Adaptive Genetic Algorithm: Scheduling Hard Real-time Control Programs with Arbitrary Timing Constraints *

Tai M. Chung and Hank G. Dietz
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285

Abstract

In hard real-time systems, a timing fault may lead the environment to catastrophe. Thus, a real-time program must be executed in accordance with the timing constraints specified in the control program. Dynamic scheduling approach which has been dominant for real-time systems are unacceptable due to run-time scheduling overhead and unpredictable risks. In contrast, a static approach takes advantage of thorough search for scheduling, no scheduling overhead, and guaranteed execution of compiled program.

However, finding an optimal schedule for real-time tasks with precedence is known as NP-hard problem, demanding a heuristic approach. In this paper, we propose an adaptive genetic approach to find a valid schedule. Adaptive genetic approach implies that the algorithm relies on not only stochastic behavior of the search but also deterministic behavior based on program structure. This paper includes new computation model with arbitrary *before* and *after* constraints, description of modified genetic algorithm, and the experimental results.

This work is an endeavor to build a scheduler that is a part of the compiler for hard real-time systems (CHaRTS). CHaRTS accepts a hard real-time program and a system configuration file, then generates a sequence of executable code for target machine.

*This work is supported by the Office of Naval Research (ONR) under grant number N00014-91-J-4013.

1 Introduction

In general, a real-time system consists of a controlling subsystem and controlled entities. A controlling subsystem is a computer that executes a program to receive input from its environment, compute appropriate actions, and then command the controlled entities appropriately. The controlled entities can be any of a broad range of systems with mechanical behavior, any device from a simple blender to a complex robot [SR91] [BHJ+82].

For a real-time system to function correctly, the controlling subsystem must be logically correct and free of timing faults. A system in which minor timing errors can be tolerated is called a soft real-time system; a hard real-time system fails catastrophically if even a single operation is performed at the wrong time. 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. Only by static analysis of the computations involved in the controlling subsystem can such a guarantee be made.

While dynamic scheduling implies runtime overhead that limits the precision of operation timing, statically scheduling individual operations (i.e., instruction-level code scheduling) makes it possible to ensure operation timing is accurate to within a single tick of the system's clock. However, such a detailed static analysis is not easily achieved, and a variety of modern computer features can blur timing properties (e.g., dynamic RAM refresh cycles) over a small range of clock ticks.

Our goal in this paper is to show that static instruction-level analysis is possible, and that this analysis can directly drive compile-time code scheduling. To simplify the discussion, we assume that the target processor has the property that each instruction executes in a precisely known number of clock cycles (for example, the TMS320C40 can be configured this way [Inc88]).

Instruction scheduling with precedence constraints is NP-hard [GJ79]. 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 [SR91] [GPS92], but such restrictions would make the technique useless for some real-time control systems. 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 is effectively usable. The most common algorithms are priority based: Rate Monotonic Algorithm (RMA) [LL73], Earliest Deadline First(EDF) [Bak74], etc. However, even with a variety of extensions [BFR75] [XP90], arbitrary *after* time constraints cannot be used.

In contrast, our approach first transforms *all* precedence constraints into timing constraints, and then treats the problem of finding a code schedule as a multidimensional search problem. Although the transformation of code scheduling into a generalized search is unusual, we have used similar methods to solve other code scheduling problems [ND90]. The difference in this case is that the timing constraints are much more complex than the constraints we have used in previous work, and hence require a more sophisticated search method.

There are many approaches to the *multiple hill climbing problem* [D. 88]. *Gradient ascent* tends to find only locally optimal solutions [GG91]. *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 [KJV83] [HC94]. We have found that a *genetic search* is most effective in finding a globally optimal solution, i.e., a code schedule that satisfies all timing constraints.

Since the genetic algorithm, which is based on the darwinian theory of evolution, was introduced by John Holland [Hol75], it has been actively studied to solve complex problems that are unsolvable by conventional approachs such as traveling salesmen's problem [GGRG85], bin-packing problem [FD92], job shop problem [Dav85], and multiprocessor scheduling problem [HAR94]. Also, the applications with the genetic approach spans vast areas like PCB assembly planning [LWJ92], machine learning [Eng85], and pattern recognition [Sta87].

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 the random behavior. Hence, in this paper, we propose an adaptive genetic search techniques 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 deterministic behavior based on program structure so that the search space is reduced without satisfying the valid schedules.

In section 2, the computational model with arbitrary *before* and *after* timing constraints is defined, and various terms 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 Computational Model

Consider the trivial control program fragment shown in Figure 1. As written, this code would fail to meet the timing constraints specified in the comments. If we assume that each instruction takes 1

Figure 1: An invalid schedule in a real-time program

```
i1 :  Load r1,sense1 ; Get value from memory-mapped sensor 1
                     ; i1 must execute no later than 1 μs after i3
i2 :  Store act1,r1  ; Send value to actuator 1
                     ; i2 must execute at least 1 μs after i5
i3 :  Load r2,sense2 ; Get value from memory-mapped sensor 2
                     ; i3 must execute no later than 1 μs after i1
i4 :  Add r3,r1,r2   ; Add the sensor inputs
i5 :  Store act2,r3  ; Send result to actuator 2
```

μs to execute, two of the three timing constraints are violated. First, the order of the two actuators being triggered is incorrect; i_2 executes 3 μs before i_5. Second, i_3 reads the second sensor's value 1 μs too late.

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 1, 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). This section formally defines the scheduling problem for a hard real-time system.

Definition 2.1 Define the set of instructions to be scheduled as $I = \{i_1, i_2, i_3, \cdots i_n\}$ where n is the number of objects to be scheduled.

Definition 2.2 The set of constraints is defined as $C = \{c_k \mid k \in N \text{ and } 1 \leq k \leq m\}$, i.e. $C = \{c_1, c_2, c_3, \cdots c_m\}$ where m is the number of constraints. Each component c_k describes a constraint between two instructions.

Definition 2.3 All timing and precedence constraints can be expressed as timing constraints in a standardized form:

1. before constraint : $i_x < i_y + \tau$ (i_x must happen at latest τ after i_y)
2. after constraint : $i_x > i_y + \tau$ (i_x must happen at earliest τ after i_y)

3. concurrent constraint : $i_x = i_y + \tau$ (i_x must happen exactly at τ after i_y)
4. exclusive constraint : $i_x \neq i_y + \tau$ (i_x must NOT happen at τ after i_y)

Each standard-form constraint c_k is denoted by $c_k =< i_x, i_y, rop, \tau >$. The value τ is an integral number of time units. The timing relation, rop, can be any of before (<), after (>), concurrent (=), or exclusive ($\neq$). [1]

A key feature of our approach is that all ordering constraints are encoded in terms of these timing relationships. All precedence constraints of the form "*x uses y's result*" are converted into timing constraints of the form "$< x, y, after, 0 >$". For example, in Figure 1, $< i_2, i_1, after, 0 >$ is implied because the value of r1 is generated in i_1 and used in i_2.

Definition 2.4 A schedule s is defined as an execution sequence of the instructions in I; i.e., $s_k = i_{k(1)} i_{k(2)} i_{k(3)} \cdots i_{k(n)}$ 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., $S = \{s_k \mid s_k \in U \ and \ \forall j, c_j \in C \ holds \ for \ s_k\}$) 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.

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.

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

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 $i_{k(x)}$ and $i_{k(y)}$, denoted as $T(s_k, x, y)$, is calculated as:

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

We use these time estimates only to determine which timing constraints have been violated; absolute execution time for the schedule is irrelevant. The quality of a schedule is measured by a penalty function, $\mathcal{F}(s_k, C)$, which counts the number of timing constraints from the set C that are violated by schedule s_k. It does this assuming that all needed resources are available (e.g., register allocation is performed separately).

3.2 Basic operations

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

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 it. Crossover is the process of creating new schedules by "mating" portions of existing schedules. The hope is that 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.

This is ensured by first selecting a single instruction $i_{p(x)}$ in a copy of the parent schedule s_p. To increase the probability of improving upon the parent, only instructions that violate constraints are considered when selecting $i_{p(x)}$. The corresponding instruction $i_{q(x)}$ from the other parent schedule s_q is then 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 2 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.

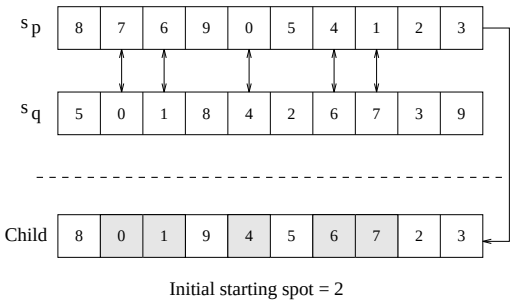

Figure 2: Crossover operator applied to pair of schedules

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.

Mutation of a schedule is performed 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 constraint.

lead to failure of the search.

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 violating timing constraint. An example is given in Figure 3.

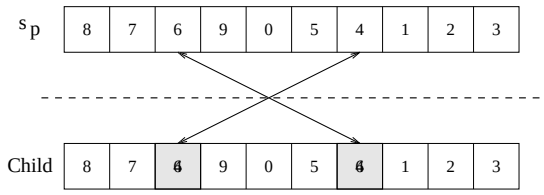

Figure 3: Mutation operator applied to a schedule

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. In rotation, the entire subsequence of the schedule between the two selected instructions is rotated as shown in Figure 4.

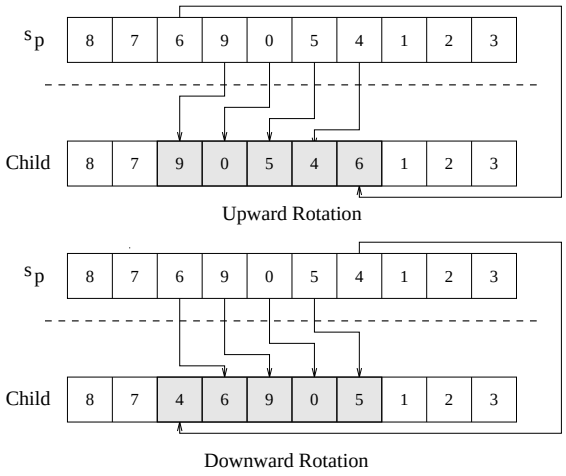

Figure 4: Rotation operators applied to a schedule

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 checking 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 $n - \beta$. If i_x is placed outside of this range, the schedule is guaranteed to be *invalid*.

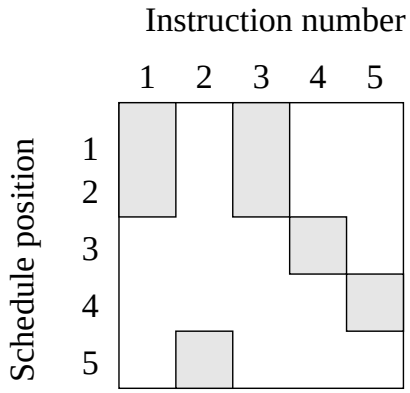

Figure 5: Search space with pruning technique

In our algorithm, this position pruning technique is implicitly employed to all the genetic operations except the crossover process 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. The figure 5 shows that how much of the search space is reduced for the instruction sequence given in example 1 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

by [HG93] 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 [ND90] 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:

$$\mathcal{H}(s_k) = \sum_{j=1}^{n} \Phi(j) * i_k(j)$$

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 discarded when $\mathcal{H}(s_p) = \mathcal{H}(s_q)$ 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 $c_{\tilde{k(b)}}$ or $c_{\tilde{k(e)}}$. $\tilde{c_k}$ denotes one of the

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

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
	$\gamma_0 = \emptyset$, $\gamma_1 = \emptyset$
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 $s_{\tilde{p}}$

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 r_2 into the corresponding positions of r_2 in s_p. In fact, the positions of the instruction in r_1 and r_2 are identical. Figure 2 shows that how the replacement is taken place.

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:	$s_{\tilde{p}} = i_{\tilde{p}(0)} i_{\tilde{p}(1)} \cdots i_{\tilde{p}(n)}$
Procedure:	
Step 1:	select $i_{p(x)}, i_{p(y)}$ from s_p
Step 2:	$s_{\tilde{p}} = s_p \oplus Swap(i_{p(x)}, i_{p(y)})$
Step 3:	return $s_{\tilde{p}}$

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:	$s_{\tilde{p}} = i_{\tilde{p}(0)} i_{\tilde{p}(1)} \cdots i_{\tilde{p}(n)}$
Procedure:	
Step 1:	select $i_{p(x)}, i_{p(y)}$ from s_p
Step 2:	$i_{temp} = i_{p(x)}$;
Step 3:	From $index = i_{p(x+1)}$ to $i_{p(y)}$
	$s_{\tilde{p}} = s_p \oplus Swap(i_{p(index-1)}, i_{p(index)})$
Step 4:	$i_{p(y)} = i_{temp}$
Step 5:	return $s_{\tilde{p}}$

Mutation and rotation algorithm The mutation algorithm and rotation algorithm similarly performs instruction 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 any constraints. Also, random number generator selects the other instruction, $i_{p(y)}$, to be in $PES_{i_{p(x)}}$ 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.

Main: Adaptive genetic algorithm Step 1 of the main algorithm determines the parametric values for 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), and by mates(k) and the proportions by any other basic operations. 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.

Main:	**Adaptive genetic algorithm**
Inputs:	$s_{in} = i_{in(0)} i_{in(1)} \cdots i_{in(n)}$, and $C = \{c_1, c_2, \cdots, c_m\}$
Output:	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 perturbing s_{in}
Step 3:	$\forall s_i \in G_j$, compute $\omega = \mathcal{F}(s, C)$.
Step 4:	If ($\exists s_i \in G_j$ such that $\omega = 0$), then return s_i
Step 5:	Set $G_{j+1} = \emptyset$
Step 6:	Select m best schedules from G_j, say $\aleph_j$.
	Then, set $G_{j+1} = G_{j+1} \cup \aleph_j$
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 $G_{j+1} = G_{j+1} \cup Crossover(s_x, s_y)$
Step 8:	Apply other operations to generate the rest((p-(m+k)) times):
	If($c_{k(rop)} = after$, $\mid c_{k(e)} - c_{k(b)} \mid < c_{k(offset)}$), $G_{j+1} = G_{j+1} \cup Stretch(c_k)$
	else if($\forall \bar{c_k}$, $\mid c_{k(e)} - c_{k(b)} \mid = \epsilon$), $G_{j+1} = G_{j+1} \cup Exhaust(c_{k(b)}, c_{k(e)})$
	otherwise, $G_{j+1} = G_{j+1} \cup Mutate(s_i)$
Step 9:	If($\exists s_x, \exists s_y, s_x = s_y$), $s_y = Perturb(s_y)$
Setp 10:	If($G_i = G_{i+1}$ or i = algorithm threshold) Stop
Step 11:	Set $j = j + 1$ and go to Step 3.

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 whose properties are equivalent to the simple 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 $\mathcal{F}(s, C^*) = 0$ are selected. 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 $\mathcal{F}(s, C^*) = 0$.

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, stretch, exhaust, rotate or mutate are selected based on the type of constraints or length of blocks with violating 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. We call the limit *algorithm threshold* rather than trying infinite generations. The program terminates when one of following conditions is met.

1. When a valid schedule is found. i.e., $\exists s$ where $\mathcal{F}(s, C) = 0$ (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., $\forall s_i \in G_{max}, \mathcal{F}(s, C) < 0$ (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.

4 Experiments

4.1 Experimental environment

Our experiments are performed on IBM RISC System/6000, and Sun Sparc10 workstations. In this section, we present the results obtained from running the 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 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 deteriorated 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 program consists of 64,80,96,112 or 128 instructions and κ values are between 3.0 and 4.0. Each generation consists of 32,64,96,128,160,192,224 or 256 schedules. 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 method [Chu94].

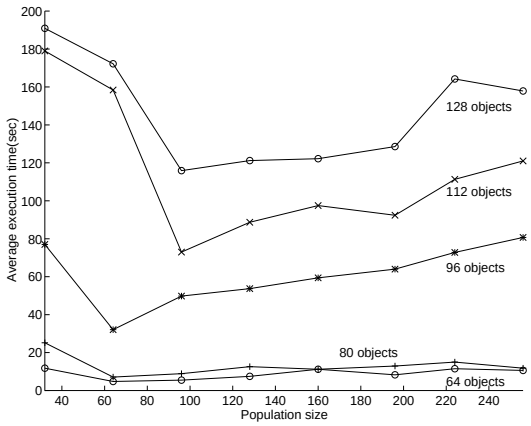

Figure 6: Execution times for various size of problems

For this experiment, the algorithm executions are repeated until 30 valid schedules are found when the algorithm threshold is 4,096 generations. Then, the execution times to find the valid schedules are averaged. The result shown in Figure 6 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 longer for larger populations while the randomness is not much improved after a certain size of population. 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 30 are given in table 1.

Table 1: Success rates for population size of 32

number of instructions				
64	80	96	112	128
0.8	0.5	0.3	0.2	0.2

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 $\delta = 0$. The results with different δ values are also obtained. The initial population is manipulated by adjusting the value of δ. Figure 7 illustrate the results for different δ 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.

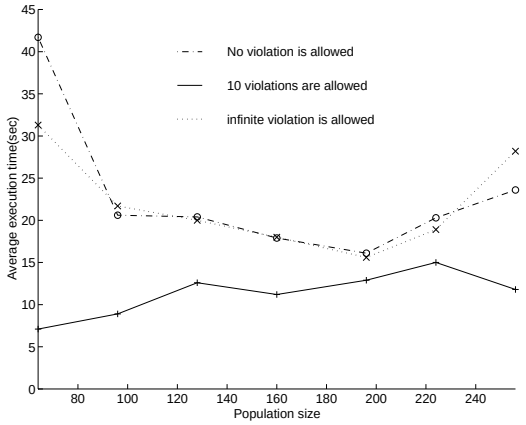

Figure 7: Execution times for different degree of discrimination

Further, we were interested in learning the improvement of having adaptive mechanisms on top of naive genetic algorithms. Hence, 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 8. 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.

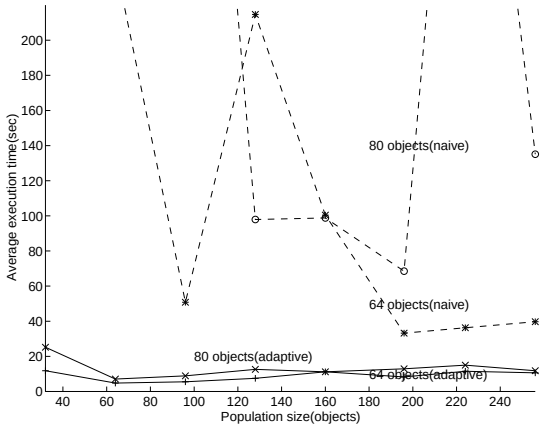

Figure 8: Execution times of naive and adaptive genetic algorithms

SCM ratio, based on the number of offsprings generated by natural selection, crossover, and other basic operations, is another significant performance factor. We vary 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 9 depicts the behavior of the algorithm with different SCM ratio. For this experiment, we ran the algorithm for 80 instructions with various population size, and the algorithm threshold is 4,096 generations. 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.

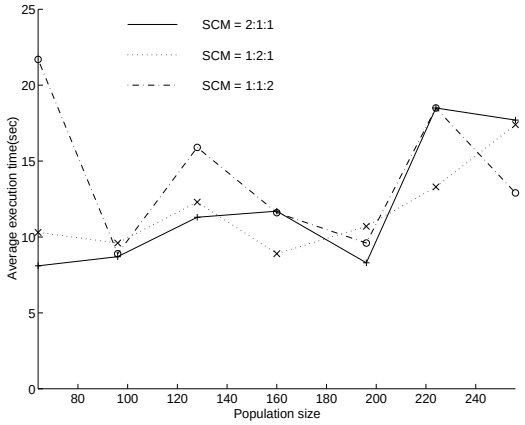

Figure 9: Execution times with various SCM ratio

5 Conclusion

In this paper, we have presented a method for statically scheduling hard real-time programs at the instruction level. Unique to our approach is the fact that fully general timing constraints are used, and all precedence constraints are converted into these timing constraints for analysis.

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 100 instructions 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.

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).

References

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, somewhere, 1974.

[BFR75] P. Bratley, M. Florian, and P. Robillard. Scheduling with earliest start and due date constraints. *Navals Research Logistics Quarterly*, 22(1):165 – 173, December 1975.

[BHJ+82] M. Brady, J. M. Hollerbachand, T. L. Johnson, T. Lozano-Perez, and M. Mason. *Robot Motion : Planning and Control.* MIT Press, Cambridge, Massachusetts, 1982.

[Chu94] T. M. Chung. CHaRTS: Compiler for Hard Real-time Systems. *PhD Thesis Proposal, Purdue University*, April 1994.

[D. 88] D. Brand. Hill climbing with reduced search space. In *IEEE International Conference on Computer-Aided Design*, pages 294 – 297, Santa Clara, CA, November 1988.

[Dav85] L. Davis. Job shop scheduling with genetic algorithms. In *Proc. of International Conference on Genetic Algorithms and Their Applications*, pages 136 – 140, 1985.

[Eng85] A. C. Englander. Machine learning of visual recognition using genetic algorithms. In *Proc. of International Conference on Genetic Algorithms and Their Applications*, pages 197 – 201, 1985.

[FD92] E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. In *IEEE International Conference on Robotice and Automation*, pages 1186 – 1192, Piscataway, NJ, May 1992.

[GG91] H. Guo and S. B. Gelfand. Analysis of gradient descent learning algorithms for multilayer feedforward neural networks. *IEEE Transactions on Circuits and Systems*, 38(8):883 – 894, August 1991.

[GGRG85] J. J. Grefenstette, R. Gopal, B. Rosmaita, and D. V. Gucht. Genetic algorithms for the traveling salesman problem. In *Proc. of International Conference on Genetic Algorithms and Their Applications*, pages 160 – 168, 1985.

[GJ79] 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.

[GPS92] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. Technical Report UMIACS-TR-92-118, University of Maryland, October 1992.

[HAR94] E. Hou, N. Ansari, and H. Ren. Genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113 – 120, February 1994.

[HC94] S. M. Hart and C. S. Chen. Simulated annealing and the mapping problem: a computaional study. *Computers Operations Research*, 21(4):455 – 461, April 1994.

[HG93] S. Hong and R. Gerber. Compiling Real-Time Programs into Schedulable Code. In *Proc of the SIGPLAN'93 Symposium on Programming Language and Implementation*, August 1993.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems.* PhD thesis, University of Michigan, Ann Arbor, MI, 1975.

[Inc88] Texas Instruments Inc. *Third Generation TMS320 User's Guide.* Texas Instruments Inc., 1988.

[KJV83] S. Kirkpatrik, C. Gelatt Jr., and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671 – 680, May 1983.

[LL73] C. L. Liu and J. Layland. Scheduling algorithms for multiprocessing in a hard real-time environments. *Journal of ACM*, 20(1):46 – 61, January 1973.

[LWJ92] M. C. Leu, H. Wong, and Z. Ji. Genetic algorithms for solving printed circuit board assembly planning problem. In *Proc. of JAPAN/USA Symposium on Flexible Automation*, volume 2, pages 1579 – 1586, 1992.

[ND90] 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, pages 61 – 64, St. Charles, IL, August 1990.

[SR91] J. A. Stankovic and K. Ramamritham. *Hard Real-Time Systems Tutorial.* IEEE Computer Society Press, 1991.

[Sta87] I. Stadnik. Schema recombination in pattern recognition problem. In *Proc. of International Conference on Genetic Algorithms and Their Applications*, pages 27 – 35, 1987.

[XP90] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines precedence, and exclusive relations. *Software Engineering*, pages 350 – 359, March 1990.