12-1994

An Evolutionary Approach to Allocating Tasks in Hard Real-Time Distributed Systems

Christian Lang

AN EVOLUTIONARY APPROACH TO ALLOCATING TASKS IN HARD REAL-TIME DISTRIBUTED SYSTEMS

by

Christian Lang

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science in Engineering
Department of Electrical Engineering

Western Michigan University
Kalamazoo, Michigan
December 1994

ACKNOWLEDGMENTS

There are many people who helped me in writing this thesis. I can only mention a few. First, I want to thank my major advisor for his guidance throughout this work. He was always open to my questions. Second, I wish to express my gratitude to the members of the thesis committee for their reading and assistance. I am grateful for the supporters of the Rotary International scholarship. Without their financial support, my thesis would not be possible. Finally I want to thank Karl Sandelin and his wife for being my host family.

Christian Lang

AN EVOLUTIONARY APPROACH TO ALLOCATING TASKS IN HARD REAL-TIME DISTRIBUTED SYSTEMS

Christian Lang, M.S.E.

Western Michigan University, 1994

In real-time systems, correctness not only depends on the result of the computation but also on the time at which this result is available. The violation of timing constraints in hard real-time systems can be critical to human life or environment. Therefore, the scheduling algorithm for distributed systems has to allocate tasks to processing nodes so that no timing constraints can be violated. In addition to timing constraints, tasks have precedence and fault-tolerance constraints.

Static scheduling allocates tasks to processing nodes before the tasks are executed. Static scheduling problems are known to be NP-hard [4]. Therefore, heuristic techniques are necessary to find schedules. Evolutionary strategies (ES) have been used to find solutions to NP-hard optimization problems by performing a directed random search in a complex fitness landscape. Recently, ES have been shown to efficiently find low schedule length task allocations in non-real time distributed systems [7].

This thesis shows, ES algorithms can find solutions to the static scheduling problem in real-time distributed systems. The effect of the type of the genetic operators, the populations size, and the fitness function on the efficiency of the ES algorithms were investigated. The ES approach was verified by solving two real-world scheduling problems from [10] and [24]. Solutions were found in less than one hour of CPU time on a Sun SPARC IPC computer.

TABLE OF CONTENTS

Table of Contents—Continued

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The use of computers in new areas demands properties which require the use of distributed real-time systems. Among such areas are laboratory control, aircraft avionics, and autonomous land rovers. In real-time systems, correctness not only depends on the result of the computation, but also on the time at which this result is available. The use of distributed systems is necessary because of two reasons.

1. A short reaction time often demands a high computational speed.
2. A fault in a real-time system can have catastrophic consequences.

Distributed systems can detect faults by comparing results from different processors. They can recover from an error by migrating tasks to other processors.

Many real-time systems in the past were designed ad hoc. Timing specifications were verified after designing the system by simulation. Slight changes required expensive new simulation of the whole system. The complexity of modern real-time systems makes it impossible to simulate all situations. Thus, a guarantee that timing constraints are met under all circumstances is difficult to obtain. This makes new design methods necessary if violation of timing constraints is critical. These methods have to include timing specifications in all stages of the design.

Conventional design methods minimized the average response time to a given computational task. In real-time systems, the response time must not exceed a specified limit. The average response time is of secondary interest. Real-time systems have to be predictable and timely rather than just fast.

Scheduling is a part of the design process of real-time systems. A schedule allocates tasks to processors and determines the order of their execution. Scheduling algorithms for non-real-time systems minimize the total execution time for a given set of tasks. Real-time scheduling algorithms have to find schedules that satisfy the *timing constraints* of each task. There are two major classes of scheduling algorithms [6].

1. Off-line or *static scheduling* allocates tasks to processors before the tasks are executed. This requires all information about the tasks, such as, the number of tasks and their duration, is known beforehand.

2. On-line or *dynamic scheduling* allocates tasks while running the real-time system. Therefore, the scheduling algorithms is part of the real-time system itself, and is subject to timing constraints as well. On-line scheduling is necessary if some information about the tasks is only available when running the system. For instance, tasks can enter the system asynchronously as a result of sensor input.

In general, off-line scheduling algorithms can find excellent schedules, but the computation time can be excessive. Modern real-time systems are likely to combine both, on-line and dynamic scheduling algorithms. Periodic tasks with known duration are scheduled off-line. All other tasks are allocated by an on-line scheduler at the time of their arrival at the system.

Hard real-time tasks have to meet their timing constraints under all circumstances. Their violation can be critical to human life or environment. A schedule is said to be *feasible* if the timing constraints of all hard real-time tasks are met. *Soft real-time tasks* may violate their deadlines, however repeated violation degrades the performance of the system, or results in a loss of functionality.

In addition to timing constraints, tasks have precedence and fault-tolerance constraints. *Precedence constraints* describe the data and other dependencies between tasks. They restrict the order of execution of the tasks. *Fault-tolerant tasks* must be

executed on different processing nodes so that correct results can be obtained even if some of the nodes fail.

This thesis discusses the use of evolutionary strategy for off-line scheduling of hard real-time tasks. Even under simplifying assumptions, off-line scheduling is a NP-hard problem [4]. This means, the complexity, for finding an optimal schedule, grows exponentially with the number of tasks. Thus, it is impossible to find optimal schedules in a reasonable amount of time when the number of tasks is large. Heuristic techniques are needed to find approximate solutions. Frequently used are rate-monotonic scheduling [14], critical-path methods [24], branch-and-bound algorithms [10], and evolutionary strategy [7]. Rate-monotonic scheduling and critical-path methods are greedy algorithms. Their potential to approximate the optimal solution is limited, however, they are fast and easy-to-use methods to find a rough estimates about the timing in early states of the design. Branch-and-bound algorithms perform a systematic search among all possible allocations of tasks to processors. Because of the huge number of possible allocations, branch-and-bound algorithms tend to have a large execution time. Evolutionary computation has been shown to be a fast method for non-real-time scheduling. This thesis investigates their application to real-time scheduling problems.

EC is a method to approximate solutions to NP-hard optimization problems. The EC algorithm is given below:

1. Generate a population μ of randomly generated chromosomes.
2. Evaluate the chromosomes in μ to find their fitness.
3. Generate offsprings λ by application of genetic operators to μ.
4. Select highly fit chromosomes for the new population μ.
5. If no feasible solution is found, go to step 2.

Each *chromosome* represents a possible solution to the problem. *Genetic operators* produce offsprings by slightly altering parent chromosomes. The parameter

fitness indicates how close a solution, represented by a chromosome, is to the optimal solution. A solution is said to be feasible if a specified optimization criteria is met.

EC can be applied to a wide range of problems because their functioning relies on the very basic principle of strong causality. Strong causality says that similar causes have similar consequences. Translated into the terms of evolutionary computation, this means small changes in the chromosomes cause small changes in fitness.

Evolutionary strategy, ES is a subclass of EC. ES allows each member of the parent generation to produce a constant number n_O of offsprings. The μ chromosomes with the highest fitness are selected from $\mu+\lambda$ for the new population, where μ is the size of the new population, $\lambda \geq \mu$.

Applied to scheduling, chromosomes represent schedules. Genetic operators alter the order of the execution of tasks, or swap tasks between processors. Schedules are evaluated by simulation. The fitness of a schedule is derived from the completion times of its tasks.

This thesis shows an ES algorithms that can find solutions to static real-time scheduling problems in distributed systems. Event-driven simulation was used to simulate schedules. The effect of the type of the genetic operators, the populations size, and the fitness function on the efficiency of the ES algorithms were investigated. The optimized algorithms was used to solve two real-world scheduling problems. These problems consisted of 64 and 89 tasks. Solutions could be found on a Sun SPARC IPC computer in 8 and 54 minutes respectively.

The remainder of the thesis is organized as follows. Chapter II gives an introduction into other scheduling algorithms. In Chapter III, the principles of the evolutionary strategy are explained. Chapter IV gives the assumptions, which were made about the computer model, to find the length of a schedule. A description of the ES algorithm can be found in Chapter V. Chapter VI shows the results of the parameter

optimization and the solution of the real-world problems. Possible extensions of the algorithm are discussed in Chapter VII.

CHAPTER II

RELATED RESEARCH

This section introduces various aspects of scheduling techniques. Scheduling determines the order of the execution of tasks on a processing node. In systems with more than one processing node, the scheduling algorithm also specifies the node on which a task is executed. Scheduling can be subject to precedence and timing constraints. The next section discusses the nature of these constraints and other criteria that classify scheduling problems. The second part of this chapter describes various scheduling algorithms and the type of problems that can be solved by them. A more detailed discussion can be found in [6].

Classification of Scheduling Problems

There are numerous ways to categorize scheduling problems. Only those factors are described below that have an impact on the structure of algorithms.

Deterministic Scheduling

A scheduling problem is said to be *deterministic* if all information about the tasks is known before executing the tasks. Information about tasks includes the length of a task, precedence constraints, and the amount of communication with other tasks. Also, if applicable, timing constraints are given.

In *non-deterministic* problems, at least some of this information becomes available only after executing some of the tasks. This situation occurs where the task exe-

cution time depends on problem data. In systems with interrupts, the time when a task becomes ready for execution, is unknown.

Number of Processing Nodes

The computing system can have one or more processing nodes. In a single-node system, a solution is a permutation of tasks. In a N-node system, the number of possible solutions increases since each task has to be assigned one of the N nodes. Solutions consist of N permutations Π_1, ..., Π_N of tasks where each task is member of one permutation. The tasks in permutation Π_i are to be executed on node i. For two tasks $\pi_m, \pi_n \in \Pi_k$ (for some k) $m<n$ implies π_m is executed before π_n.

Communication

Many scheduling algorithms neglect the time that is needed for communication between processors. This approach is only valid for tightly coupled systems with a large computation to communication ratio.

If the amount of communication is small compared to the bandwidth of the communication network, it can be assumed to be congestion free. In such systems, the total communication time is equal to the message propagation time. As the amount of communication between processors becomes larger, messages may have to wait for some time until a link becomes available. This affects the mapping of tasks onto the processors. Algorithms that neglect communication tend to balance the load between processors. If communication time is large, minimizing dilation will give good schedules [13, 25]. *Dilation* is the average number of links used to transmit a message.

In [2] it is shown that strategies that find good schedules are likely to compromise between load balance and dilation.

Preemptive Tasks

In a *non-preemptive* schedule, tasks cannot interrupt each other. Once a task T has been started on a certain processor, this processor cannot execute any other task until the computation of T has been finished.

If switching between tasks is possible at any time, the schedule is said to be *preemptive*. Preemptive schedules give less processor idle time and thus, larger speedup. However there is a certain amount of overhead for each preemption, which can reverse this effect. The *speedup* S is defined as $S=t_1/t_N$, where t_1 is the execution time of the best sequential algorithm, and t_N is the execution time on a distributed N-node system.

Precedence Constraints

In many cases, precedence constraints are described by constraint graph. This is a directed acyclic graph (DAG) where nodes represent tasks. An arc leading from node i to node j indicates that the task at node i provides data to the task at node j. Thus, the task at node i must execute before the task at node j can begin execution. A set of tasks is said to be *independent* if there are no precedence constraints. That means, a task can be started without synchronization to other tasks.

Timing Constraints

Much research has been done to find schedules that maximize the total speedup, or minimize the finishing time of the last task [8,11,19,20]. Real-time systems impose additional constraints on the task execution time. These constraints can be earliest start times, latest start times, or deadlines for each task. The meaning of timing constraints depends on the type of the task.

1. Hard real-time tasks have to meet their timing constraints under all circumstances. Their violation can be critical to human life or environment. A schedule is said to be *feasible* if the timing constraints of all hard real-time tasks are met.

2. Soft real-time tasks may violate their deadlines, however repeated violation degrades the performance of the system, or results in a loss of functionality. Soft real-time tasks can be scheduled by assigning a weight to each task. The scheduling algorithm minimizes the total weight of the tasks that miss their deadlines.

Periodicity of Tasks

Periodic tasks arrive at the system at regular intervals. If it is possible to find a period common to all tasks, it is sufficient to schedule only those tasks that arrive in that common period [19]. Tasks that do not arrive at the system periodically are referred to as *aperiodic*.

Scheduling Algorithms

If the scheduling problem is non-deterministic, or if aperiodic tasks are present; some of the scheduling decisions have to be made at run-time. That implies that the scheduling algorithm has to be executed at the same time as the problem. To keep the amount of overhead small, the algorithms have to be fast. They are called *on-line* scheduling algorithms [6].

In *deterministic* scheduling problems, with only periodic tasks present, there is enough information to compute a complete schedule before running the system. Once the system has been started the allocation of tasks to processors, and their order of execution is fixed.

On-line Scheduling Algorithms

These algorithms have to schedule tasks without complete knowledge about the problem. Therefore, their ability to optimize certain parameters (resource utilization, satisfying deadlines) is limited. Many of these algorithms assign some kind of priority to the tasks [1]. Tasks are executed in the order of their priority.

Off-line Scheduling Algorithms

Deterministic scheduling problems can be solved before executing the tasks. In principal, it is possible to find an optimal schedule. However, except for some trivial cases, scheduling problems are NP-complete [4]. Therefore, computing an optimal schedule becomes computationally prohibitive if the number of tasks is large. Heuristics are needed to approximate an optimal solution.

Rate Monotonic Scheduling

Rate monotonic scheduling (RMS) applies to the case of preemptive scheduling of independent tasks. It can be used for scheduling tasks with deadlines on a single processor. Tasks are assigned priorities in the order of their periods. The task with the shortest period is assigned the highest priority. It can be shown that the tasks will always meet their deadlines if the total utilization of the processor does not exceed a certain limit [14]. In [23], Sha and Sathaye introduced a generalized rate monotonic scheduling (GRMS), which extends RMS in that, that this technique is now applicable to multiprocessor environments, and to tasks with precedence constraints. This technique is computationally inexpensive, and gives a good theoretical understanding of the problem. However, RMS and GRMS can only be applied to preemptive systems,

which are difficult to implement. Because of the preemption overhead, the achievable speedup can be smaller than that of non-preemptive schedules.

Critical-Path Method

This method can be used for non-preemptive scheduling of tasks with precedence constraints. The technique uses a latest start time (LST) to determine when a task should be scheduled. The LST of a task is defined recursively as the maximum LST of all of its successor tasks plus the execution time of the current task. A task is called ready task if the computation of all of its predecessors has been finished. The critical-path method (CPM) assigns ready tasks to processors in the order of increasing LST [24]. Although this algorithm is very fast, the accuracy of the solution cannot be guaranteed. CPM is the basis for various branch-and-bound techniques.

Branch-and-Bound

Branch-and-Bound algorithms (BBA) can be applied to a variety of off-line scheduling problems. A systematic investigation of all possible assignments of tasks to processors would find an optimal schedule. Since the number of possible schedules increases exponentially with the number of tasks, this is only possible for small problems. BBA use heuristic techniques to find near optimal schedules first. Therefore, good schedules can be found by investigating only a small fraction of all possible schedules.

BBA maintain a list of ready tasks, and another list of idle processors. At each point of time there are several possible assignments of ready tasks to idle processors. A heuristic gives the order in which these assignments are tried. For each assignment, the problem is solved recursively for the remaining tasks. Once a feasible schedule has been found, the search is terminated.

Ramaritham gave an example of the latest start time/maximum immediate successors first heuristic for fault tolerant systems in [19]. Kasahara and Narita showed the use of the depth first/implicit heuristic search algorithms for the case of a robot-arm control in [10].

The total number of all possible schedules grows exponentially. Therefore, the number of schedules, which has to be investigated until a feasible schedule is found, can be rather large for a large number of tasks. This makes BBA computationally expensive for large problems.

CHAPTER III

EVOLUTIONARY STRATEGY

Evolutionary strategies (ES) are used to approximate solutions of optimization problems. They model the process of the biological evolution. A major advantage of ES is that they are not a priori limited to certain types of problems. However, ES are weak computational methods [22]. If other types of algorithms can be applied, like greedy or divide-and-conquer algorithms, they may be faster because they exploit specific properties in the structure of the problem to find an solution. ES are only applied to NP-hard problems where finding an exact solution in a reasonable amount of time is impossible. Although there are many variations, all ES are based on the algorithm below:

1. Create an initial population μ of randomly generated chromosomes.
2. Evaluate the chromosomes in the population to determine their fitness.
3. N_g=0.
4. Generate λ offsprings by application of genetic operators to the individuals in μ. Each chromosome in μ produces n_o offsprings.
5. Evaluate the chromosomes λ to determine their fitness.
6. Select the μ chromosomes with the highest fitness in $\mu+\lambda$ for the new population μ.
7. $N_g=N_g+1$.
8. If no feasible solution was found and $N_g<N_{max}$, go to step 4.

ES improve a *population* of chromosomes until a feasible solution is found. Each *chromosome* represents a potential solution to the problem. *Genetic operators*

produce *offsprings* by slightly altering parent chromosomes. The parameter *fitness* indicates how close a solution, represented by a chromosome, is to the optimal solution. A solution is said to be *feasible* if a specified optimization criteria is met.

Potential solutions are usually encoded by arrays of real or integer numbers, and are referred to as *genotypes*. Simulation of a genotype reveals its behavior concerning an optimization parameter. This behavior is called the *phenotype*.

For example, consider the traveling-salesman problem (TSP) [17]. An instance of the problem consist of n cities where the distances between cities are known. TSP asks for the shortest closed-loop tour including all n cities. A possible solution, or chromosome, is a permutation of the n cities. An array of n integers can be used as genotype where each integer represents the number of a city. Computation of the length of the tour reveals the phenotype. Since the length L of the tour is to be minimized, the fitness F can be computed by $F=-L$. This gives good solutions, with a small length L, the highest fitness.

The remainder of this chapter discusses the parts of ES in greater detail.

Representation of Individuals

Simple genetic algorithms used bit strings as the most general representation of genotypes. For many problems, this is a rather unnatural encoding. ES frequently use arrays of integer for combinatorial optimization problems, such as, scheduling problems [8], or the TSP shown above.

Genetic Operators

There are two types of genetic operators which modify the parameter values encoded in the genotype. Mutation operators change one parent to produce an offspring. Recombination operators combine the properties of two parents in the new off-

spring. The question, which type of operators should be used, has been the subject of much discussion, and it seems the answer depends on the problem to be solved. Rechenberg suggests to only use mutation [21]. Mühlenbein and Schlierkamp-Voosen derived a relation between the convergence rate and the type of the genetic operator [18]. They could not show clear advantage for either recombination or mutation operators. *Convergence rate* is the average number of generations that has been computed when one genotype is distributed throughout the entire population.

ES only work better than random search if there is a correlation between the fitness of a genotype and its offspring. This means highly fit genotypes should have a high probability to produce highly fit offsprings. This relationship can be expressed by the covariance of the fitness of parent and offspring. Manderick and Spiessens show how covariance coefficients can be used to select efficient genetic operators for combinatorial optimization problems [16]. Mutations must satisfy the principle of strong causality. That means small changes in the genotype have a high probability to result in small changes in the phenotype [21].

Correlation of the fitness between parent and offspring requires a small mutation step size. However, if the mutation step size is too small the evolution may converge to a local optimum instead of the global optimum. Therefore, efficient evolution only takes place within a small interval of the mutation step size. Some algorithms adapt the mutation step size according to the mean fitness of the population. It is also possible to optimize the mutation step size by evolution of the second kind. The mutation step size is part of the genotype and optimized by evolution as well [21].

Generation and Selection of Chromosomes

Selection determines the direction of the evolution. Offsprings λ are generated by application of genetic operators to a parent population μ. Selection determines

which individuals from $\mu+\lambda$ are chosen for the next population. Selection can be based on the fitness, or on the rank of a chromosome in the population. Most ES use *truncation selection*, a rank-based method. Each individual of the population produces a constant number n_O of offsprings. Therefore, λ contains $n_O*\mu$ chromosomes. From the total $\mu*(n_O+1)$ chromosomes in $\mu+\lambda$, the μ individuals with the highest fitness are selected for the new population. Usually $n_O=1$ is used. In situations where mutation operators can produce invalid offsprings, $n_O>1$ can provide a surplus of chromosomes. In this thesis, $n_O=1$ was used.

CHAPTER IV

PROBLEM DESCRIPTION

This section explains the details of the scheduling problem that can be solved by the algorithm described in Chapter V.

The problem consists of p tasks, which are to be executed on a multicomputer with n nodes. Each task has precedence, timing, and fault tolerance constraints. The nodes of the multicomputer communicate by passing messages over a network. The scheduling problem is to find a feasible schedule, i.e., an ordered assignment of tasks to nodes; so that no precedence or fault-tolerance constraints are violated, and all timing constraints are satisfied.

Tasks are non-preemptive and periodic. Once a task T_0 has been started on a certain node this node cannot execute any other task until T_0 was completed. All tasks have the same period. Therefore, it is sufficient to find a feasible schedule for one period [20].

Precedence constraints are given in the form of a directed acyclic graph, referred to as precedence constraint graph (PCG). Each vertex V_i of the graph represents a task T_i. An arc from vertex V_i to V_j requires that task T_j not be started before T_i has been completed. If T_i and T_j are executed on different nodes N_k and N_l of the multicomputer, a message has to be sent from N_k to N_l upon completion of T_i. T_j cannot be started until this message arrives at N_l.

Timing constraints consist of a deadline t_{di} for each task T_i. Timing constraints are violated if the completion time t_{ci} of a task is larger than its deadline, $t_{ci} > t_{di}$.

It is assumed that all processing nodes are homogenous. That implies that the execution of each task is possible on any node, and that the length of the execution of a task is the same on each node.

To describe fault tolerance, tasks are divided into classes. Each task is member of exactly one class. Fault tolerance constraints require that each member of a class is executed on a different node of the multicomputer. This implies that a class can have at most as many members as the multicomputer has nodes. The special case, where each class consists of one task, is equivalent to the case with no fault tolerance constraints.

The nodes are connected by point-to-point links. Communication is assumed to be reliable and full-duplex. Each link has a buffer for messages that are waiting for access to the link. No store-and-forward scheme was used. At intermediate nodes, messages are relayed as soon as the outgoing link at the intermediate node becomes available.

If task T_i completes on node N_k, messages have to be sent to all immediate successors of T_i in the PCG that are scheduled on nodes other than N_k. The successor tasks cannot be started before these messages have been delivered. The time, between the completion of T_i and the delivery of a message, is referred to as communication delay d_C. The communication delay (d_C) is the sum of three parts: (1) transmission delay (d_t), (2) propagation delay (d_p), and (3) queuing delay (d_q). Specifically:

1. Transmission delay, d_t, is due to the bandwidth of the communication channels; $d_t=B*l_M$, where B is the channel bandwidth in byte/s, and l_M is the length of the message M in bytes. In this thesis, all messages are assumed to be of same length. Therefore, d_t is constant for all messages.

2. Propagation delay, d_p, is due to the finite speed, c, of signals; $d_p=c*d$, where d is the distance between nodes. Propagation delay was assumed to be small against the other two parts of delay.

3. Queuing delay, d_q, occurs if a message has to be delayed because the scheduled link for this message is used by other messages. Queuing delay was modeled by introduction of communication tasks in the simulation of schedules.

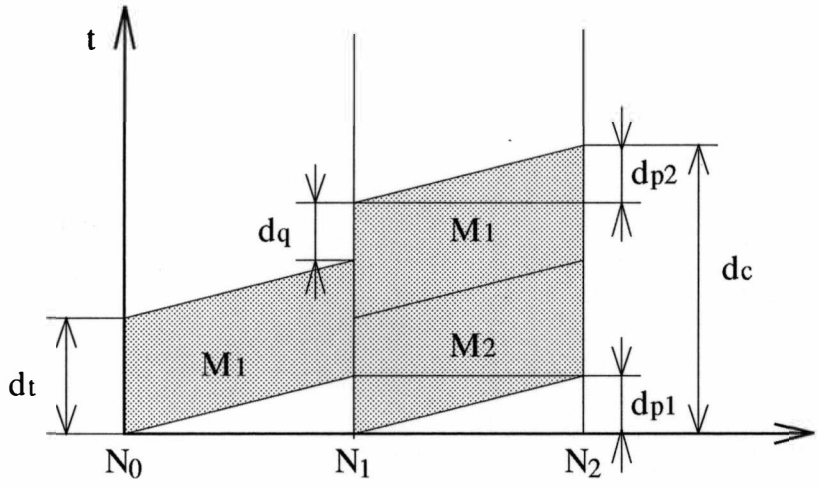

Figure 1. Timing Scheme for Messages Routed Via Intermediate Nodes.

Figure 1 shows the different types of delay for the transmission of a message $M1$ from node $N0$ via $N1$ to $N2$. $M1$ is delayed at the intermediate node $N1$ because the link from $N1$ to $N2$ transmits another message $M2$.

Routing of messages is oblivious to existing traffic, and messages in transit cannot be preempted. If more than one message requests access to a link, they are transmitted in the order of the requests.

CHAPTER V

EVOLUTIONARY STRATEGY APPROACH

This chapter shows how the scheduling problem from Chapter IV can be solved using an evolutionary strategy. The algorithm was implemented in C++. The object-oriented concept of this language allowed an easy implementation of simulation of the schedules.

The main program consists of a loop of generating and selecting schedules. The loop is terminated when a feasible schedule is found, or when the number of generations N_g exceeds the maximum number of generations N_{max}:

1. Create an initial population of random schedules, N_g=0.
2. Create new schedules by mutation.
3. Simulate all schedules to determine their fitness.
4. Select the surviving schedules for the next generation, N_g=N_g+1
5. If no feasible schedule was found and $N_g < N_{max}$, then go to step 2.

A schedule consists of n sequences of task numbers. There is one sequence for each node of the distributed system. The order in the sequence determines the order of execution on the node.

The next sections describe these steps in greater detail.

Initial Population

The first generation consists of a population of random schedules. All tasks in the PCG are numbered by breadth-first search. Tasks are the assigned to processing nodes of the distributed system in increasing numerical order, where the processing

nodes are randomly selected. This method has the advantage of producing schedules that are deadlock free because no precedence constraints are violated.

Generation of Offsprings

The algorithm only uses mutation operators to generate new offsprings. No recombination operator was used. There are many ways to mutate a schedule. Since it is difficult to derive the efficiency of a particular operator theoretically, the performance of 6 different mutation operators was tested. This section describes the operators. Examples are given for the task graph in Figure 2.

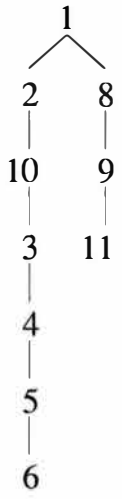

Figure 2. Precedence Constraint Graph for the Mutation Operators.

The tasks are executed on a multicomputer with two processing nodes. The mutation operators were applied to the schedule below.

N_1: 1(0,1) 2(1,2) 3(6,7) 4(7,8) 5(8,9) 6(9,10) 7(10,11)

N_2: 8(2,3) 9(3,4) 10(4,5) 11(5,6)

The numbers in brackets are the start and completion time of the tasks. They refer to a message transmission delay of 1 unit of time. The duration of all tasks is 1 unit of time.

1. Operator $M1$ randomly selects a processing node N_i and a task T_m scheduled on N_i. T_m and its immediate successor are swapped. For i=1 and m=4, $M1$ gives the schedule:

$N1$: 1 2 3 **5 4** 6 7

$N2$: 8 9 10 11

2. Operator $M2$ randomly selects a processing node N_i and two tasks, T_m and T_n, scheduled on N_i. T_m and T_n are swapped. For i=2, m=8, and n=10, $M2$ gives the schedule:

$N1$: 1 2 3 4 5 6 7

$N2$: **10** 9 **8** 11

3. Operator $M3$ randomly selects two processing nodes N_i and N_j. It randomly selects a task T_m and T_n scheduled on N_i and N_j respectively. T_m and T_n are swapped. For i=1, j=2, m=2, and n=9, $M3$ gives:

$N1$: 1 **9** 3 4 5 6 7

$N2$: 8 **2** 10 11

4. Operator $M4$ randomly selects two processing nodes N_i and N_j. It randomly selects tasks, T_m and T_n, scheduled on N_i and N_j respectively. All tasks, scheduled after T_m and T_n, are swapped. For i=1, j=2, m=3, and n=9, $M4$ gives:

$N1$: 1 2 3 **10 11**

$N2$: 8 9 **4 5 6 7**

5. Operator $M5$ randomly selects two processing nodes N_i and N_j. It randomly selects two tasks, T_{m1} and T_{m2}, scheduled on N_i, and one task T_n scheduled

on N_j. The sequence of tasks from T_{m1} to T_{m2} is inserted into the execution thread of N_j after T_n. For i=1, j=2, m_1=4, m_2=5, and n=10, M_5 gives:

N_1: 1 2 3 6 7

N_2: 8 9 10 **5 4** 11

6. Operator M_6 works much like M_4, but T_n is not randomly selected. T_n is chosen so that the crossover point on N_j is close in time to the crossover point on N_i. Let T_p and T_q be the tasks scheduled immediately after T_m and T_n respectively, see Figure 3.

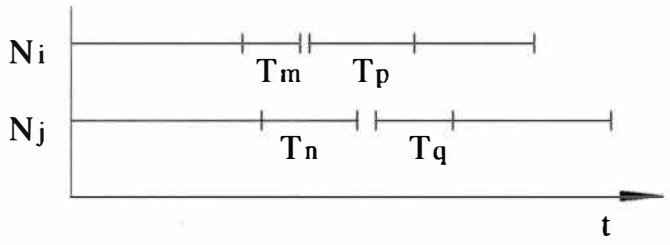

Figure 3. Schedule for the Crossover Operator.

T_n is selected so that the timing condition TC is satisfied.

TC $\quad t_{sm} < t_{cq}$ and $t_{sn} < t_{cp}$

Let t_{si} denote the start time and t_{ci} the completion time of task T_i respectively. These times are known from the computation of the fitness for the schedules in the parent generation. Therefore, the start and completion time of the tasks have been computed previously to find the fitness. Assume that i=1, j=2, and m=3. A valid choice for M_6 is n=10, which gives p=4, q=11, t_{sm}=8, t_{cq}=9, t_{sn}=6, t_{cp}=10. Therefore, TC is satisfied:

N_1: 1 2 3 **11**

N_2: 8 9 10 **4 5 6 7**

The reason for TC is that M_6 always produces deadlock-free offsprings from deadlock-free parents. This is not the case if T_n is chosen randomly as in M_4. The example for M_4 had m=3, n=10, q=10, and t_{sm}=8, t_{cq}=7. This violates $t_{sm}<t_{cq}$ in TC. The schedule deadlocks after the execution of T_2 because T_3 is waiting for the completion of T_{10}, but T_{10} is never executed since it is scheduled on N_1 after T_3.

Check for Fault-Tolerance

Some mutation operators exchange tasks between processing nodes. This can result in schedules that violate fault-tolerance constraints. Each schedule is checked for violation fault-tolerance before simulation. If a schedule violates fault-tolerance constraints, it is removed from the population.

Fault-tolerance constraints are given as a set $G=\{g_1, g_2, ..., g_{ng}\}$ of n_g groups. Each group $g_i=\{T_{i1}, T_{i2}, ..., T_{ik}\}$ consists of a set of fault-tolerant tasks, which have to be scheduled on different processing nodes. For example, let a scheduling problem have 4 tasks $T=\{T_1, T_2, T_3, T_4\}$. Assume that T_1, T_2, and T_3 have to be scheduled on different processing nodes, then $G=\{g_1\}$ and $g_1=\{T_1, T_2, T_3\}$.

To check schedules for fault-tolerance, a two-dimensional array $A[i,j]$, with i=1..n_g and j=1..n_N, is used, where n_N is the number of processing nodes. Assume that the array is initialized with $A[i,j]$=0 for all elements. Let $g[T_i]$ be the number of the group to which T_i belongs ($g[T_i]$=0 if T_i is not a fault-tolerance task). Let $N[T_i]$ be the number of the processing node on which T_i is scheduled by a schedule S. The algorithm below returns *not_valid* if S violates any fault-tolerance constraints:

1. for all tasks T_i in S
2. if $g[T_i]>0$
3. if $A[g[T_i]][N[T_i]] > 0$
4. return *not_valid*
5. $A[g[T_i]][N[T_i]]=1$
6. return *valid*

If two tasks, T_i and T_j, belong to the same group g_k and are scheduled on the same processing node N_l, $A[g_k][N_l]$ is set to 1 by step 5 for T_i. The test in step 3 detects the violation when the loop is executed for T_j. After the test, the original state of A is restored by the same algorithm The only change is in step 5:

5. $A[g[T_i]][N[T_i]]=0$

Let n_T be the total number of tasks. There can be at most $n_g \leq n_T$ different groups in G. The initialization of A has the complexity of $O(n_T * n_N)$. Each test of a schedule has the complexity of $O(n_T)$. Since A has to be initialized only at the beginning of the evolution, the complexity of the check for fault-tolerance is $O(n_T)$ if many schedules (more than n_N) have to be tested. This is faster than simulation of schedules. Therefore, schedules are checked for violation of fault-tolerance first.

Computation of the Fitness

To determine the fitness of a schedule, the completion time t_{ci} of each task T_i is computed by simulation. The fitness is found by decreasing the fitness F for each task that does not meet its deadline t_{di}, (e.g., $t_{di} < t_{ci}$).

Event-Driven Simulation

The completion time of the tasks is found by event-driven simulation. Events are the start and the completion of a task. The event heap contains the tasks that are being executed at a particular point of time. The insertion of a task into the event heap models the start of a task, and its extraction from the event heap models its completion. The heap-order is the completion time of the tasks. Except for the tasks starting with the begin of the simulation at t=0, the start of a task is always caused by the completion of another task. This is due to precedence and scheduling constraints. Therefore, the execution of the tasks can be modeled by the algorithm below:

1. Insert all tasks into the event heap that are scheduled as the first task on a processing node and have no precedences.
2. Extract the task with the minimum completion time from the event heap.
3. Insert all ready tasks into the event heap.
4. If the heap is not empty, go to step 2.

A task becomes ready if all of precedences have been completed. Two types of tasks can be elements of the event heap: (1) computation tasks and (2) communication tasks. Computation tasks are the tasks of the schedule. Their number and their order of execution on a processing node are given with the schedule. Communication tasks are created dynamically in the process of simulation. They model the communication delay of messages. This leads to the data structure shown in Figure 4.

Computation tasks are denoted as Task, communication tasks as C_Task. The event heap points to a set of tasks currently being executed. Each task points to its successor and its descendants in the PCG. The successor is the task scheduled next on the same processing node.

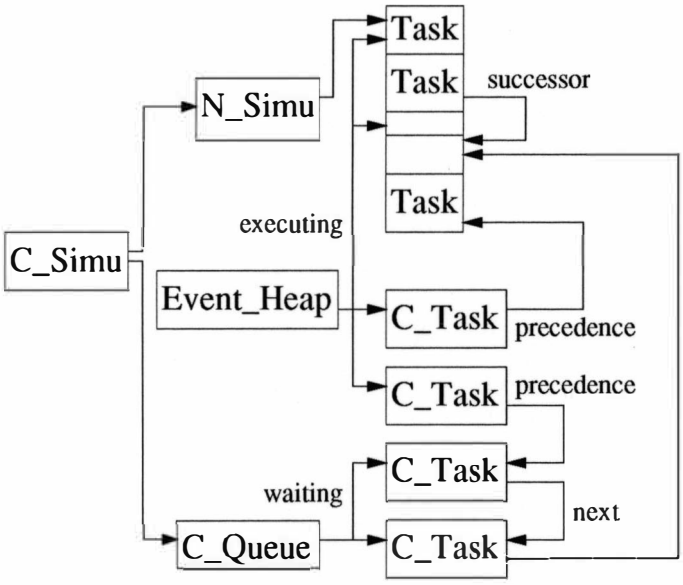

Figure 4. Data Structure for Simulation of the Schedules.

Communication queues, C_Queue (only one is shown in Figure 4), contain all tasks that are waiting for access to a link, which is the case if the event heap already contains another communication task for this link. Upon extraction of a task from the event heap, the precedences of all dependent tasks are updated. If the extracted task was the last precedence being extracted form the heap, the dependent task is inserted into the event heap. Communication tasks are created upon completion of a computation task and are deleted upon their extraction from the event heap. The objects N_Simu and C_Simu monitor the use of processing nodes and communication links. They ensure that at most one task for each processing node or link is member of the event heap.

Modeling of Communication

Communication tasks are created upon extraction of a computation task T_0 from the event heap if T_0 has to send messages to other processing nodes. If T_0 is sched-

uled on processing node N_i, then let T_C be the set of tasks that are immediate successors of T_0 in the PCG, and that are scheduled on processing nodes N_C different from N_i. Upon completion of T_0, a set of messages M has to be sent from N_i to the processing nodes N_C. The messages in M are routed along the shortest paths from N_i to the processing nodes N_C. One communication task is created for each link that is used by the messages in M. Consider the example of Figures 5 and 6.

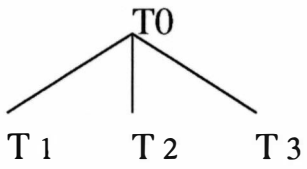

Figure 5. Precedence Constraint Graph for Communication.

The task T_0 is scheduled on processing node N_0 and has the descendants T_1 on N_1, T_2 on N_2, and T_3 on N_3, then, $T_C=\{T_1, T_2, T_3\}$, $N_C=\{N_1, N_2, N_3\}$, and $M=\{m_1(N_0->N_1), m_2(N_0->N_2), m_3(N_0->N_3)\}$. For the ring topology of Figure 6, the set of communication tasks is shown in Figure 7. The arrows between tasks indicate the new precedence constraints. The links are denoted with $L_0=N_0->N_1$, $L_1=N_1->N_3$, and $L_2=N_0->N_2$:

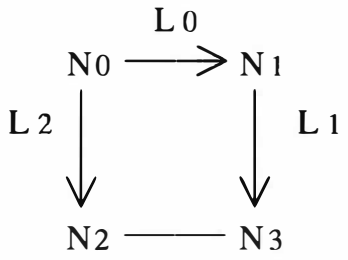

Figure 6. 4-Processing-Node Ring Topology.

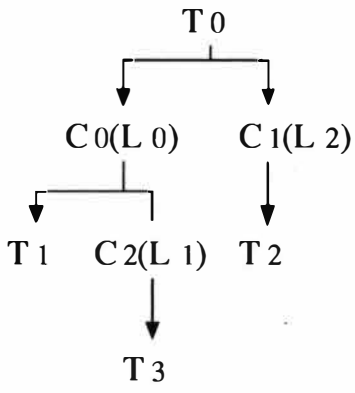

Figure 7. Set of Communications Tasks for a Ring Topology.

Each computation task counts the number of its parents in the PCG that have been extracted from the event heap. Assume that task T_i is scheduled on processing node N_k immediately after task T_j. It is inserted as soon as all of its parent tasks and T_j have been extracted from the event heap.

Queuing delay of communication is modeled by communication queues. A communication queue Q_i for the link L_i contains all communication tasks that are ready for execution and waiting for access to L_i. No store-and-forward routing was used. Therefore, a communication task becomes ready for execution with the insertion of its preceding communications task into the event heap (store-and-forward would require to delay the start of a communication task until the extraction of its preceding task from the event heap). There is one communication queue for each link. When a communication task C_j for the link L_i is created, it is inserted into the event heap if the event heap contains no communication tasks for L_i. Otherwise it is enqueued into Q_i. Upon extraction of a communication task C_n for link L_m, the top of Q_m is dequeued and inserted into the event heap.

Consider the example of Figure 7. Assume that the event heap and the communication queues contain the following tasks before extraction of task T_0:

Event Heap: T_0, $C_7(N_0->N_2)$, $C_5(N_1->N_3)$

Q_0 for L_0: empty

Q_1 for L_1: empty

Q_2 for L_2: $C_6(N_0->N_2)$

Extraction of T_0 from the event Heap would give:

Event Heap: $C_7(N_0->N_2)$, $C_5(N_1->N_3)$, $C_0(N_0->N_1)$

Q_0 for L_0: empty

Q_1 for L_1: $C_2(N_1->N_3)$

Q_2 for L_2: $C_6(N_0->N_2)$, $C_1(N_0->N_2)$

Fitness Function

A deadlock occurs when the event heap becomes empty before all computation tasks have been inserted into the event heap. In this case, the fitness is set to $F = -\infty$ ($-\infty$ is simulated by a number smaller than the smallest fitness possible). Otherwise the fitness is computed by

$$F = -\sum_i (t_{ci} - t_{di})^E \qquad \text{where } T_i \in M \tag{1}$$

where M is the set of tasks T_i that missed their deadlines, i.e., $t_{ci} > t_{di}$. E is referred to as *fitness-function exponent*. Results for $0.6 \leq E \leq 6.4$ are given in Chapter VI. It follows from (1) that all feasible schedules have a fitness of $F=0$.

Selection of Schedules

Assume that the population μ_g of generation g consists of μ schedules. Each schedule S_i in μ_g produces one offspring by application of a mutation operator, see Figure 8.

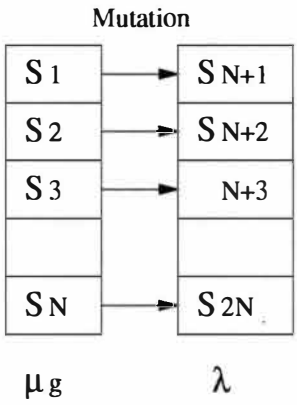

Figure 8. Creation of Offsprings by Mutation.

This creates a collection λ of $\lambda=\mu$ offsprings. The fitness is computed for each schedule in λ. The 2μ schedules from $\mu_g+\lambda$ are now sorted by decreasing fitness. A rank is assigned to each schedule. All schedules with the same fitness have the same rank. The schedules with the largest fitness get the rank r_1, schedules with the second largest fitness get the rank r_2, ... Assume that the total number of ranks is n. Starting with r_1, one schedule is randomly chosen from each rank and moved to the new population μ_{g+1} until μ_{g+1} contains μ schedules. If $n<\mu$, a second member is chosen from each rank, starting with r_1.

Example: The notation $S_i(j)$ means that schedule i has the fitness j. Let be $\mu=4$, $\mu_g = \{S_1(-1), S_2(-1), S_3(-2), S_4(-3)\}$, and $\lambda = \{S_5(-2), S_6(-1), S_7(-3), S_8(-3)\}$. There are three ranks:

r_1: S_1, S_2, S_6

r_2: S_3, S_5

r_3: S_4, S_7, S_8

A selection for μ_{g+1} can be $\mu_{g+1} = \{S_2, S_3, S_8, S_6\}$. Truncation selection would have selected the best four individuals: S_1, S_2, S_6, and S_3.

The proposed selection method has the advantage that it is less likely to lose the genetic variety in the population. An example may show that. Consider the task graph in Figure 9.

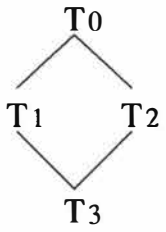

Figure 9. Task Graph for Selection.

Assume that the tasks $T1$ and $T2$ have the same duration and the same deadline. Let S be a highly fit schedule in the population μ_g. A mutation can produce the offspring S', where $T1$ and $T2$ are swapped. S and S' have the same fitness. Let S be the schedule:

$N1$: $T0$ $T1$

$N2$: $T2$ $T3$

The schedule S' can be obtained from S by mutation operator $M3$:

$N1$: $T0$ $T2$

$N2$: $T1$ $T3$

Both schedules have the same fitness. Truncation selection is likely to select both schedules for μ_{g+1}. Subsequent mutations can produce an exceptionally increasing number of copies of S and S' by swapping $T1$ and $T2$. Since both, S and S', have the same fitness, they have the same rank. Therefore, it is unlikely that both schedules are selected for the population μ_{g+1}.

CHAPTER VI

RESULTS

The efficiency of ES depends on various parameters (e.g. population size and type of the genetic operators). It is difficult to optimize these parameters theoretically. The relatively small problem, introduced in [19], was chosen to optimize parameters of the ES. The optimal values, found with the small problem, were used find a solution for two real-world problems. The problem in [19] consists of 11 tasks. The real-world problems have 64 and 89 tasks. ES found a solution for the small problem in 2 seconds and for the real-world problems in 8 to 54 minutes of CPU time on a Sun SPARC IPC.

Parameter Optimization

Figure 10 shows the task graph of the scheduling problem from [19].

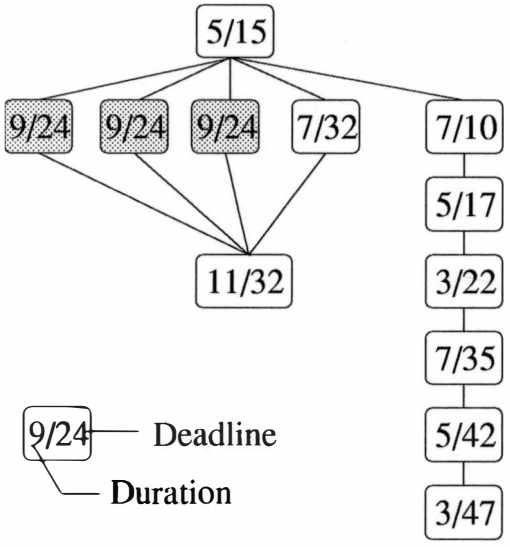

Figure 10. Task Graph of the 11-Task Sample Problem.

The shaded boxes represent tasks that are the result of three replications of a fault-tolerant task. These tasks must be scheduled on different processing nodes. The tasks are executed periodically in 50μs intervals. The tasks were assumed to be executed on a multicomputer with three processing nodes communicating over a ring topology. Transmission delay for all messages was 6μs.

Three parameters were being optimized: (1) type of the genetic operators, (2) population size, (3) fitness-function exponent E (see fitness function (1) on page 30).

Type of the Genetic Operators

Some of the mutation operators described in Chapter V cannot reach all possible solutions. For instance, M_1 and M_2 do not exchange tasks between processing nodes. Therefore, combinations of mutation operators were tested. The offsprings of a schedule are produced with probability P_a by operator M_a and with probability P_b=1-P_a by operator M_b. The efficiency of five combinations was tested: (1) M_1-M_4, (2) M_1-M_5, (3) M_1-M_6, (4) M_2-M_6, and M_3-M_6. The diagrams in Figures 11 and 12 show the average number of generations N_g that had to be computed until a feasible schedule was found.

The x-axis shows the probability of application of the mutation operators. M_1, M_2, and M_3 were tested in combination with M_6. M_4, M_5, and M_6 were tested in combination with M_1. The results were obtained with a population size p=50 and a fitness-function exponent E=2. The diagrams show the average over 100 runs. The search for a feasible solution was terminated after 200 generations. This was only necessary for the combinations M_3-M_6 and M_2-M_6. The best results were obtained from M_1-M_6 with P_1=0.6 and P_6=0.4. This combination finds a feasible schedule in typically 27 generations. These values were used in all further work.

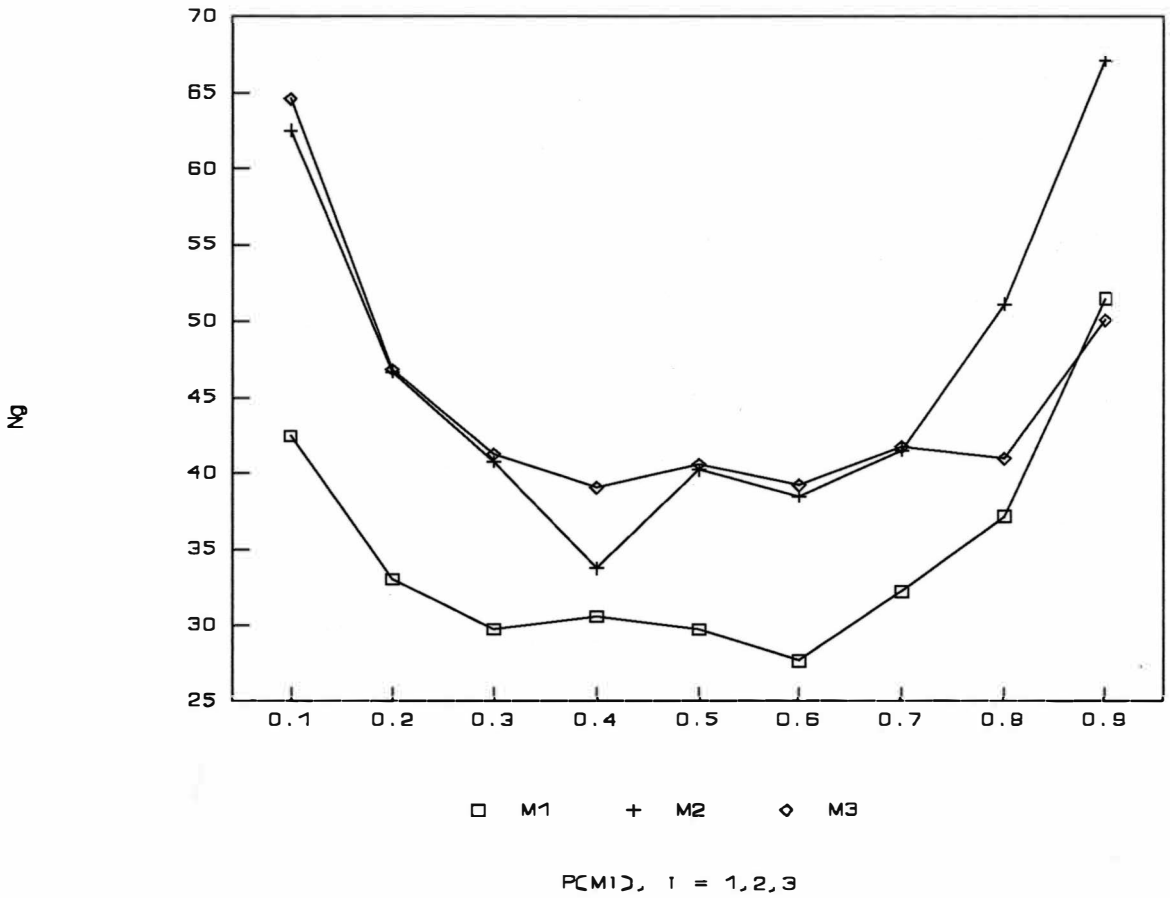

Figure 11. Performance of the Mutation Operators M1, M2, and M3 Against M6.

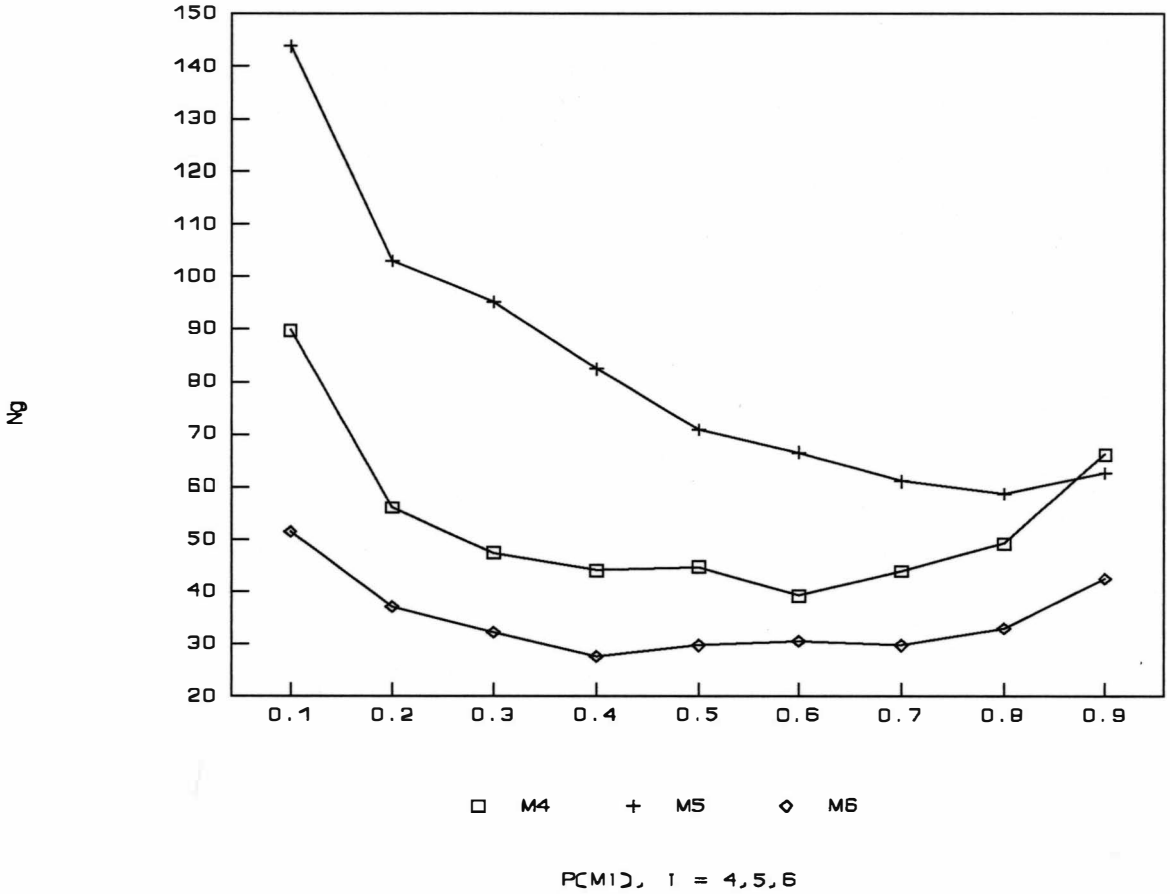

Figure 12. Performance of the Mutation Operators M4, M5, and M6 Against M1.

The results confirm the work of Manderick and Spiessens, who found that efficient genetic operators have a high correlation between the fitness of parent and offspring [16]. M_1 and M_6 only swap tasks that are close to each other in terms of completion time. This should introduce only a slight change in violation of deadlines and thus in fitness. The other operators exchange tasks without relation to the completion time.

Population Size

The influence of the population size μ on convergence is shown in Figures 13, 14, and 15. Figure 13 shows the average number of generations N_g to find a feasible schedule.

As expected, N_g decreases with increasing population size. The computational complexity depends on the total number of offsprings $N_O=N_g*\mu$, which is shown in Figure 14.

The diagrams show the average over R_t=100 runs. The search for a feasible schedule was terminated after N_{max}=200 generations. Figure 15 shows the probability $P_f=R_f/R_t$ for finding a feasible schedule where R_f is the number of runs that found a feasible schedule within N_{max} generations.

A small population converges faster, but small populations are more likely to converge to a local optimum, and do not always find a feasible schedule as shown in Figure 15. The parameter population size can be used to balance between computational complexity and accuracy of the solution. Large populations need more time to converge, but small populations may not always find feasible schedules if deadlines are tight.

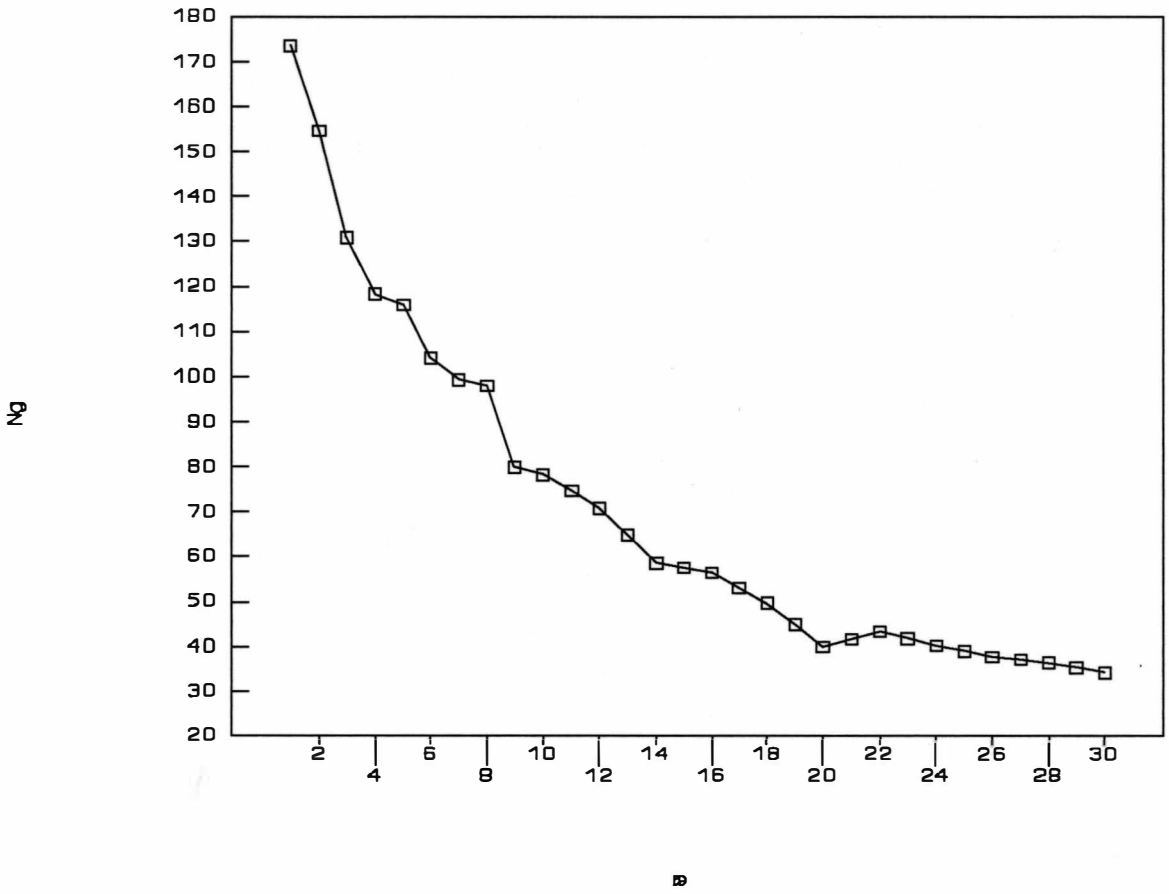

Figure 13. Number of Generations Over Population Size.

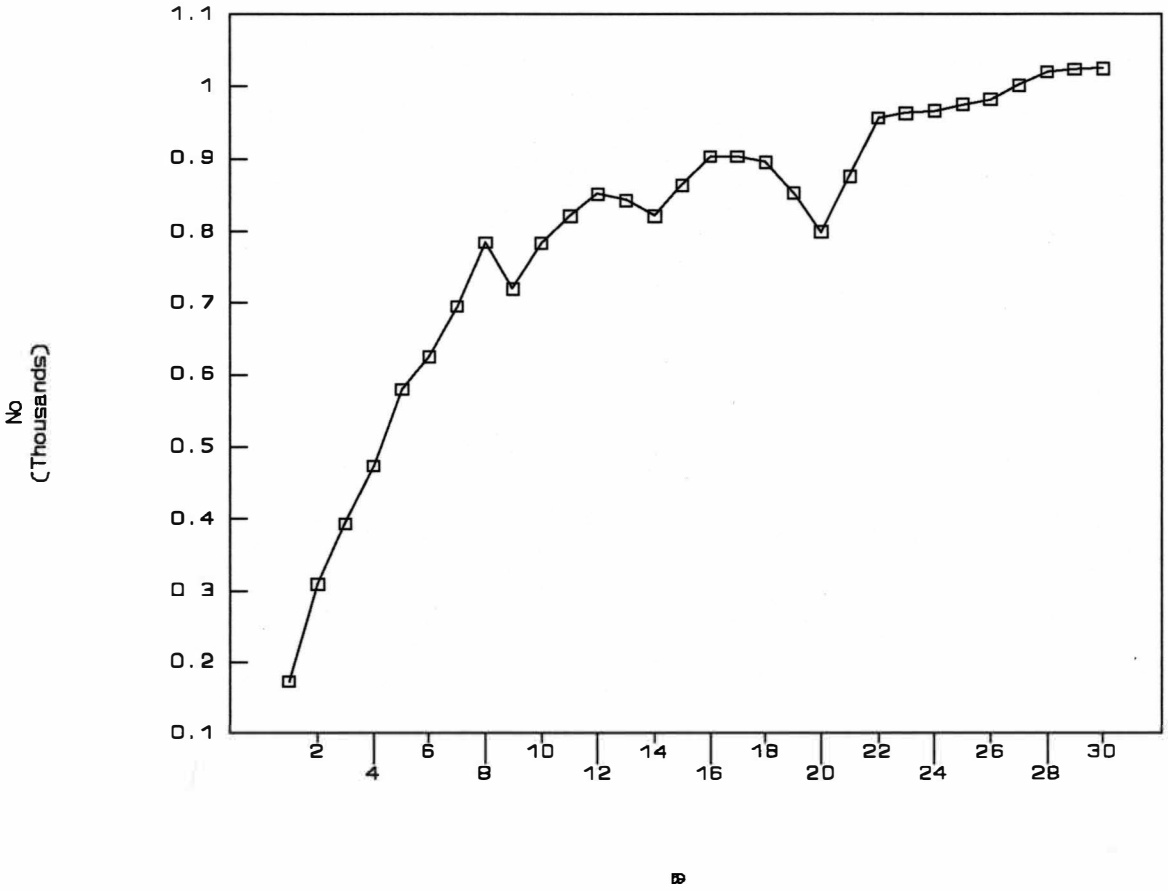

Figure 14. Number of Offsprings Over Population Size.

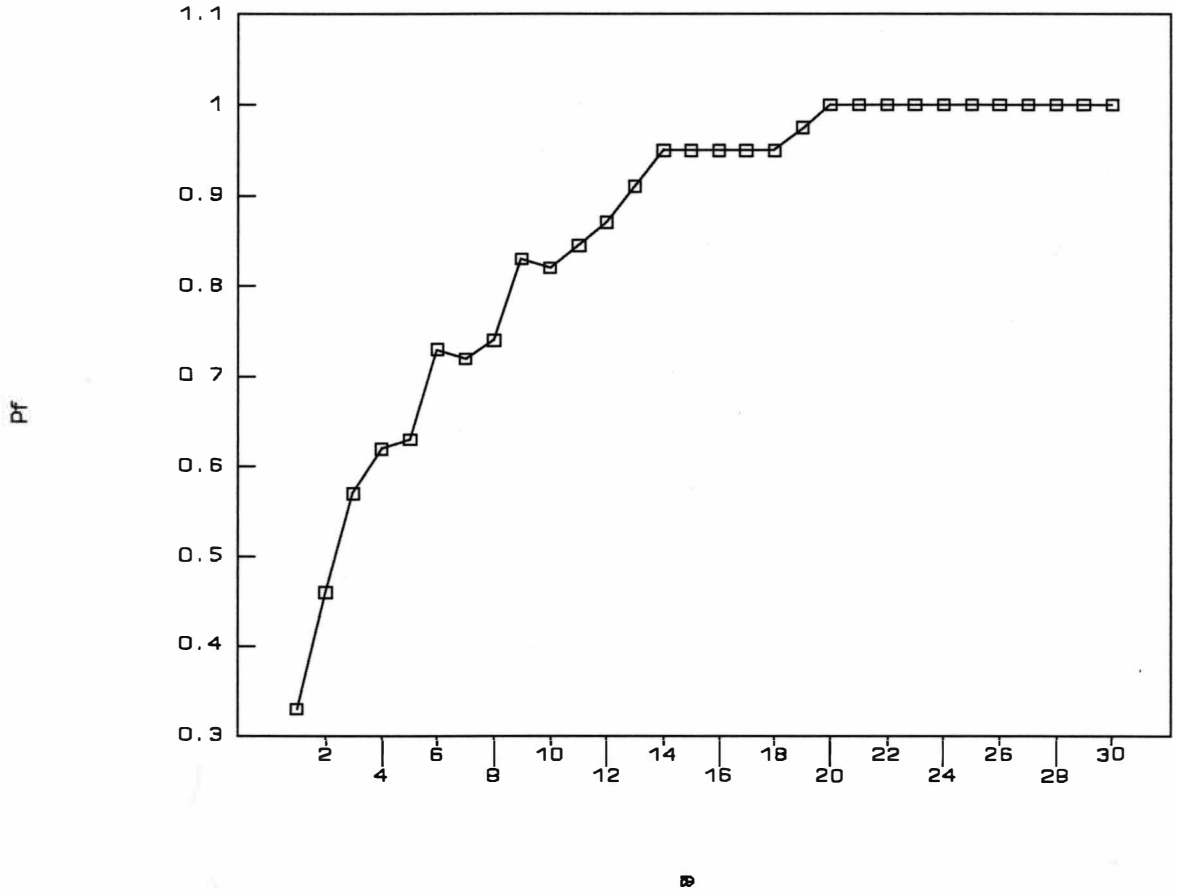

Figure 15. Probability for Finding a Feasible Schedule Over Population Size

Fitness-Function Exponent

The fitness-function exponent E in (1) (see page 30) affects the way, in which the solution is approached. If E is small, ES tends to satisfy the deadlines of many tasks at the expense of a large violation of the deadlines of a few tasks. A large exponent E leads to schedules where the deadline violations are more evenly distributed.

For example, compare the fitness F_1 and F_2 of two schedules S_1 and S_2. S_1 has 5 tasks that violate their deadlines by 1 unit of time. S_2 has one task that violates its deadline by 5 unit of time. All other tasks meet their deadlines. E=0.5 gives F_1=-5 and F_2=-2.2. E=2 gives F_1=-5 and F_2=-25. A fitness function with E=0.5 would select S_2, while E=2 would select S_1.

If only a few tasks violate their deadlines, most mutations do not affect the completion time of these tasks. As a result, progress in evolution is slow. With E<1, this point is reached earlier because solutions with a small number of violating tasks are preferred. For E>1 the amount of the large deadline violations is reduced rather than the number of violating tasks. Therefore, feasible solutions can be found faster if E>1. The average number of generations over E is shown in Figure 16. The results are the average over 200 runs.

Real-World Problems

The ES could find solutions for two real-world problems. The problem P_1 was introduced by Shaffer in [24]. The tasks of P_1 are the control program for a turbojet engine. The controller processes the inputs from 15 sensors, and computes the output for 5 actuators. The controller gets position feedback from each of the actuators. The program consists of 64 control procedures.

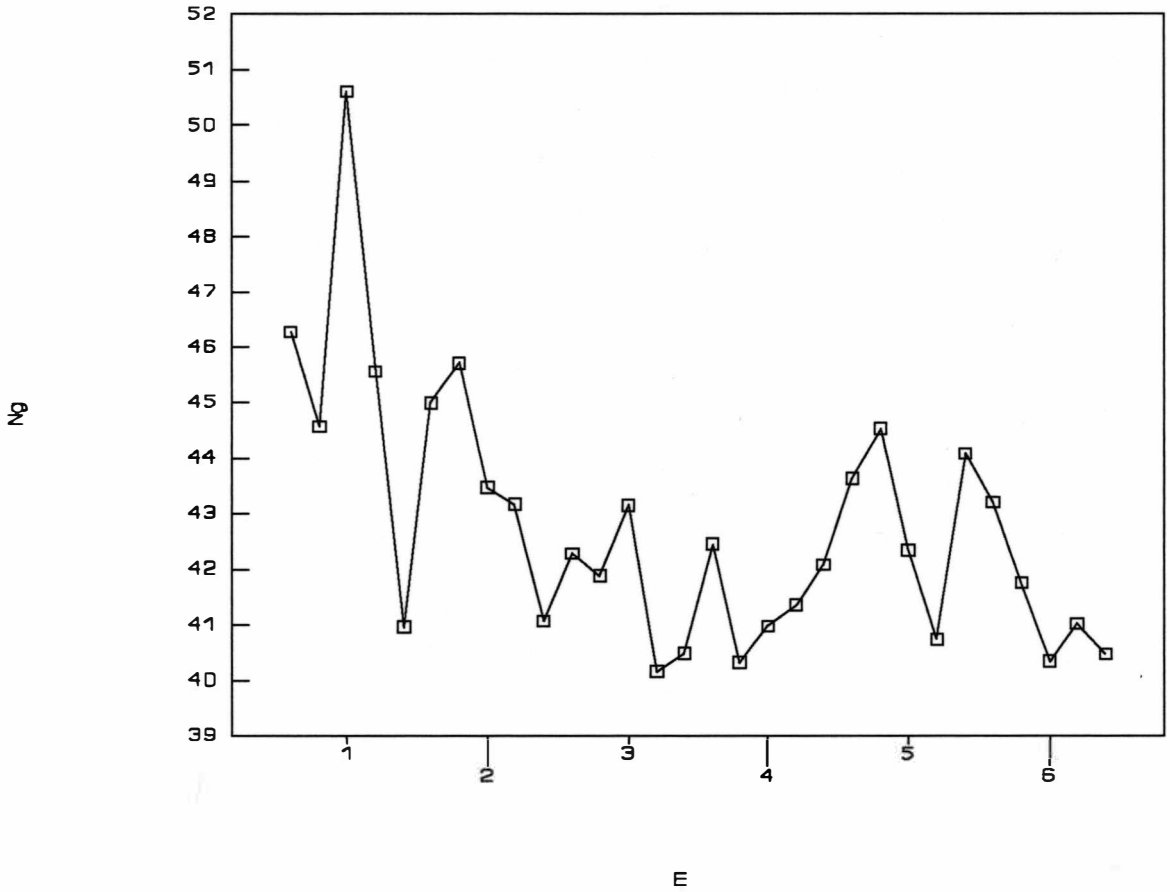

Figure 16. Number of Generations Over Fitness-Function Exponent.

The problem was partitioned so that each control procedure is one task. The PCG in Figure 17 is the result of a data-flow analysis. Each task is labeled with the maximum execution time in μs.

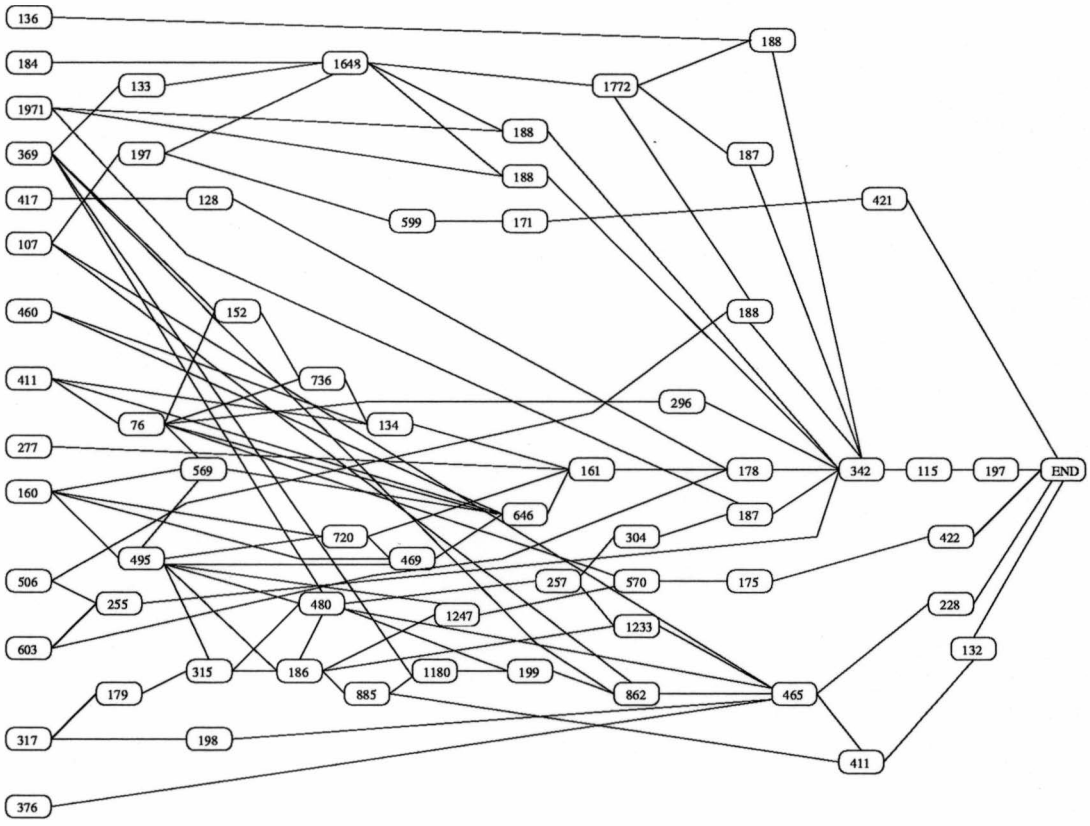

Figure 17. Precedence Constraint Graph of the Turbojet-Engine Controller.

The tasks were assumed to be executed on a multiprocessor with 4 processors where each processor had local RAM. The processors communicated by accessing global RAM over a bus. Shaffer found a near-optimal schedule by using a list-scheduling method. The schedule was optimized for minimum total execution time. A speedup of 3.94 was shown, which results in a processor utilization of 98%.

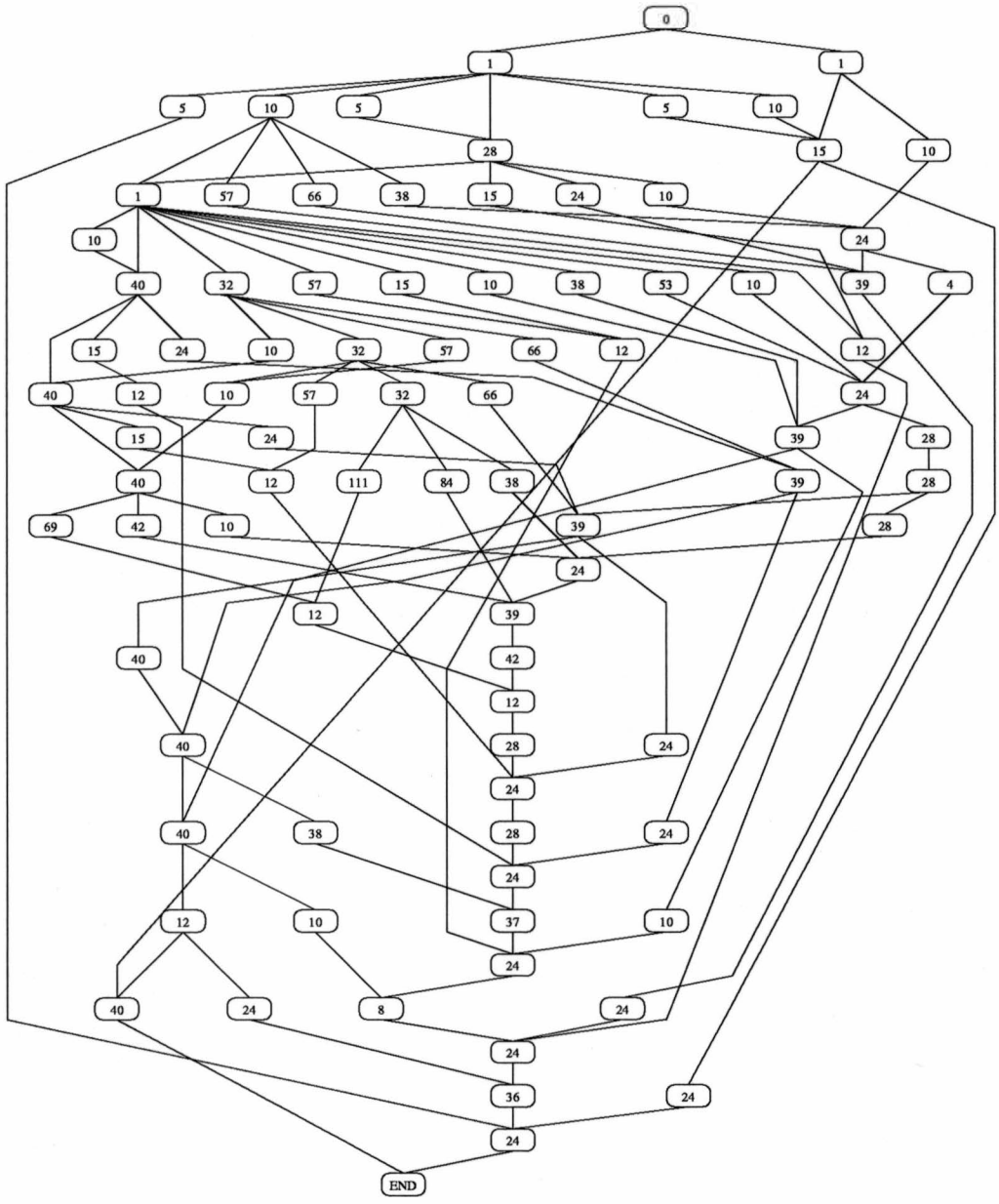

Figure 18. Precedence Constraint Graph of the Robot-Arm Controller.

Problem P_2 was introduced by Kasahara and Narita in [10]. The tasks of a robot-arm controller were to be scheduled on a multiprocessor. The authors used the Newton-Euler method to compute force and torque for each joint of a 6-joint robot arm.

The Newton-Euler method results in equations in 3x1 vectors and 3x3 matrices. Each task computes one equations. The data dependencies are shown in Figure 18.

Each task is labeled with the execution times in μs. The tasks were assumed to be executed on a multiprocessor with 6 processors. Each processor had local RAM attached. A bus allowed access to global memory. The authors found a schedule that was optimal with respect to total execution time by using depth first/implicit heuristic search (a branch-and-bound method). The total execution time equals the length of the critical path.

Since the original problems neglected communication delay and had no deadlines assigned to tasks, the problems were slightly changed. P_1 and P_2 were assumed to be executed on a multicomputer with 4 and 6 processing nodes respectively. They communicate by a ring topology. The message transmission delay was assumed to be 100μs for P_1 and 200μs for P_2. The schedules from [24] and [10] were simulated to find the completion time of the tasks. An instance of a real-time scheduling problem can be created from these schedules by taking the completion times of the tasks as their deadlines. Since the schedules did not show much idle processor time, these deadlines are near optimal. It is unlikely to find schedules that close to the optimal schedule. Therefore, the deadline t_{di} of a task T_i was multiplied with a delay factor D, $t_{di}=t_{ci}*D$. Solutions were found for D=1.2 in 8 minutes for P_1 and 54 minutes for P_2. The times are CPU time on a Sun SPARC IPC computer. A fitness-function exponent E=2 was used for both problems. The population size was μ=20 for P_1 and μ=100 for P_2. No solution could be found with μ=20 for problem P_2.

CHAPTER VII

CONCLUSION AND FURTHER WORK

Evolutionary strategy can be used find solutions to hard real-time scheduling problems in a small amount of time. Problems with up to 100 tasks can be solved in less than 60 minutes of CPU time on a Sun SPARC IPC. Event-driven simulation is a fast and flexible way to compute the fitness of schedules. The simulation can readily be adapted to other types of scheduling problems. Unlike many other algorithms, such as, shortest path methods, consideration of communication delay is not a problem. This makes ES especially suitable for distributed systems with a high communication/computation ratio.

It could be shown that the type of the genetic operators chosen has a significant influence on the efficiency of the ES. Good genetic operators introduce only small changes. By using information about the execution time of tasks from previous simulation, the performance of mutation operators could be improved. Population size is another major factor affecting the convergence of the ES. It seems difficult to determine the optimal population size before running the evolution. The optimal size of the population depends on the accuracy of the required solution with respect to the optimal solution, but the optimal solution is unknown. In general, more accurate solutions require larger populations. An open question is if the optimal population size can be estimated from other parameters, like amount of idle processor time or critical path length.

This work on ES for real-time scheduling can be extended in three basic directions: (1) parameters of the ES, (2) properties of the real-time scheduling problem, and (3) implementation of the ES on multicomputers.

Many more genetic operators can be investigated, for instance, no efficient recombination operator is known so far. The performance of new genetic operators cannot only be tested by running ES for sample problems, but also by statistical methods as proposed in [16]. Combinations of more than two operators with a fitness dependent probability may improve fine-tuning of a near-optimal solution. Other selection methods than truncation or rank-proportional selection may also prove to be useful.

ES can be applied to other types of real-time scheduling problems, such as, those which include both hard and soft deadlines. A weight is assigned to each task, and ES minimizes the total weight of the tasks that miss their deadlines, but with the constraint that all hard tasks must be scheduled.

Network traffic can also be optimized by ES. Instead of routing a message along a minimum-distance path, the genotype is extended by a path and a priority for each message. This can be of advantage for dense networks, such as, hyper-cube architectures, with many short paths between processing nodes.

Since ES exhibits a high potential of parallelism, an implementation on a multicomputer may give high speedup. Most of the computation time is needed for the simulation of the schedules. This part can be done in parallel until the number of processing nodes reaches the population size. New selection methods that have only partial knowledge about the population are necessary.

BIBLIOGRAPHY

[1] Biyabani, S. R., Stankovic, J. A., and Ramamritham, K., "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling", *Proceedings of the IEEE Real-time Systems Symposium*, pp. 152-160, Dec. 1988.

[2] Chu, W. W., Holloway, L. J., Lan, and M., Efe, K., "Task Allocation in Distributed Data Processing", *IEEE Computer*, pp. 57-69, Nov. 1980.

[3] DeJong, K. A., "Genetic Algorithms: A 25 Year Perspective", in *Computational Intelligence Imitating Life*, edited by Zurada, J. M., Marks II, R. J., and Robinson, C. J., *IEEE Press*, New York, pp. 125-134, 1994.

[4] Garey, M. R. and Johnson, D. S., "*Computer and Intractability: A Guide to the Theory of NP-Completeness*", San Francisco, CA: Freeman, 1979.

[5] Goldberg, D. E., "Sizing Populations for Serial and Parallel Genetic Algorithms", in *Genetic Algorithms*, edited by Buckles, B. and Petry, F., *IEEE Computer Society Press*, pp. 20-29, 1992.

[6] Gonzalez, M. J., Jr., "Deterministic Processor Scheduling", *ACM Computing Surveys,* vol. 9, no. 3, pp. 173-204, Sep. 1977.

[7] Greenwood, G. W., Gupta, A., and Mahadik, V., "Multiprocessor Scheduling of High Concurrency Algorithms", *Proceedings of the Florida AI Research Symposium*, pp. 265-269, 1994.

[8] Hou, E. S.H., Ansari, N., and Ren, H., "A Genetic Algorithm for Multiprocessor Scheduling", IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 2, pp. 113-120, Feb. 1994.

[9] Huang, J. P., "Modeling of Software Partition for Distributed Real-Time Applications", *IEEE Transactions on Software Engineering*, pp. 1113-1126, Oct. 1985.

[10] Kasahara, H. and Narita, S., "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System", *IEEE Journal of Robotics and Automation*, vol. RA-1, no. 2, pp. 104-113, Jun. 1985.

[11] Kasahara, H. and Narita, S., "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", *IEEE Transactions on Computers*, vol. c-33, no. 11, Nov. 1984.

[12] Khemka, A. and Shyamasudar, "Multiprocessor Scheduling of Periodic Tasks in a Hard Real-Time Environment", *Proceedings of the 6th International Parallel Processing Symposium*, 1992.

[13] Leighton, F. T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kaufman Publishers Inc., 1992.

[14] Liu, C. L. and Layland J. W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[15] Liu, J. W. S., Lin K.-J., Shih, W.-K., and Yu, A. C., "Algorithms for Scheduling Imprecise Computations", *IEEE Computer*, pp. 58-68, May 1991.

[16] Manderick, B. and Spiessens, P., "How to Select Genetic Operators for Combinatorial Optimization Problems by Analyzing their Fitness Landscape", in *Computational Intelligence Imitating Life*, edited by Zurada, J. M., Marks II, R. J., and Robinson, C. J., *IEEE Press*, New York, pp. 170-181, 1994.

[17] Michalewicz Z., *Genetic Algorithms+Data Structures=Evolution Programs*, 2nd ed., Springer Verlag, 1994.

[18] Mühlenbein, H. and Schlierkamp-Voosen, D., "Theory and Application of the Breeder Genetic Algorithm", in *Computational Intelligence Imitating Life*, edited by Zurada, J. M., Marks II, R. J., and Robinson, C. J., *IEEE Press*, New York, pp. 182-193, 1994.

[19] Ramamritham, K., "Allocation and Scheduling of Complex Periodic Tasks", *Technical Report 90-01*, University of Massachusetts, Jan. 1989.

[20] Ramamritham, K., Fohler, G., and Adan, J. M., "Issues in the Static Allocation and Scheduling of Complex Periodic Tasks", *Proceedings of the RTOSS*, pp. 11-16, 1993.

[21] Rechenberg, I., "Evolution Strategy", in *Computational Intelligence Imitating Life*, edited by Zurada, J. M., Marks II, R. J., and Robinson, C. J., *IEEE Press*, New York, pp. 147-159, 1994.

[22] Schwefel, H.-P., "On the Evolution of Evolutionary Computation", in *Computational Intelligence Imitating Life*, edited by Zurada, J. M., Marks II, R. J., and Robinson, C. J., *IEEE Press*, New York, pp. 116-124, 1994.

[23] Sha, L. and Sathaye, S. S., "A Systematic Approach to Designing Distributed Real-Time Systems", *IEEE Computer*, pp. 68-78, Sep. 1993.

[24] Shaffer, P. L., "A Multiprocessor Implementation of Real-Time Control for a Turbojet Engine", *Proceedings of the American Control Conference*, 1989.

[25] Shin, K. G. and Chen, M., "On the Number of Acceptable Task Assignments in Distributed Systems", *IEEE Transactions on Computers*, vol. 39, pp. 99-110, Jan. 1990.

[26] Stankovic, J. A., "A Serious Problem for Next-Generation Systems", *IEEE Computer*, pp. 10-18, Oct. 1988.