iterations have a higher chance to be selected because of the reduced penalty value. Normally, the penalty is only used to *diversify* searching but here we also use it to *intensify* searching on the promising region using Q'. As we shall see later, TS with the penalty reward—called TSPR—provides better solutions compared to those using the standard TS algorithm.

4. Reference Architecture and Timing Model

Timing model, in this context, refers to the algorithm used to estimate processing time taken by all hardware and software tasks which may be executed concurrently. The timing parameters include communication time, execution time on hardware or software, and/or configuration time in the case of reconfigurable hardware.

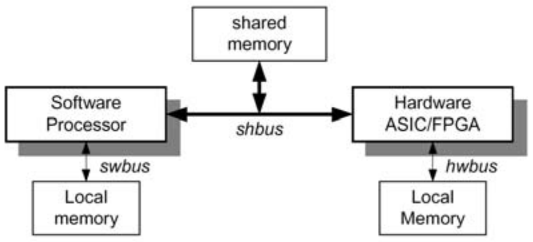


Figure 6. Reference architecture.

4.1. *Reference System Architecture*

The system architecture used here covers important features of the target system such as shared resource conflicts and communication time overhead. In the general model, the system architecture consists of one processor (software representative) and one Application-Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) chip (hardware representative). Both software and hardware have their own local memory and communicate with each other through shared memory (system memory). There is no memory limitation in this model but shared resource conflicts are taken into account. Waiting time is added when a shared resource is engaged by another task, and a waiting task can only be executed after the shared resource is released (non-preemptive). The software processor is a uniprocessing system and it can execute only one task at a time while hardware can execute multiple tasks concurrently.

In case of ASIC, hardware and software can work in parallel and all hardware tasks are bounded to on-chip hardware and there is no reconfiguration time required. In the case of an FPGA, tasks can share hardware resources at different times, but will incur additional configuration time that must be taken into account. In this paper, only an ASIC or an FPGA without reconfiguration is used for implementing hardware tasks.

4.2. *Criteria for Building the Timing Model*

A node in the task graph is placed in a *ready list* only if all its predecessors have been executed. Hence nodes in the current ready list are at the same level of precedence. To find the processing time, we accumulate the maximum execution time between nodes in software and nodes in hardware in each precedence level. The basic criteria for timing calculation are shown in Table 1.

4.3. *Example of DAG and Timing Diagram*

A Directed Acyclic Graph (DAG) is used to represent the process to be partitioned and scheduled. A typical DAG is shown in Figure 7(a). The numbers on the edges represent the

Table 1. The Basic Criteria for Timing Calculation

Software	Hardware
1. Software can handle a single task at a time. Consequently, tasks located in software are sequentially executed.	1. Tasks in the same precedence can operate concurrently but have to be aware of bus conflicts during both read and write periods.
2. Local memory acts like a medium between tasks. Data are read from memory to be executed and then written back in every task.	2. Data have to be buffered in hardware local memory to prevent the loss of data, especially if a subsequent hardware block could be waiting for data from software.
3. To communicate with hardware, software data output is written into a shared memory using the shared bus and waits to be read by hardware.	3. To communicate with software, hardware data output is written into shared memory using the shared bus and wait to read by software.
4. Multiple tasks cannot read (write) data from (to) memory at the same time, even for data located in different area of memory, because all software tasks are executed in sequence.	4. As long as bus contention is avoided, concurrent memory accesses to different memory blocks are allowed.

amount of data dispatched between the nodes. Tasks implemented in hardware are shaded and execution times are shown in square boxes. To calculate communication time, read and write access times on each bus are required. We denote *shrd*, *shwr*, *swrd*, *swwr*, *hwrd*, *hwwr* respectively as the time taken to send or receive unit data for the shared memory, the software, and the hardware. Communication time is obtained by multiplying the numbers on the edges with the bus speed.

Figure 7(b) shows a possible schedule for Figure 7(a), assuming that all bus read/write speeds are unity and that tasks 2, 4, 5 and 6 are implemented in hardware. The level of precedence progresses from top to bottom. The task number is labeled inside the square boxes. Delay may be added due to bus conflict (as at the end of task 2). Our algorithm finds the worst-case time at each precedence level and these are summed together to give the processing time. Since hardware is in general faster and more flexible, software tasks are allocated before hardware tasks so that software tasks have a higher priority in using the shared bus. As an example, consider precedence level 2 in Figure 7(b), which includes tasks 2, 3 and 4. Task 3 is first allocated and a portion of the shared bus access is utilized to write data to shared memory. Task 2 also requires the use of the shared bus, but is forced to wait until task 3 has finished accessing the shared bus. As a result, a delay is imposed on

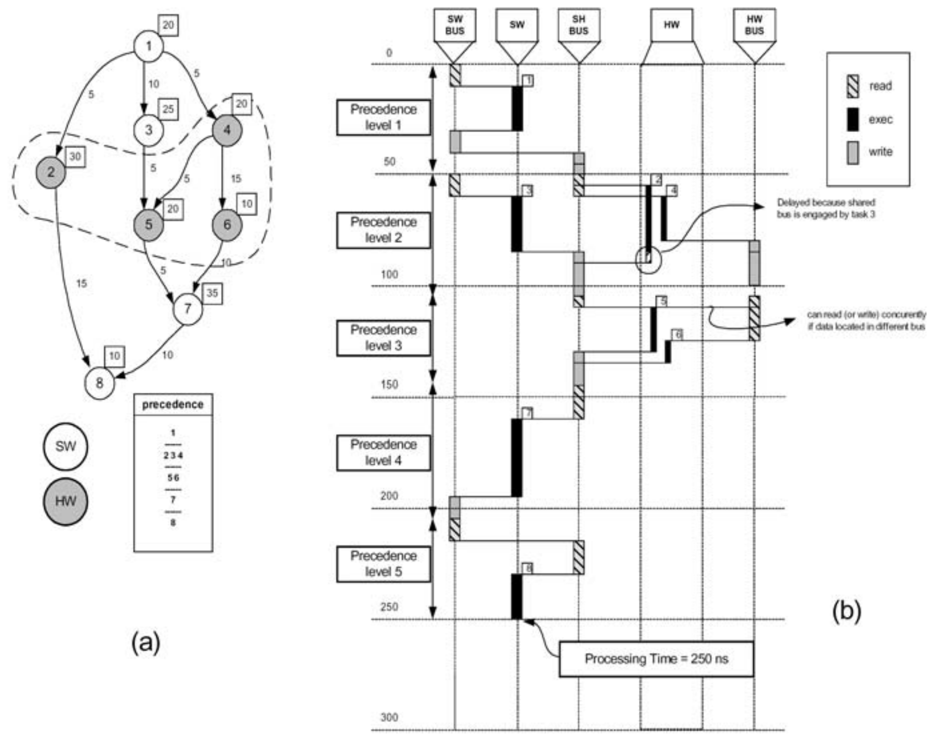


Figure 7. An example of (a) DAG, (b) Task precedence and Process Timing.

task 2.

4.4. Processing Time Calculation Model

Initially, tasks at the same precedence level are grouped together. Software tasks are allocated before hardware tasks; the one that has the minimum use of shared bus will be allocated first. Then hardware tasks are placed based on the *non-increasing first fit algorithm* since, heuristically, packing objects in a rigid storage is effectively done by starting with the placement of the largest one first. Hardware tasks are sorted into a descending sequence before being allocated; the longer execution time the hardware task has, the earlier it is in the sequence. The allocation algorithm is described in Figure 8 in the form of pseudo-code.

Software tasks are allocated first in step 2.2, and then followed by hardware tasks, which are already sorted in decreasing order of execution time. Allocated time of both *shbus* and *hwbus* are updated after each task is placed without any bus conflicts. The subprogram *allocate_hardware*(), responsible for placing hardware tasks, is shown in Figure 9.

The subprogram *allocate_hardware*() gradually increases the time of placing hardware

```
1.  $Vmap[N]$ from each trial

2.  LOOP: while (not reach the exit nodes)
    2.1 Find $RdyList[\,]$ that contain tasks at the same precedence level
    2.2 FOR (all tasks in $RdyList[\,]$)
        $time = allocate_sw()$;
              // place software tasks first, the one that has the
              //minimum use of shared bus will be allocated first.
        $time\_sw = time\_sw + time$;
        Update allocated time on $shbus$
    END FOR
    2.3 $HwListDec[\,] = sort(RdyList[\,])$
              // sort hardware tasks in $RdyList[\,]$ in
              // non-increasing order and keep in $HwListDec[\,]$
    2.4 FOR (all tasks in $HwListDec[\,]$)
        $time = allocate\_hardware()$;
              //place task with the greatest execution time first.
        $time\_hardware = \max(time)$;
        Update allocated time on $shbus$ and $hwbus$
    END FOR
    2.5 $TimePrecedence = \max(time\_sw, time\_hardware)$
    2.6 $ProcessTime = ProcessTime + TimePrecedence$;
    2.7 $EmptyRdyList()$;
```

Figure 8. Processing time calculation algorithm.

tasks from the starting point ($StartRead = 0$) of the current precedence, or the latest point of finishing time in the previous precedence, in order to find the earliest location as possible. The amount of increment, *step*, depends on whether we wish to reduce the search time or to improve the results accuracy. If a writing (reading) period on one or both of *hwbus* and *shbus* is in conflict (the condition in step 8) indicating that data cannot be written after finishing execution, the task will be re-allocated to further locations using the new values of $StartRead = StartRead + (Time - StartWrite)$.

5. Experimental Results and Discussions

In general, the two main measures of the performance of a searching algorithm are the quality of solution and the search time. The quality of solution is generally measured by a cost function mainly based on heuristic criteria. In our case, we use the processing time as a cost function. The best solution is the one that yields the shortest processing time. Our investigation is divided into two parts. In the first part, we compare the results from all three heuristic search algorithms in terms of the quality of solution and the search time. To obtain a fair comparison, we use the standard algorithms, so at this stage penalty reward is not included in TS. Also values of input parameters for each algorithm are

```
1.  Find the length of reading and writing in both *shbus* and *hwbus* of task $i$
2.  $StartRead = 0$;
3.  $Time = StartRead$;
    DO
    Check conflict of placing *shbus* reading time at $Time$
    Check conflict of placing *hwbus* reading time at $Time$
    $Time = Time + step$;
    WHILE (still have conflict)
4.  $StartExec = \max$ (End of reading time on *shbus*,
    End of reading time on *hwbus*)
5.  $StopExec = StartExec + TaskExec$;
6.  $StartWrite = StopExec$;
7.  $Time = StartWrite$;
    DO
    Check conflict of placing *shbus* writing time at $Time$
    Check conflict of placing *hwbus* writing time at $Time$
    $Time = Time$ + step;
    WHILE (still have conflict on both buses)
8.  IF $(Time \neq StartWrite)$
    THEN $StartRead = StartRead + (Time - StartWrite)$;
        GOTO 3.
    END IF
9.  $StopWrite = \max$ (End of writing time on *shbus*,
    End of writing time on *hwbus*)
10. RETURN $(StopWrite)$
```

Figure 9. Subprogram allocate_hardware().

carefully selected by several pre-simulations to get the most promising values. In the second part, we focus on the TS algorithm only. We compare results from the penalty reward strategy in TS with those from using the conventional TS.

In our experiments, randomly generated task graphs with a uniform distribution are used as input, as well as task graphs of commonly encountered structure: in-tree, out-tree, fork-joint, mean value analysis, and FFT (Figure 10).

Table 2 shows the common parameters used in all experiments. Reasonable assumptions are used to produce these parameters. Hardware tasks are usually faster than software tasks, but there are exceptions. Floating point calculations and pointer operations may be faster in software than hardware. As a result the range of software and hardware execution time is shown to overlap. System buses, *hwbus* (hardware local bus), *swbus* (software local bus) and *shbus* (shared bus), are assumed to have different speed. *hwbus* is assumed to be four times faster than *swbus*, and two times faster than *shbus*.

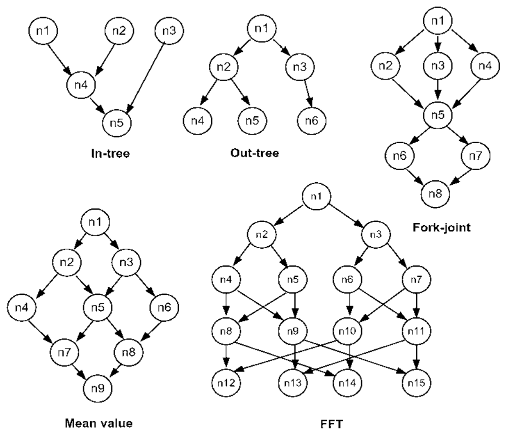


Figure 10. Examples of task graphs of commonly encountered structures.

Table 2. Values Used to Randomly Generate Task Graph

Parameters	Values
Range of software execution time	800–2000 (ns)
Range of hardware execution time	200–1200 (ns)
Range of hardware area	100–400 (unit)
Range of no. of bytes on edges	10–500 (bytes)
Hardware local bus speed (rd/wr)	1/1 (ns/byte)
Software local bus speed (rd/wr)	4/4 (ns/byte)
Shared bus speed (rd/wr)	2/2 (ns/byte)

Read and write cycle times are assumed to be equal.

The number of data (bytes) on the edges is randomly generated. We also assume that the area constraint is around 30% of the area used by a hardware-only solution.

5.1. Comparison between Three Heuristic Searches

Results from Randomly Generated Task Graphs

The algorithms are written in C and run on a 866 MHz Pentium III processor. In the case of GA, the population size is chosen to be between N and $2N$, where N is the number of genes in the chromosome, as suggested in [23]. Population sizes of 40, 50, 100 and the corresponding task graphs with node = 20, node = 50, node = 100 are used respectively. Others parameters are: K_1 = Processing time of the current member in Pop/Pop size, $K_2 = 2$, $\phi = 1$, $P_{cr} = 0.9$, $P_{mu} = 0.01$; for the values K_1 we found that, from several experiments, K_1 usually gives us a good result in various problem sizes when it is adaptively changed by that proportion. For SA, the parameters are set as follows to slowly

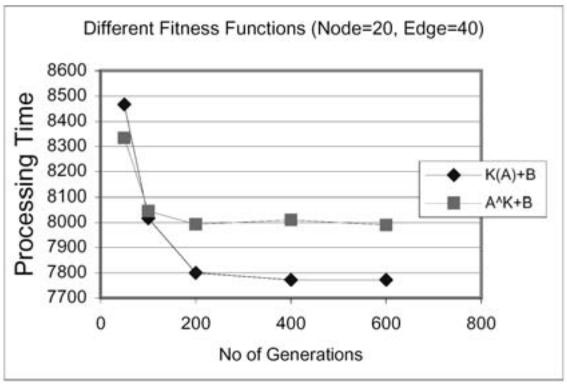


Figure 11. GA with different fitness functions.

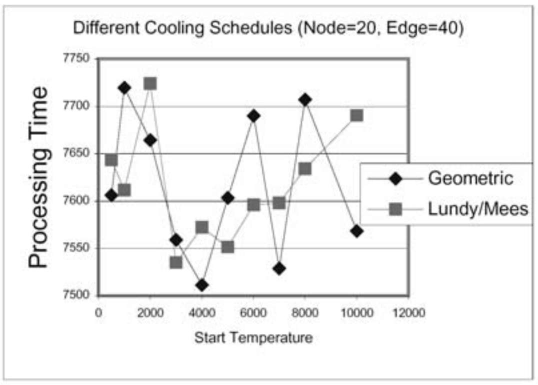


Figure 12. SA with different cooling schedules.

reduce temperature: $\alpha = 0.999$, $\beta = 0.0001$, $\gamma = \beta/100$. In the case of TS, we do not use penalty reward at this stage, so parameters are: neighborhood size = 20, tabu length = 20, $Q = 80$, and 10-bit for the LSB defined region.

We also set the terminating conditions in each algorithm as follows. In GA the program will stop when there are 400 generations. In SA the program will stop when the temperature reaches T_{stop} or when there is no improvement over the last 500 iterations. In TS the program will stop after 400 iterations.

Because SA has two cooling schedules and GA has two different fitness functions, we first determine which cooling schedule and which fitness functions would give better results. The results from GA with different fitness functions and SA with different cooling schedules are shown in Figures 11 and 12 respectively.

From Figure 11, it is clear that the first fitness function $(K(A) + B)$ provides a shorter processing time. Results in Figure 12 are less conclusive. Notice that a higher start temperature may not result in a shorter processing time. Since the Lundy/Mees cooling

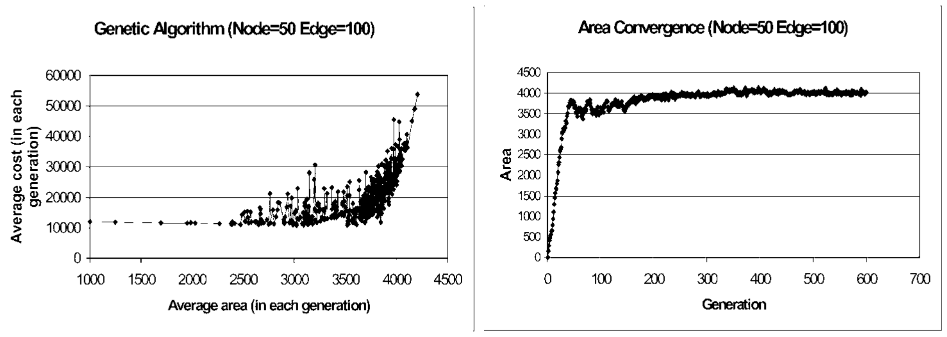


Figure 13. Variation of cost function and area convergence in genetic algorithm.

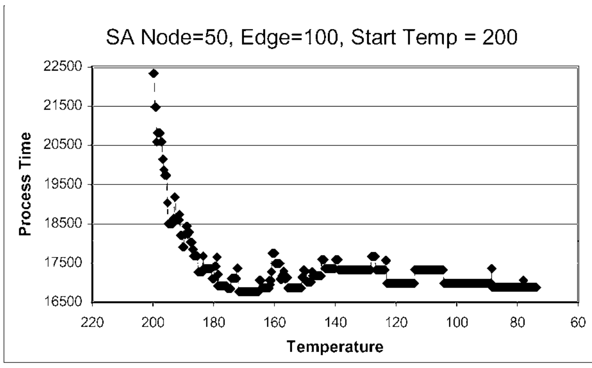


Figure 14. Variation of processing time (cost function) in simulated annealing.

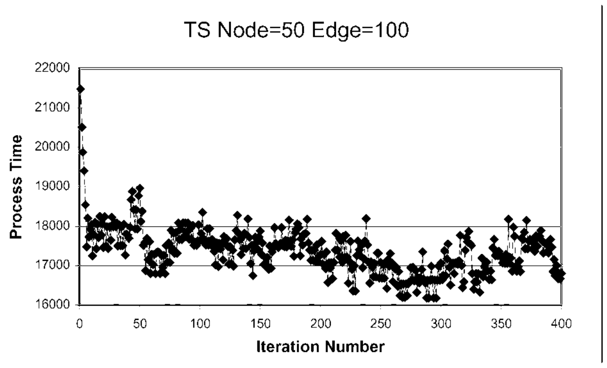


Figure 15. Variation of processing time (cost function) in tabu search.

schedule with re-annealing strategy takes longer to execute than the geometric cooling schedule to reach a solution of similar quality, the geometric cooling schedule is selected for further investigation.

Figures 13, 14, 15 show the convergence characteristic of GA, SA and TS respectively.

Table 3. Processing Time and Search Time of the Best Solution From Each Algorithm

	Genetic Algorithm			Simulated Annealing			Tabu Search		
	Mean Proc. Time	Mean Search Time (sec)	Mean Best of Proc. Time	Mean Proc. Time	Mean Search Time (sec)	Mean Best of Proc. Time	Mean Proc. Time	Mean Search Time (sec)	Mean Best of Proc. Time
Node = 20 Edge = 40	6141.85	4.19	5844.417	5968.52	2.90	5740.615	**5837.64**	**0.96**	**5762.64**
Node = 50 Edge = 100	16391.08	34.57	15863.15	15319.94	19.87	14730.79	**14887.50**	**6.96**	**14656.67**
Node = 100 Edge = 200	33788.93	203.07	32647.71	31169.21	118.13	29996.82	**29943.19**	**44.70**	**29275.14**

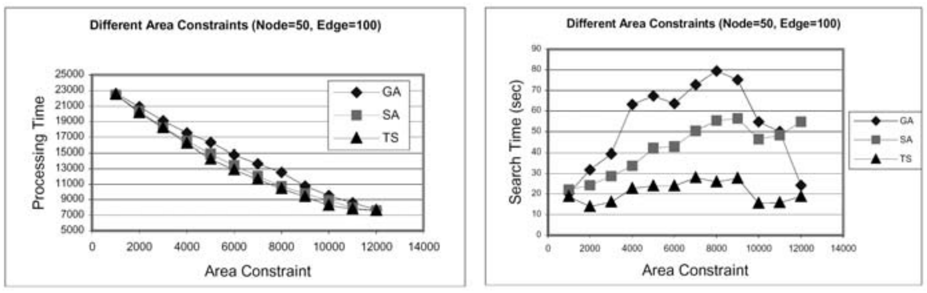


Figure 16. Processing time and search time with a variety of area constraints.

They are all applied to a task graph with 50 nodes and 100 edges. The area constraint is set such that the total area should not exceed 4000 units. Figure 13 shows that the cost for the generation, which contains some members exceeding this constraint, is increased rapidly.

Table 3 compares the three algorithms in terms of processing time and search time in random task graphs. 30 instances of each random graph size are generated, and the results are obtained from 20 runs in each problem instance. SA and TS both produce solutions that are slightly better than GA. TS produces the best solutions, while taking the least amount of computation time. TS also scales better with the complexity of the problem.

RESULTS FROM SOME REALISTIC TASK GRAPHS

In this experiment, rather than using random graphs, we use graph structures that are common in real applications. These structures include out-tree, in-tree, fork-joint, mean value and FFT with a constant number of nodes and edges as shown in Table 4. For each graph type, 20 runs are performed.

Results obtained from all realistic graphs clearly show that TS is superior to SA and GA in terms of both processing time and search time. On average, TS provides the shortest

Table 4. Results From Some Realistic Task Graphs (N.B. 31/30 : Node=31, Edge = 30)

	Genetic Algorithm			Simulated Annealing			Tabu Search		
	Mean Values		Best of Proc. Time	Mean Values		Best of Proc. Time	Mean Values		Best of Proc. Time
	Proc. Time	Search Time (sec)		Proc. Time	Search Time (sec)		Proc. Time	Search Time (sec)	
In-Tree 31/30	6713.5	13.27	6577	6629.9	7.91	6279	**6446.9**	**6.51**	**6327**
Out-Tree 31/30	6388.1	12.09	6181	6452.9	9.15	6137	**6328.4**	**6.95**	**6063**
Fork-Joint 31/50	8967.1	20.11	8373	8838.4	10.56	8483	**8653.3**	**10.48**	**8468**
Mean Value 36/60	9515.0	32.74	9335	9699.5	18.09	9123	**9198.3**	**14.70**	**8922**
FFT 31/46	7888.3	22.25	7681	7926.85	13.83	7523	**7666.5**	**12.54**	**7415**

processing time and the lowest search time compared to the other search algorithms. Also most of the best values of processing time—the lowest ones—belong to solutions getting from TS.

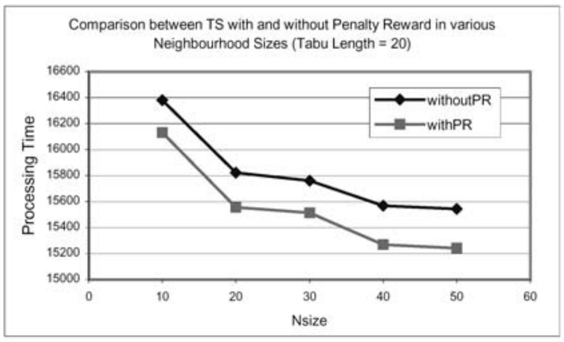
5.2. *TS with Penalty Reward*

We implement TS with the penalty reward modification described earlier with different values of neighborhood size (Nsize). The results are averaged over 20 runs to improve reliability. In experiments graph sizes of 50 nodes and 100 edges, and 100 nodes and 200 edges are used. 400 iterations are performed to select the best solution that offers the minimum processing time. In the case of supporting the penalty reward strategy in TS, we define $Kq = 20$, $Q = 80$, $Q' = 10$.

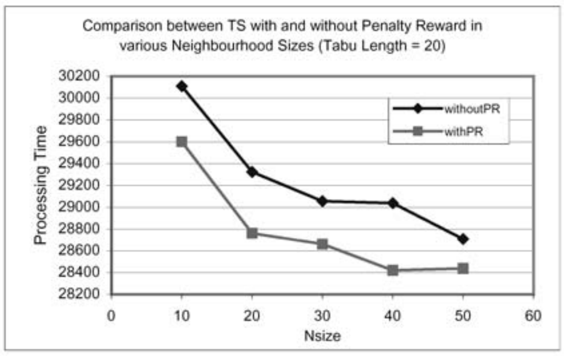
Results in Figure 17 indicate that TS with the penalty reward strategy (PR) *always* produces designs with a shorter processing time than those from TS. Although the improvement is only about 2% (from Figure 17(a)) and 1.6% (from Figure 17(b)), the ability of finding a new solution better than the best from the conventional TS is shown. From additional experiments, we also found that neighborhood size has greater effect on processing time than tabu length.

6. Conclusions and Future Work

An approach that combines partitioning and scheduling has been presented. The results from our heuristic partitioner are used successfully in providing a schedule that



(a) Random Graph of 50 Nodes 100 Edges



(b) Random Graph of 100 Nodes 200 Edges

Figure 17. Comparison between TS with and without penalty reward.

minimizes processing time.

Three popular combinatorial optimization algorithms, namely genetic algorithm, simulated annealing and tabu search, have been compared using a reference system architecture on various sizes and types of problems. In partitioning a design into hardware and software components, we find that tabu search provides higher quality results in a shorter time than both simulated annealing and genetic algorithm. Furthermore genetic algorithm demands more memory to store information about a large number of solutions, while tabu search and simulated annealing are more memory efficient. We have also implemented the *penalty reward* scheme in an intensification strategy for tabu search, which can further improve the quality of solutions.

This work has only examined the partitioning and scheduling problem using simulated inputs in the form of directed acyclic task graphs. While this approach allows investigation of all three algorithms with a variety of problem complexities, our approach needs to be verified on real hardware–software systems. Future work includes the use of realistic benchmarks in evaluating the partitioning and scheduling algorithms on appropriate hardware platforms, such as the SONIC reconfigurable computing system for real-time video processing [24]. We also plan to explore other approaches (such as those in [28], [29], [30]) when refining our work.

Acknowledgements

We thank the four reviewers for their comments and suggestions.

References

1. Oudghiri, H., and B. Kaminska. Global Weighted Scheduling and Allocation Algorithms, In *European Conference on Design Automation*, pp. 491–495.
2. Jinwoo, S., D.-I. Kang, and S. P. Crago. A Communication Scheduling Algorithm for Multi-FPGA Systems, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 299–300.
3. Diessel, O., H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs, Computers and Digital Techniques, *IEE Proceedings*, vol. 147, no. 3, pp. 181–188, May 2000.
4. Gerez, S. H. *Algorithm for VLSI Design Automation*, John Wiley & Sons, England, 1999.
5. Neapolitan, R., and K. Naimipour, *Foundations of Algorithms using C++ Pseudo Code*, 2nd ed., Jones and Bartlett Publishers, London, 1998.
6. Rayward-Smith, V. J., I. H. Osman, C. R. Reeves, and G. D. Smith. *Modern Heuristic Search Methods*, John Wiley and Sons, England 1996.
7. Kalavade, A., and E. A. Lee. A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem, *Proceedings of the Third International Workshop on Hardware/Software Codesign*, 1994, pp. 42–48.
8. Coley, D. A. *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific, 1998.
9. Hidalgo, J. I., and J. Lanchares. Functional Partitioning for Hardware–Software Codesign Using Genetic Algorithms, *EUROMICRO 97. New Frontiers of Information Technology, Proceedings of the 23rd EUROMICRO Conference*, 1997, pp. 631–638.
10. Chatha, K. S., and R. Vemuri. An Iterative Algorithm for Partitioning and Scheduling of Area Constrained Hardware–Software Systems, *IEEE International Workshop on Rapid System Prototyping*, 1999, pp. 134–139.
11. Chatha, K. S., and R. Vemuri. A Tool for Partitioning and Pipelined Scheduling of Hardware–Software Systems, *11th International Symposium on System Synthesis Proceedings*, 1998, pp. 145–151.
12. Bakshi, S., and D. Gajski. Partitioning and Pipelining for Performance-Constrained Hardware/Software System, *IEEE Transaction on Very Large Scale Integration Systems*, vol. 7, no. 4, pp. 419–432, Dec. 1999.
13. Chatha, K. S., and R. Vemuri. An Iterative Algorithm for Hardware–Software Partitioning, Hardware Design Space Exploration and Scheduling, *Journal of Design Automation for Embedded Systems*, vol. 5, pp. 281–293, 2000.
14. Ernst, R., J. Henkel, and T. Benner. Hardware–Software Co-Synthesis for Micro-Controllers, *IEEE Design and Test of Computer*, vol. 10, no. 4, pp. 64–75, 1993.
15. Vahid, F., and T. Le. Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning, *Journal of Design Automation for Embedded Systems*, vol. 2, pp. 237–261, 1997.
16. Harkin, J., T. M. McGinnity, and L. P. Maguire. Partitioning Methodology for Dynamically Reconfigurable Embedded Systems, *Computers and Digital Techniques, IEE Proceedings*, vol. 147, no. 6, pp. 391–396, Nov. 2000

17. Bianco, L., M. Auguin, G. Gogniat, and A. Pegatoquet. A Path Analysis Based Partitioning for Time Constrained Embedded Systems, *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE '98)*, 1998, pp. 85–89.
18. Maestro, J. A., D. Mozos, and H. A. Mecha. Macroscopic Time and Cost Estimation Model Allowing Task Parallelism and Hardware Sharing for the Codesign Partitioning Process, *Proceedings on Design, Automation and Test in Europe*, 1998, pp. 218–225.
19. Axelsson, J. Architecture Synthesis and Partitioning of Real-Time Systems: A Comparison of Three Heuristic Search Strategies, *Proceedings of the Fifth International Workshop on Hardware/Software Codesign, (CODES/CASHE '97)*, 1997, pp. 161–165.
20. Gajski, D. D., F. Vahid, S. Narayau, and J. Gong. *Specification and Design of Embedded System*, Prentice Hall, 1994.
21. Reeves, R., *Modern Heuristic Techniques for Combinatorial Problem*, Blackwell Scientific Publication, UK, 1993.
22. Eles, P., and Z. Peng et al. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search, *Design Automation for Embedded Systems*, vol. 2, pp. 5–32, 1996.
23. Alander, J. T. Optimal Population Size of Genetic Algorithms, *Proceeding in IEEE Computer Society Press*, pp. 65–70, 1992.
24. Haynes, S. D., J. Stone, P. Y. K. Cheung, and W. Luk. Video Image Processing with the Sonic Architecture, *IEEE Computer*, vol. 33, no. 4, pp. 50–57, April 2000.
25. Kirkpatrick, S., C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing, *Science*, vol. 200, no. 4598, pp. 671–680, 1983.
26. Buck, J., S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, *International Journal on Computer Simulation*, vol. 4, pp. 155–182, 1994.
27. Chou, P., and R. Ortega, and G. Borriello The Chinook Hardware/Software Co-Synthesis System, *International Symposium on System Synthesis (ISSS)*, pp. 22–27, 1995.
28. Teich, J., T. Blickle, and L. Thiele, An Evolutionary Approach to System-Level Synthesis, *Proceedings of International Workshop on Codesign (Codes)*, 1997.
29. Dick, R. P., 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.
30. Sih, G. C., and E. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, Feb. 1993.