적합 유전자 알고리즘을 이용한 실시간 코드 스케쥴링

정 태 명[†]

요 약

실시간 시스템에서 시간적 제약의 불이행은 커다란 손실을 가져오며, 이를 위한 동적 스케쥴링은 유연성을 제공하는 대신 스케쥴링 오버헤드와 분석작업의 복잡성으로 인하여 스케쥴성을 예측하기에 어려움이 있다. 반면, 정적 스케쥴링은 수행 중 오버헤드가 없으므로 정확한 시간을 예측할 수 있는 장점이 있다. 따라서 명령어 수준의 정적 스케쥴링과 시간 분석을 통하여 시스템의 시간적 정확도를 보장할 수 있다.

본 논문에서는 확정된 시간 분석을 위하여 before와 after의 시간 제약을 고급 언어에 표현하고 이를 근거로 시간적 분석에 기반을 둔 컴파일러의 명령어 수준의 스케쥴링 알고리즘을 제안하였다. 이 스케쥴링의 특징은 명령어 수준의 스케쥴링을 위한 도메인이 지나치게 과대하므로 향상된 적합 유전자 알고리즘을 적용한 것이다.

Fine Grain Real-Time Code Scheduling Using an Adaptive Genetic Algorithm

Tai-Myoung Chung[†]

ABSTRACT

In hard real-time systems, a timing fault may yield catastrophic results. Dynamic scheduling provides the flexibility to compensate for unexpected events at runtime; however, scheduling overhead at runtime is relatively large, constraining both the accuracy of the timing and the complexity of the scheduling analysis. In contrast, static scheduling need not have any runtime overhead. Thus, it has the potential to guarantee the precise time at which each *instruction implementing a control action* will execute.

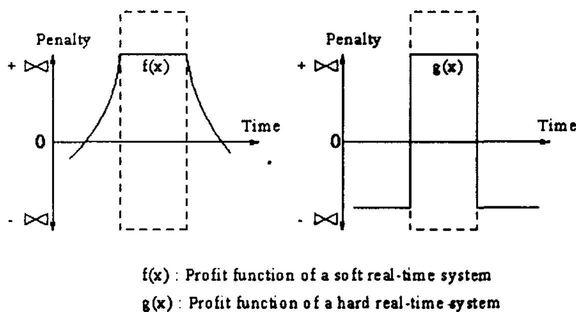
This paper presents a new approach to the problem of analyzing high-level language code, augmented by arbitrary *before* and *after* timing constraints, to provide a valid static schedule. Our technique is based on instruction-level compiler code scheduling and timing analysis, and can ensure the timing of control operations to within a single instruction clock cycle. Because the search space for a valid static schedule is very large, a novel adaptive genetic search algorithm was developed.

1. Introduction

For a real-time system to function correctly, the controlling software must be logically correct and free of timing faults. Clearly, either a flaw in the control program's logic or untimely control action at runtime may cause inappropriate system behavior, resulting in equally severe consequences. A system in which minor timing errors can be tolerated is called a soft real-time system; a hard real-time system can fail catastrophically if even a single operation is performed at the

† 정 회 원: 성균관 대학교 정보공학과 교수
논문접수: 1997년 1월 27일, 심사완료: 1997년 5월 15일

wrong time. Figure 1 depicts the profit functions of soft and hard real-time systems where a negative profit indicates the penalty. While the profit is slowly degraded when a job is not done within the required time range in soft real-time systems, the penalty is not tolerable in hard real-time systems for even a small timing error. The penalty might be the destruction of the real-time environment and even loss of lives. Given that we are concerned primarily with hard real-time control, it is critical that the programming system be able to ensure that all timing constraints will be met no matter what events occur at runtime. Examples of hard real-time systems are command and control systems, flight control systems, and space shuttle avionics systems [1, 5].



(Fig. 1) Profit functions of soft and hard real-time systems.

In order to meet timing constraints, control operations are scheduled at either runtime or compile time, called dynamic or static scheduling respectively [18]. Dynamic scheduling provides flexibility in that the system can adjust its schedule for unpredicted events, but runtime overhead limits the precision of operation timing. This overhead limits dynamic scheduling to relatively coarse grain tasks. The imprecision of dynamic scheduling for a safety critical hard real-time system is potentially dangerous [5]. In contrast, even though static scheduling requires prior knowledge of the timing properties, statically scheduling individual operations (i.e., instruction-level code scheduling) requires no runtime overhead, making it possible to ensure operation timing be accurate to within a single tick of the system's clock.

In safety critical systems, because high precision operation timing is generally necessary, the flexibility of dynamic scheduling should be sacrificed for safety [1]. For example, an emergency brake control process of an automobile should activate an anti-skid function and inflate air-bags in a timely manner; thus, it is important for high precision control operations to be executed accurately. However, such detailed static analysis is not easily achieved, and a variety of modern computer features can blur timing properties (e.g., cache misses, dynamic RAM refresh cycles) over a small range of clock ticks. Static scheduling requires predictable behavior of the hardware systems to support accurate operation level timing analysis.

Instruction scheduling with precedence constraints is NP-hard [10]. Clearly, addition of timing constraints does not simplify the problem, and performing the analysis at the level of individual instructions means that problem size is typically large. Complexity can be reduced by allowing only a restricted class of timing constraints [18], but such restrictions would make the technique impractical for some real-time control systems [6]. To handle arbitrary hard real-time control problems, our analysis must understand fully general timing constraints.

One might hope to build an appropriate analysis and scheduling technique from existing methods, but none of them effectively handles both *before* and *after* constraints between arbitrary instructions in our computational model. The most common scheduling algorithms are priority based: Rate Monotonic Algorithm (RMA) [16], Earliest Deadline First(EDF) [2], etc. However, even with a variety of extensions [4][19], a model with arbitrary *after* timing constraints is not still supported. To our best knowledge, only [11] has a similar view of the scheduling problem and partially solves it by enhancing the schedulability using a structural code motion technique.

In this research, we treat the problem of finding a

valid schedule as a multidimensional search problem [3] for which many approaches are proposed to solve. *Gradient ascent* tends to find only locally optimal solutions [12]. *Simulated annealing* is less likely to miss the globally optimal solution because the range of alternatives that it considers narrows only as it nears the solution, but it is very difficult to determine an appropriate formula by which the search should be narrowed [13, 15]. We have found that a genetic search [14], which is based on the darwinian theory of evolution is effective in finding a globally optimal solution, i.e., a code sequence that satisfies all timing constraints.

Even though the genetic algorithms applied to those applications show potential for the complex problems, the stochastic behavior makes the algorithm inefficient because the search relies too much on random behavior. Hence, in this paper, we propose an adaptive genetic search technique to find a valid schedule more efficiently and more reliably. Adaptive genetic approach implies that the search relies on not only stochastic behavior of the algorithm, but also on deterministic behavior based on program structure.

In Section 2, the timing constraint model with arbitrary *before* and *after* timing constraints is defined, and some notations are introduced. The basic operations and algorithms of adaptive genetic search are described in Section 3 with theoretical analysis. Then, the experimental results are illustrated and analyzed in Section 4. Finally, the conclusion is derived in Section 5 along with current research progress and possible future work.

2. Timing Constraint Model

Consider the trivial control program fragment shown in Figure 2. As written, this code would fail to meet the timing constraints specified in the comments. If we assume that each instruction takes 1 μs to execute, two of the three timing constraints are violated. First, the order of the two actuators being triggered is incorrect; i_3 executes 3 μs before i_5. Second, i_3 reads the second sensor's value 1 μs too late.

γ_1 :	Load r1, sensor1 ;	Get value from memory-mapped sensor 1
γ_2 :	Store act_1 , r1 ;	Send value in r1 to actuator 1
γ_3 :	Load r2, sensor2 ;	Get value from memory-mapped sensor 2
γ_4 :	Add r3, r1, r2 ;	Add the sensor inputs
γ_5 :	Store act_2 , r3 ;	Send result in r3 to actuator 2
θ_1 :	γ_1 must execute no later than 1 μs after γ_3	
θ_2 :	γ_2 must execute at least 1 μs after γ_5	
θ_3 :	γ_3 must execute no later than 1 μs after γ_1	

(Fig. 2) An invalid schedule in a real-time program

In this case, it is possible to reschedule the operations so that all timing constraints are met and correctness of the dependences within the control algorithm are preserved. However, finding a valid sequential order for n instructions involves a search of $n!$ complete schedules - which is very difficult for large n. For the code in Figure 2, we would have to find one of the 120 possible schedules that satisfy all constraints (either i_1, i_3, i_4, i_5, i_2 or i_3, i_1, i_4, i_5, i_2).

Definition : 1 I is defined as a set of instructions to be scheduled, i.e., $I = \{i_1, i_2, i_3, \ldots, i_n\}$ where $i_k \in I$ is k_{th} instruction in the original sequence of n instructions.

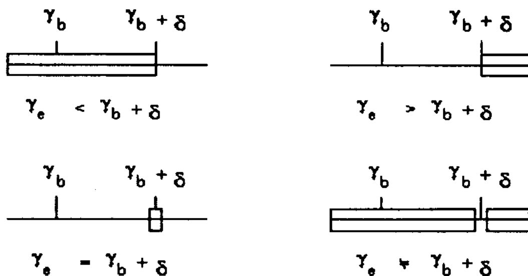
Definition : 2 C is defined as a set of timing constraints associated with the instructions, i.e., $C = \{c_1, c_2, c_3, \ldots, c_m\}$ where m is the number of constraints in the problem. Each component c_k is a tuple denoted as $c_k = \langle i_b, i_e, \eta, \delta \rangle$ where i_b and i_e are the end-point instructions (source and sink), η implies a timing relation between i_b and i_e, and δ is the offset associated with η.

In this model, any timing relations can be expressed as one of the following standard forms which are also

used to express the precedence constraints. For example, a precedence constraint "*x uses y's result*" is converted into a timing constraint "⟨*b*, *e*, *after*, 0⟩" that is a standard form of *after* constraint[1].

1. before constraint: $i_b < i_e + \delta$ (i_b must happen at latest δ after i_e)
2. after constraint: $i_b > i_e + \delta$ (i_b must happen at earliest δ after i_e)
3. concurrent constraint: $i_b = i_e + \delta$ (i_b must happen exactly at δ after i_e)
4. exclusive constraint: $i_b \neq i_e + \delta$ (i_b must NOT happen at δ after i_e)

This conversion makes our scheduling analysis much simpler by encoding ordering constraints in terms of the timing relationships, and by taking into account only those forms of timing constraints. Figure 3 depicts the semantics of the standard forms of the timing constraints.



(Fig. 3) Ranges of i_e to satisfy timing constraints

Definition: 3 A schedule s is defined as an execution sequence of the instructions in I;

i.e., $s_k = i_{k(1)} i_{k(2)} i_{k(3)} \ldots i_{k(n)}$ where $i_{k(x)} \neq i_{k(y)}$ if $x \neq y$, and $i_x \in I$ iff $i_x \in s_k$.

Because there are n instructions and each can logically be placed in any position in the sequence, there are $n!$ schedules in the search space U. Of these, only schedules satisfying all constraints (i.e., $S = \{s_k | s_k \in U$ and $\forall j$, $c_j \in C$ holds for $s_k\}$) are valid solutions to the hard real-time control problem.

3. Scheduling Algorithm

Because the search space is large and has a complex structure, finding a solution by genetic search is appropriate. However, mapping the problem of creating a valid schedule into a genetic search is not straightforward. First, our genetic search requires the definition of:

- An evaluation function that measures how close to valid a particular schedule is.
- A set of basic genetic operations by which each new "generation" of schedules to be considered can be derived from the current population of schedules.

3.1 Evaluation function

Given a schedule s_k and a function τ which returns the time taken to execute a given instruction, the execution time for the block between $\gamma_{k(x)}$ and $\gamma_{k(y)}$ inclusive, denoted as $T(s_k, x, y)$, is calculated as:

$$T(s_k, x, y) = \sum_{j=x}^{y} \tau(\gamma_{k(j)})$$

where x and y are the x_{th} and y_{th} instructions respectively.

We use this time estimate to determine which timing constraints have been violated; the absolute execution times for the schedule is irrelevant. The quality of a schedule is measured by a penalty function, $F(s_k, \Theta)$, which counts the number of timing constraints from the set Θ that are violated by schedule s_k. i.e. $F(s_k, \Theta) \Rightarrow w_s$ where w_s indicates the number of constraints that would cause timing faults when s is executed. We

[1] For serial machines, *concurrent* constraints are unsatisfiable and *exclusive* constraints are tirvially met.

assume that all needed resources are available (e.g., register allocation is performed separately). That is, when $|\Theta| = m$, the range of w is $0 \leq w \leq m$ and $w \in N$. That is, $w_s = 0$ is a necessary condition for s to be a valid schedule. We define the penalty function with unlimited resources as

$$F(s, \Theta) = \sum_{j=1}^{m} f(s, \theta_j)$$

where

$$f(s, \theta_j) = \begin{cases} 1 & \text{if } \theta_j \text{ does not hold} \\ 0 & \text{otherwise} \end{cases}$$

3.2 Basic operation

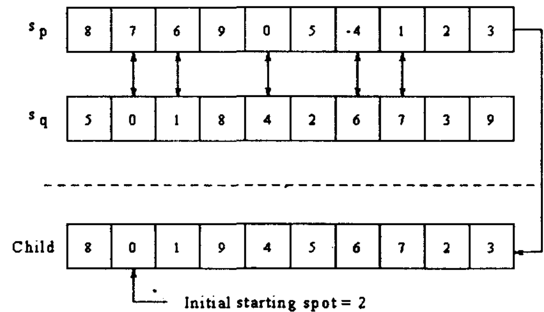
Our search technique combines three basic types of operations for creating a new population of schedules : naive genetic operations, adaptive mechanisms, and reconcilement techniques.

3.2.1 Genetic operations

Natural selection Natural selection stochastically prefers to construct the next generation from the most fit individuals in the current generation. Likewise, our algorithm prefers to preserve schedules that may be close to a valid solution. This is approximated by maintaining some schedules which have relatively low adjusted penalty values from each generation to the next. The adjustment is performed by adding a small stochastic bias to better model the natural process.

Crossover While natural selection preserves quality, it does not enhance quality. Crossover is the process of creating new schedules by *mating* portions of existing schedules. In the process of mating, some schedules generated in this way will combine the good features of both parents, thus surpassing either parent. For each new schedule to be created in this way, two relatively fit schedules are selected from the current population. The "mating" operation is then performed by initially copying one parent schedule and then replacing a subset of its instructions with the corresponding subset from the other parent. The complication is that the new schedule must be a member of the search space U. The following scheme ensures that it is in U.

At the first step, we select a single instruction $i_{p(x)}$ in a copy of the parent schedule s_p that is an endpoint of a violated constraint to increase the probability of improving upon the parent. Then, the corresponding instruction $i_{q(x)}$ from the other parent schedule s_q is substituted for $i_{p(x)}$. If the resulting schedule is in U (i.e., has no duplicate instructions), the process is complete. Otherwise, the instruction $i_{q(x)}$ is located within s_p, and this exchange process is repeated until the resulting schedule is in U. Figure 4 illustrates how this crossover process would proceed given the initial choice of exchanging $i_{p(2)}$. The order of exchange is i_7, i_0, i_4, i_6, i_5.

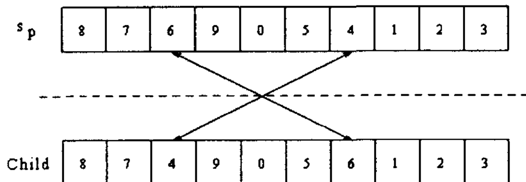


(Fig. 4) Crossover operator applied to pair of schedules

Mutation The problem with the combination of natural selection and crossover is that both tend to reduce genetic variation, which could focus the search on a portion of U that might not contain any solutions. Adding small random perturbations, or mutations, to some schedules prevents this behavior by exchanging the instructions at two randomly selected positions within a schedule. Much like our biasing of the crossover process, one of the selected positions is always an instruction that violates a timing con-

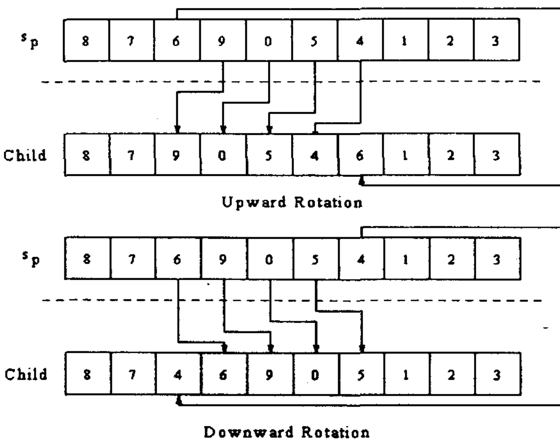
straint.

While conventional mutation increases the variation to the schedules by simply exchanging two random instructions in a schedule, our mutation purposely have one of the exchanged instructions in the parent to be the one causing a timing fault. This modification is applied because mutating the selected instruction probablistically improves the schedule better than mutating the random instruction by moving one of the violated timing constraint. An example is given in Figure 5.



(Fig. 5) Mutation operator applied to a schedule

Rotation Although mutation is effective, the number of mutation operations required to "shift" a portion of the schedule is very large. Thus, rotation is an alternative form of mutation that essentially mutates a schedule in a different dimension. By this operation, the entire subsequence of the schedule between the two selected instructions is rotated as shown in Figure 6.

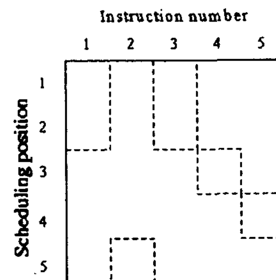


(Fig. 6) Rotation operators applied to a schedule

3.2.2 Adaptive mechanisms

In addition to the (somewhat unusual versions of) standard genetic operations outlined above, the hard real-time scheduling problem has special properties that can greatly improve the success rate. One property of this problem is that, although the fully general problem proposed here cannot be efficiently solved, versions of the same problem with simpler constraints can be solved efficiently. The other property is that, in most cases, if only a few constraints are violated, it is common that a solution can be found by permuting *only* the instructions involved in the violated constraints.

Position pruning Although verifying timing constraints requires that the complete schedule be evaluated, it is inexpensive to determine that many potential instruction placements are invalid. This is done by reducing the timing constraints to simple precedence relations, and then translating these relations into possibly valid ranges of schedule positions for each instruction. Suppose that the precedence information shows that i_x has α ancestors and β descendants among n instructions. Clearly, the instruction i_x must be in a schedule position between $\alpha-1$ and $n-\beta$ invalid. If i_x is placed outside of this range, the schedule is guaranteed to be *invalid*.



(Fig. 7) Search space for example 1 with pruning technique

In our algorithm, this position pruning technique is implicitly employed to all the genetic operations except the crossover in order to avoid the search space not containing any valid schedule. For example, the position where a selected instruction is placed is chosen among the positions not pruned by this position pruning. Figure 7 shows that how much of the search space is reduced for the instruction sequence given in example 2 by position pruning. With a simple calculation, we find that the search space is reduced from 120 to 2 schedules. In fact, in this particular case, both remaining schedules are valid solutions because each timing constraint can be accurately simplified into a precedence constraint.

Insertion stretch Scheduling with only *before* constraints, usually called "deadlines", has been studied by many researchers. When *after* constraints are added, the problem becomes much more complex due to interactions between *before* and *after* constraints. For example, a code reorganization to satisfy a *before* constraint may destroy a previously satisfied *after* constraint, and vice versa.

However, there are often independent instructions within the schedule: instructions that can be freely moved without affecting other constraints. Insertion stretch simply moves these instructions to just before each instruction that fails an *after* constraint. A similar code motion has been used in [11] to satisfy *before* constraints.

Exhaustive search Exhaustive search always yields an optimal solution, but the computational complexity explodes as the schedule length increases. However, if the only invalid timing constraints occur within a short sequence of instructions that do not impact any timing constraints outside this sequence, exhaustive search *within that sequence* may be an effective method for finding a solution.

Naive exhaustive search requires O(n!) time complexity to evaluate all possible schedules. For example, a 15 instruction sequence could require testing all 15! or 2,004,310,016 schedules. However, using an exhaustive search modified by pruning techniques (based on the technique used by [17] to schedule instructions for multiple-pipeline processors), scheduling 15-instruction sequences is practical. Thus, this technique is applied whenever the sequence size drops below 16 instructions.

3.2.3 Reconcilement techniques

Adaptive genetic search is most effective when the search is a mix of stochastic and deterministic behavior, balancing ability to refocus the search with efficiency in searching within the current focus. Reconcilement techniques attempt to maintain this balance. Too much determinism is prevented by perturbation; too much randomness is avoided by discrimination.

Perturbation In the genetic algorithm, one of most critical performance improvements is to avoid repetitive generation and evaluation because it is quite possible that the random behavior of the algorithm repeatedly produces the same schedules. The adaptive mechanisms also may lead the search toward the same space. Thus, it is necessary to detect and eliminate repeated schedules.

Perfectly detecting repeat schedules would require a time complexity of $O((n-1)!)$ at each generation. Instead, we use a novel algorithm to identify repeated schedules by exploiting a hashing technique and prime number multiplication. In this technique, the hash index of a schedule is obtained by summing the results of the multiplication of the instruction id by the corresponding prime number, Φ. Namely, the hashing index for s_k can be computed as:

$$H(s_k) = \sum_{j=1}^{n} \Phi(j) \times i_{k(j)}$$

where $\Phi(j)$ is j_{th} prime number and $i_{k(j)}$ is the instruction identifier for j_{th} instruction in s_k as an integer.

By this algorithm, hash indices are rarely equivalent unless the schedules are identical. If the hashing entry for the index already exists, the schedule is perturbed[2]. The hashing index calculation does not have to be done separately because the calculation can be a part of the evaluation process.

When a schedule consists of large number of instructions, the schedules can be simply partitioned into subschedules and individually compared to reduce both space and time complexity. Let s_p and s_q are partitioned into $s_p^1, s_p^2, \cdots, s_p^k$, and $s_q^1, s_q^2, \cdots, s_q^k$ respectively. Then, s_q is perturbed when $H(s_p) = H(s_q)$ implying that $(s_p^1 = s_q^1) \cap (s_p^2 = s_q^2) \cap \cdots \cap (s_p^k = s_q^k)$.

Discrimination If a high penalty is associated with a schedule, this suggests that the search space has become too random, and the schedule may not be in the right direction. In this case, the discriminating decision is made by the algorithm to reduce the randomness. When a schedule associated with high penalty is generated by one of the operations, the algorithm simply discards the schedule and adopts the parent schedule instead.

In this paper, we propose a more sophisticated detection mechanism to find the schedules to be discarded based on the concept of *simple constraints*. The simple constraints have the offset value of zero, equivalent to the precedence constraints. Indeed, the simple constraints are an improper superset of the originally supplied precedence constraints. When a schedule generated by a basic operation has a certain number of violated simple constraints, say σ, the discrimination determines to discard the schedule. The value of σ should be tuned for optimal performance out of discrimination operation because it may waste the time to compute the schedule when it eliminates right path with σ value fixed too low. This operation can be disabled when σ is set to be infinite.

3.3 Algorithm description and analysis

Crossover algorithm As explained earlier, the transitive closure set enforces that new schedules be in the search space. In the crossover algorithm, Step 2 through Step 7 of the crossover algorithm compute transitive closure sets, γ_1 and γ_2 to exchange without missing or duplicating any particular instructions. It is achieved by transitively finding the union set of instructions in corresponding positions of s_p and s_q.

The modification of traditional crossover algorithm is made in Step 2 to favor deterministic behavior by selecting $i_{p(x)}$ such that the instruction is either $i_b(\tilde{c}_k)$ or $i_e(\tilde{c}_k)$. $\tilde{c}_k$ denotes one of the constraints that would be violated. From the second iteration, $i_{p(x)}$ is the one identical to $i_{q(x)}$ in the previous iteration as appeared in Step 6.

The $\oplus$ operator in Step 8 inserts γ_2 into the corresponding positions of γ_2 in s_p. In fact, the positions of the instruction in γ_1 and γ_2 are identical.

	Crossover Algorithm
Input:	$s_p = i_{p(0)} i_{p(1)} \cdots i_{p(n)}$, $s_q = i_{q(0)} i_{q(1)} \cdots i_{q(n)}$ and $C = \{c_1, c_2, \cdots, c_m\}$
Output:	$s_{\tilde{p}} = i_{\tilde{p}(0)} i_{\tilde{p}(1)} \cdots i_{\tilde{p}(n)}$
Procedure:	
Step 1:	initialize transitive closure sets: $\gamma_0 = 0$, $\gamma_1 = 0$
Step 2:	select $i_{p(x)}$ from s_p
Step 3:	set $\gamma_0 = \gamma_0 \cup i_{p(x)}$
Step 4:	set $\gamma_1 = \gamma_1 \cup i_{q(x)}$
Step 5:	If ($\gamma_0 = \gamma_1$), then goto Step 8
Step 6:	x = index($i_{q(x)}$ in s_p)
Step 7:	goto Step 3
Step 8:	$s_{\tilde{p}} = s_p \oplus \gamma_2$
Step 9:	return $s_{\tilde{p}}$

Mutation and rotation algorithm The mutation algorithm and rotation algorithm similarly performs in-

[2] There is a low probability the schedules being different, but comparing multiple schedules for the small probability make the algorithm less efficient.

struction exchanges to generate probablistically improved schedule. The difference is that the rotation shifts all the instructions to upward or downward while mutation simply swap two instructions. Note that Step 1 of both algorithms select one of the target instructions, $i_{p(x)}$ in this algorithm, to be the one violating a constraint. Also, random number generator selects the other instruction, $i_{p(y)}$, to be in the valid range for $i_{p(y)}$ in both mutation and rotation algorithms.

The operator $\oplus$ used in Step 2 of mutation algorithm and Step 3 of rotation algorithm is identical to the one used in Step 8 of crossover algorithm. Hence, the corresponding positions in s_p are updated in such way that the resulting schedule remains in search space.

	Mutate Algorithm
Input:	$s_p = i_{p(0)} i_{p(1)} \cdots i_{p(n)}$ and $C = \{c_1, c_2, \cdots, c_m\}$
Output:	$s_{\bar{p}} = i_{\bar{p}(0)} i_{\bar{p}(1)} \cdots i_{\bar{p}(n)}$
Procedure:	
Step 1:	select $i_{p(x)}, i_{p(y)}$ from s_p
Step 2:	$s_{\bar{p}} = s_p \oplus Swap(i_{p(x)}, i_{p(y)})$
Step 3:	return $s_{\bar{p}}$

	Upward Rotation Algorithm
Input:	$s_p = i_{p(0)} i_{p(1)} \cdots i_{p(n)}$ and $C = \{c_1, c_2, \cdots, c_m\}$
Output:	$s_{\bar{p}} = i_{\bar{p}(0)} i_{\bar{p}(1)} \cdots i_{\bar{p}(n)}$
Procedure:	
Step 1:	select $i_{p(x)}, i_{p(y)}$ from s_p
Step 2:	$i_{temp} = i_{p(x)}$;
Step 3:	From $j = i_{p(x+1)}$ to $i_{p(y)}$ $s_{\bar{p}} = s_p \oplus Swap(i_{p(j-1)}, i_{p(j)})$
Step 4:	$i_{p(y)} = i_{temp}$
Step 5:	return $s_{\bar{p}}$

Main: Adaptive genetic algorithm Step 1 of the main algorithm determines the parametric values of the algorithm for more efficient execution. Because this algorithm is based on random heuristics, the parametric values obtained from the experiments make more sense. The parameters are population size (p), number of children by natural selection (m), by mates (k), and by the other basic operations (p − m − k). In particular, the population size determines the number of hills that are simultaneously searched. The detailed discussion of the parametric values are given in the next section.

In Step 2, the main algorithm generates initial population G_0. It is very important to generate G_0 that contains the schedules close to the valid space because the search starts at the initial schedules in G_0 and adjusts the schedule toward the valid ones. One of the ways creating G_0 is to perturbing a schedule that conventional compilers produce. The schedule usually satisfies all the precedence constraints Hence, G_0 is generated by perturbing the input sequence s_{in}, produced by conventional compilation techniques. However, preserving the simple constraints, i.e. $\delta = 0$ while we randomly perturb s_{in} is costly because only the schedules satisfying $F(s, C^*) = 0$ are selected where C^* denotes the set of simple constraints. However, G_0 can be easily produced if the schedules with $\delta > 0$ are allowed. In fact, it not only reduces the execution time, but also offers more randomness to the algorithm.

An alternative to generate G_0 is applying topological sort on s_{in}. In the case that the topological sort does not produce enough schedules for G_0, the schedules that have already been generated are simply duplicated. This scheme guarantees that all the schedules have property of $F(s, C^*) = 0$.

After the natural selection and crossover are performed in Step 6 and Step 7 of the main algorithm, the rest of the offsprings are generated by applying one of the minor operations. In the Step 8, insertion stretch, exhaust, rotate or mutate are selected based on the type of constraints or length of blocks with violated constraints. The pseudo code for stretch and exhaust are not given because they are trivial. In Step 9, identical schedules are found by comparing hashing index, and perturbed if they are already considered.

This algorithm always finds a solution if it exists and infinite generations are tried. However, the algorithm can be improved by restarting the process when it exceeds a certain number of generations called the limit *algorithm threshold.* The program terminates when one of following conditions is met.

1. When a valid schedule is found. i.e., $\exists s$ where $F(s, C)=0$ (Step 4).
2. When new offsprings is not generated. i.e., $G_i = G_{i+1}$, namely, $\forall s_j \in G_i$, $s_j \in G_{i+1}$ (Step 9).
3. When a valid schedule is not found after the descendant threshold specified by user. i.e., $\forall s_i \in G_{max}$, $F(s, C) < 0$ (Step 9).

Among the three termination criteria, the second and third conditions are added to the algorithm appeared in the previous section for the case of the unsuccessful termination. In those cases, an execution of the algorithm is repeated because the randomized search paths based on the time function are usually different each time.

Main: Adaptive Genetic Algorithm

Inputs: $s_{in} = i_{in(0)} i_{in(1)} \cdots i_{in(n)}$ and
$C = \{c_1, c_2, \cdots, c_m\}$

Outputs: Valid schedule: $s_{out} = i_{out(0)} i_{out(1)} \cdots i_{out(n)}$
or NIL if fails.

Procedure:

Step 1: Set variables $\mapsto$
j = current generation, p=population size
m = # of children by natural selection
k = # of children by mate

Step 2: Generate $G_j = \{s_{j(1)}, s_{j(2)}, \cdots s_{j(p)}\}$ by perturbin s_{in}

Step 3: $\forall s_i \in G_j$, compute $w = F(s, C)$.

Step 4: If ($\exists s_i \in G_j$ such that $w=0$), then return s_i and terminate

Step 5: Set $G_{j+1} = 0$

Step 6: Select m best schedules from G_j, say $\aleph_j$.
Then, set $G_{i+1} = G_{i+1} \vee \aleph_i$

Step 7: Apply crossover to generate k offsprings (k times):
step 7.1: Randomly select $s_x, s_y \in G_{j+1}$
step 7.2: set $G_{i+1} = G_{i+1} \vee Crossover(s_x, s_y)$

Step 8: Apply other operations to generate the rest((p-m-k) times):
If($\eta(c_k)=after$, $|i_e(c_k)-i_b(c_k)| < \delta(c_k)$),
$G_{j+1} = G_{j+1} \vee Stretch(c_k)$
else if ($\forall \tilde{c}_k$, $|i_e(c_k)-i_b(c_k)| = \varepsilon$),
$G_{j+1} = G_{j+1} \vee Exhaust(i_b(c_k), i_e(c_k))$
otherwise, $G_{i+1} = G_{i+1} \vee Mutate(s_i)$

Step 9: If($\exists s_x$, $\exists s_y$, $s_x = s_y$), $s_y = Perturb(s_y)$

Step 10: If ($G_i = G_{i+1}$ or $i >$ algorithm threshold) Terminate

Step 11: Set $j = j + 1$ and go to Step 3.

---

4. Experiments

4.1 Experimental environment

Our experiments were performed on IBM RISC System/6000, and Sun Sparc10 workstations. In this section, we present the results obtained from running the adaptive genetic algorithm on Sun Sparc10 workstations, because our goal is not comparing the performance of the platforms, but identifying performance factors in the algorithm. In fact, the results from IBM RS/6000 workstation are proportionally scaled to the ones from Sun Sparc10 workstations.

The algorithm is implemented in the C language under the UNIX operating system. The grammar for the real-time control system is parsed using PCCTS (the Purdue Compiler Construction Tool Set) to generate an intermediate form representing an instruction sequence and the set of constraints. Then, the algorithm presented in the previous section is applied to manipulate the intermediate form. Thus, the initial input data file is generated from the real-time language that is under development as a part of CHaRTS project (Compiler for Hard Real-time Systems). The CHaRTS project includes design of language construct phase [7] and code scheduling phase for complete compilation of real-time control programs.

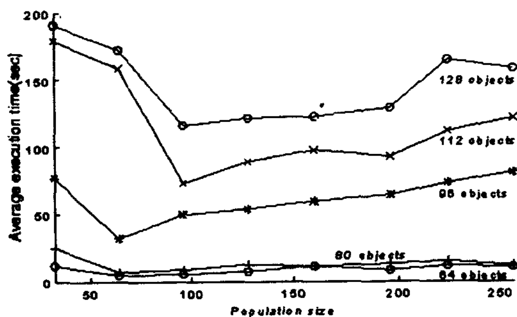
4.2 Results and analysis

In these experiments, we focus on determining the factors that directly influence on the compile time as

well as the success rates. That is, the measurements in which we are interested are the **search time** for a valid schedule and the **success rate**; hence, our results are given in two categories. One: average search time for 30 successful executions of the algorithm. Two: success rate that indicates how many times the algorithm successfully terminates with a valid schedule. The results of the success rates along with the average execution time indicates that this algorithm has potential to be employed to code scheduling for real-time systems.

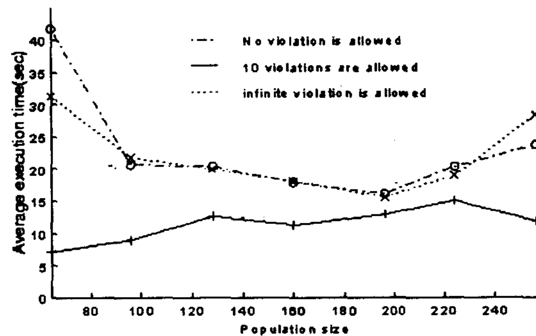
A critical performance factor is the population size that has trade-off between execution time to generate the next population and the probability of having variety of sequences on the paths in a generation. If the population size is too small, then the randomness is drastically diminished while the execution time for a generation to next is reduced. Hence, the first experiment was focus on the performance of the algorithm for different values of population size. In this experiment, the programs consist of 64, 80, 96, 112 or 128 instructions, and population size varies from 32 to 256 schedules. At this stage, small problems are solved quickly, but scheduling problems involving more than 200 instructions are not yet practical. To handle larger problems, we are developing a hierarchical decomposition scheme.

For this experiment, we ran the algorithm until 30 executions found valid schedules when the algorithm



(Fig. 8) Execution times for various size of problems

threshold is set to 4,096 generations. Then, the average execution time of the successful runs was computed. The results in Figure 8 shows that there exists an optimal population size that is proportional to the problem size. The larger population size is not always better because the execution time for one generation to next is greater for it while the randomness is not much improved after a certain population size. However, the graph does not correctly reveals the success rates obtained for the executions not finding a valid schedule until the algorithm threshold. Indeed, the success rates are all 1 when the population size is greater than 64. The success rate for population size of 32 are linearly decreased from 0.8 to 0.2 as the number of instructions are incresed from 64 to 128.

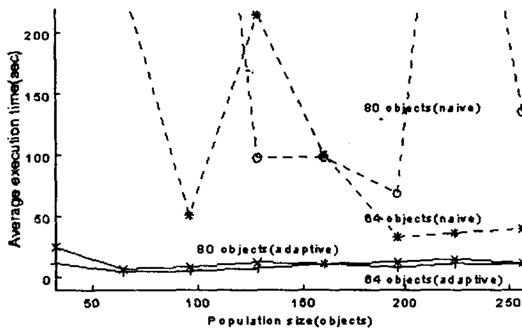


(Fig. 9) Execution times for different degree of discriminatior

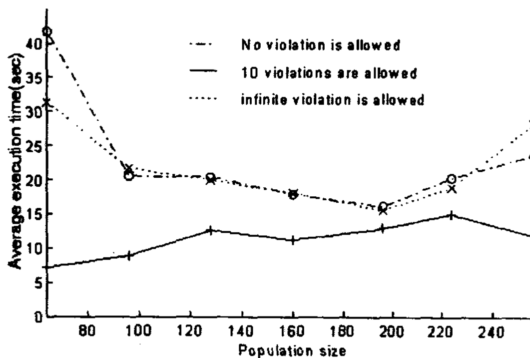
The initial population is another critical performance factor. It is generated from an input sequence that is provided by the compiler's front-end. The preliminary implementation of CHaRTS provides the input sequence which is, in turn, perturbed to generate the initial population. Note that the initial population is a subset of the sequences that would be generated by topological sorts if $\sigma=0$. The results with different σ values are also obtained. The initial population is manipulated by adjusting the value of σ. Figure 9 illustrate the results for different σ values, and it shows that the discrimination operation is optimal when it is in a certain range. One interesting result from this

experiments is that if a valid schedule is not found in small number of generations, then the probability of find it in larger number of generations.

Further, we were interested in learning the improvement by the adaptive algorithm compared with a naive genetic algorithm. Hance, an experiment was performed to schedule instructions under the same environment for both algorithms, scheduling 64 and 80 instructions. The compared execution times of naive genetic algorithm and adaptive genetic algorithm are depicted in Figure 10. The result illustrates the naive search works reasonably, but the adaptive mechanisms improves not only the search time but also the success rate. It implies that the adaptive technique is more reliable as well as more efficient than naive genetic algorithm for the code scheduling problem.



(Fig. 10) Execution times of naive and adaptive genetic algorithms



(Fig. 11) Execution times with various SCM ratio

SCM ratio, based on the number of offsprings generated by natural selection, crossover, and other basic operations, is another significant performance factor. We varied the ratio to 1:1:2, 2:1:1, 1:2:1 to see what operations are more effectively generate offsprings that are close to the valid space. Figure 11 depicts the behavior of the algorithm with different SCM rato. For this experiment, we ran the algorithm for 80 instructions with various population size. In this experiment, the result shows that the SCM ratio is not a critical factor in this algorithm, indicating that the basic operations equally contribute to the algorithm except the case that the number of offsprings by natural selection is too small.

5. Conclusion

In this paper, we have presented a method for statically scheduling hard real-time programs at the instruction level. Unique to our model is fully general timing constraints that support timing relations between arbitrary instructions. The analysis of the model is simplified by encoding all constraints into a form of timing constraints, then an adaptive genetic search technique is applied to find a valid schedule.

Realizing that this very general formulation results in a very difficult instruction scheduling task, we have proposed, implemented, and evaluated an adaptive genetic search code scheduler. This scheduler is capable of scheduling instruction sequences containing over 120 instructions (or tasks) augmented by very complex *before* and *after* timing constraints. However, techniques like hierarchical decomposition need to be developed so that much larger codes can be scheduled efficiently. Of course, using the existing scheme to schedule instructions for soft real-time systems would be effective for much larger programs.

This paper only focuses on code scheduling for serial machines, but various parallel architectures can be exploited to improve schedulability. Thus, the next step is to develop scheduling techniques that can

make use of parallel control computers. We hope to target static scheduling to a collection of RISC microprocessors connected by a device with static communication timing proerties; the communication hardware would be based on barrier synchronization, much like PAPERS (Purdue's Adaptor for Parallel Execution and Rapid Synchronization) [9], which links clusters of workstations.

It is also significant that the instruction scheduling techniques developed here represent several significant advances over the compiler code scheduling technology that was drawn upon for this project. Consequently, the adaptive genetic search instruction scheduling presented here should be applicable to instruction scheduling for applications like fine-grain parallel code scheduling (e.g., for VLIW or pipelined computers) [8].

References

[1] Anon, "Putting a stop to vehicles slipping and sliding", Design Engineering, page 24, July. 1994.

[2] K. R. Baker, 'Introduction to Sequencing and Scheduling', John Wiley and Sons, New York, NY, 1974.

[3] D. Brand, "Hill climbing with reduced search space", In IEEE International Conference on Computer-Aided Design, pp.294-297, Nov. 1988.

[4] P. Bratley, M. Florian, and P. Robillard, "Scheduling with earliest start and due date constraints", Navals Research Logistics Quarterly, Vol. 22, No.1, pp.165-173, Dec. 1975.

[5] A. Burns, "Scheduling Hard Real-Time Systems: A Review", Software Engineering Journal, Vol.6, No.3, pp.116-128, May. 1991.

[6] T. M. Chung, "CHaRTS: Compiler for Hard Real-time Systems", Ph.D. Thesis, Purdue University, July. 1995.

[7] T. M. Chung and H. G. Dietz, "Language constructs and transformation for hard real-time programs with fine-grained timing constraints", ACM SIGPLAN Notices, Vol.30, No.11, pp. 41-49, Nov. 1995.

[8] T. M. Chung and D. J. Hwang, "Fine-grain real-time code scheduling for vliw architecture", Journal of Electrical Engineering and Information Science, Vol.1, No.1, pp.118-128, Mar. 1996.

[9] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox, "PAPERS: Purdue's Adapter for Parallel Execution and Rapid Synchronization", Technical Report TR-EE 94-11, Purdue University, Mar. 1994.

[10] M. R. Garey and D. J. Johnson, 'Computers and Intractability, a Guide to Theory of NP-Completeness', W. H. Freeman Company, San Francisco, CA, 1979.

[11] R. Gerber and S. Hong, "Compiling real-time programs with timing constraint refinement and structural code motion", IEEE Transactions on Software Engineering, Vol.21, No.7, pp.389-404, May. 1995.

[12] H. Guo and S. B. Gelfand, "Analysis of gradient descent learning algorithms for multilayer feed-forward neural networks", IEEE Transactions on Circuits and Systems, 38(8), pp.883-894, Aug. 1991.

[13] S. M. Hart and C. S. Chen, "Simulated annealing and the mapping problem: a computational study", Computers Operations Research, Vol.21, No.4, pp.455-461, Apr. 1994.

[14] J. Holland, "Adaptation in Natural and Artificial Systems", Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1975.

[15] S. Kirkpatrik, C. Gelatt Jr., and M. Vecchi, "Optimization by Simulated Annealing", Science, 220(4598), pp.671-680, May. 1983.

[16] C. L. Liu and J. Layland, "Scheduling algorithms for multiprocessing in a hard real-time environments", Journal of ACM, Vol.20, No.1, pp.46-61, Jan. 1973.

[17] A. Nisar and H. G. Dietz, "Optimal code scheduling for multi-pipeline processors", In Proc of

1990 Int'l Conf. on Parallel Processing, volume II, pp.61-64, St, Charles, IL, Aug. 1990.

[18] J. A. Stankovic and K. Ramamritham, "Hard Real-Time Systems Tutorial", IEEE Computer Society Press, 1991.

[19] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines precedence, and exclusive relations", Software Engineering, pp. 350-359, Mar. 1990.

정 태 명

1981년 2월 연세대학교 전기공학과 학사
1984년 5월 일리노이 주립대학 전산학과 학사
1988년 12월 일리노이 주립대학 컴퓨터공학과 석사
1995년 8월 퍼듀대학 컴퓨터공학과 박사
1985년~1987년 Waldner & Co., Systems Engineer
1987년~1990년 Bolt Beranek and Newman, Staff Scientist
1995년~현재 성균관대학교 정보공학과 조교수
관심분야: 실시간 시스템, 컴파일러, 네트워크 관리