Order Number 9510660

Techniques to improve the performance of mutation analysis

Fleyshgakker, Vladimir Nathan, Ph.D.

City University of New York, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

TECHNIQUES TO IMPROVE THE PERFORMANCE OF MUTATION ANALYSIS

by

VLADIMIR NATHAN FLEYSHGAKKER

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1994

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

Date: 9.26.94

Chair of Examining Committee

Date: 9/26/94

Executive Officer

Professor P. Frankl

Professor T.C. Wesselkamper

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

TECHNIQUES TO IMPROVE THE PERFORMANCE OF MUTATION ANALYSIS

by

VLADIMIR NATHAN FLEYSHGAKKER

Adviser: Professor Stewart N. Weiss

Although mutation analysis is a potentially effective method of software test data evaluation, it is computationally intensive and time-consuming. Existing algorithms are not very efficient, failing to eliminate redundant computations. We present a universal mutation analysis data structure and new serial algorithms for both strong and weak mutation analysis that on average perform much faster than existing ones, and can never do worse. We describe these algorithms and analyze their run time complexities as well as the complexities of existing algorithms.

We also present a novel, special purpose, dynamically reconfigurable multiple-SIMD architecture designed for mutation analysis. We describe the architecture and its use for mutation analysis. Our performance analysis indicates that the utilization factor of the processing elements and the potential speed-up are impressive. The results of our software simulation corroborate the theoretical analysis. On our architecture mutation analysis is a feasible method of evaluating software test data.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to Professor Stewart N. Weiss who was kind enough to work with me as dissertation adviser. I would also be remiss if I failed to acknowledge my debt to Professor Weiss, first of all for his assistance in choosing the extremely timely and rewarding topic of performance improvement of mutation analysis, and secondly for his constant technical guidance.

Table of Contents

List of Tables

List of Figures

1. Introduction

Today, when computer programs control so much of vital importance from nuclear weapons to life support systems, the problem of software correctness and, therefore, software testing is of paramount importance. This dissertation addresses improving the time performance of an important method of software testing, namely, mutation analysis by proposing new algorithms and a new special purpose parallel architecture.

Mutation analysis is a method of evaluating the adequacy of a set of test data for a program. Informally, test data are considered *mutation-adequate* for a program if they can distinguish the program from programs that differ from it by small syntactic changes. Although mutation analysis is also the basis for a test data *selection* criterion, in this research we are concerned only with its use as an adequacy criterion.

The fundamental questions that must be asked about any proposed method of test data selection or evaluation are how effective the method is at exposing faults in the tested software, and how much effort is required to use it. Roughly speaking, the effectiveness of an adequacy criterion is the probability that a program is correct, given that an arbitrary test set that satisfies the criterion exposed no faults in the software. Many researchers believe that mutation analysis is very effective at exposing software faults, and there have been some studies that support this [8,31,48]. Although it has not been proved irrefutably that mutation analysis is an effective test data adequacy criterion, the theoretical arguments and the experimental evidence suggest that it may very well be a powerful technique, and is therefore a topic of importance to the software testing community.

The focus of this work is not about the effectiveness of mutation, but about the effort required to use it for test data evaluation. Few researchers deny that mutation is costly to use, but little is known about how much effort is really required to use it in practice, either in a worst case or expected case. In fact, it is not entirely clear how to measure the cost of using mutation analysis as an adequacy, or even a selection criterion. Clearly, both test set size and the number of mutants of the subject program *may* contribute to this cost, but how much each contributes depends upon many factors. For example, if all of the mutants of a program are distinguished from the original program (i.e., "killed") by the first few test cases evaluated, then the remaining test cases need not be evaluated, and the size of the test set does not influence cost. On the other hand, all of the mutants might need to be executed on all of the test cases, and the cost is then proportional to the test set size and to the number of mutants. The picture is further complicated by the fact that some mutants might be semantically equivalent to the original program, making it more difficult to decide whether a test set is mutation-adequate for the program. While the general question of how to measure the cost of using test data selection and evaluation criteria has been raised and studied somewhat [62,66], it has yet to be answered satisfactorily. For mutation analysis it is a particularly complex question. In this work, we are concerned about just one aspect of this cost, namely the run time complexity of performing mutation analysis.

The potentially high cost of performing mutation analysis has led researchers to develop parallel mutation systems for various multiprocessor architectures[14, 27, 37, 67]. Many of the papers describing these parallel systems make claims about the "speed-up" they achieved. But in our opinion, the concept of speed-up had been applied

incorrectly in the first place.

To say that a parallel algorithm attains a speed-up β in solving a problem means that the parallel algorithm is β times faster than *the fastest known serial algorithm* for that problem[53]. However, no one has yet established a lower bound on the time complexity of serial mutation analysis. The articles that describe parallel mutation systems compare the running times of their systems to the sum of the running times of each mutant executing each test separately. Implicitly then, they compare their systems to a rather naive method of doing mutation, in which each mutant is executed on each test until either the mutant is killed or all tests have been tried. The running time of this *naive algorithm* is, in the worst case, simply the sum of the running times of all mutants on all tests, which is not necessarily the best possible serial running time. Thus, speed-up was measured not against the *best* serial algorithm, but against a naive, possibly *worst*, serial algorithm, akin to comparing a parallel sort to an $O(n^2)$ sort such as bubble sort.

To determine the speed-up attained in a parallel mutation system, one first needs to know the best one can do serially. However, we do not plan to establish the lower complexity bounds of mutation analysis in this work. The goal of this thesis is twofold:

(1) to develop faster serial algorithms for mutation analysis and analyze their time complexity, and

(2) to design a novel, special purpose, dynamically reconfigurable, multiple-SIMD architecture, called HiTest, to be used for parallelization of the most time consuming steps of our strong mutation serial algorithm.

2. Background

2.1. Preliminaries

By program P we mean a sequence of statements that agrees with the syntax of some given programming language. We assume that P can be decomposed in a set $B_1, B_2, \cdots, B_K$ of nonintersectable *basic blocks*. Each basic block is a maximal subsequence of statements from P such that flow of control enters only at the first statement and leaves only at the last statement of this subsequence[4]. Each block ends in either an *unconditional* or a *conditional* transfer of control. An unconditional transfer passes control to the same block regardless of the program state; a conditional transfer passes control to the first statement of one of two possible successor blocks, depending on the evaluated condition.

A program P is considered to be correct if its output $P(t)$ agrees with the specification of P for any input t from the input domain of P. A *test case* or simply *test* t is an input used to run a program P, and the output $P(t)$ is assumed to agree with a specification of P (if $P(t)$ does not agree with the specification of P then P is proved to be incorrect, and there is no need for the further testing). The set T of all tests is called a *test set*. We treat test set T as an *sequence* of tests $t_1, t_2, \cdots, t_{|T|}$, although no particular ordering is assumed.

2.2. Mutation Analysis

A mutation operator is applied to a program to produce a new program, called a *first-order mutant* [23], differing from the original by a single, small syntactic change. Mutants can also be obtained by applying multiple syntactic changes to the original program, and are referred to as *higher-order mutants*. In this work we restrict our attention to first-order mutants, which we henceforth refer to simply as mutants. An example of a mutation operator is the *variable replacement operator*, which when applied to a variable reference replaces it by a different program variable, subject to the constraint that the resulting program be syntactically correct. Other mutation operators do such things as replace relational or arithmetic operators, or replace constants. The *Mothra* mutation system, for example, implements 22 different mutation operators [18].

Given a fixed set of mutation operators M and a program P, we can methodically use these mutation operators to generate the set $M(P)$ of all possible mutants that differ from P by applying each operator in all possible ways. Some of these mutants, of course, may be *semantically equivalent* to program P, that is they produce the same output as the original program on any input. Formally, the mutant $E \in M(P)$ is considered equivalent to P if for every input τ belonging to the input domain of P, $E(\tau) = P(\tau)$. For a fixed set of mutation operators M, we say that test set T is *mutation adequate* for program P if, for every mutant $Q \in M(P)$ that is not equivalent to P, there is a test $t \in T$ such that the program and the mutant produce different output or $Q(t) \neq P(t)$.

Two premises form the rationale for mutation adequacy: the *competent programmer hypothesis* and the *coupling effect* [23]. The competent programmer hypothesis, simply put, is that most faults in a program P are relatively simple discrepancies between P and

a correct version of P. The coupling effect asserts that test data that detect small changes in a program are sensitive enough to detect more complex errors. A few empirical studies have been done to investigate one or both of these assumptions [40, 44, 50], but, as Marick [44] puts it, "...there is yet no evidence against the technique ... nor for it" and more work needs to be done.

Particular sets of mutation operators have been proposed, studied, and implemented for FORTRAN [12, 23] and C [3]. The implementation of some of these mutation systems on multiprocessor architectures is the topic of Section 2.3.

The number of mutants of a single program can be quite large. With some sets of mutation operators, the number of mutants can even be an exponential function of the size of program [8]. The most commonly cited bound on this number is $v \cdot w$, where v is the number of variables in the program and w is the number of variable references in the program [1, 8, 18, 34]. This bound can be achieved when the variable replacement operator is used. Since the number of variable references is proportional to program size, this bound is roughly $v \cdot n$, where n is program size. Whatever the exact bound may be, there is little dispute that mutation analysis can be prohibitively time-consuming for large or even moderate size programs, since one must determine whether or not each unequivalent mutant is killed by some test in the test set. The most obvious way of doing this is to run each inequivalent mutant on each test until it is killed or has been executed on all tests. This method presupposes that one has already decided which mutants are inequivalent to the original program, itself not a trivial task.

2.3. Parallel Mutation Systems.

Given that large scale parallel computers are becoming less expensive and more powerful, an obvious solution is to run all of the mutants on an MIMD architecture with as many processors as possible, one mutant per processor, each in its own address space.

This is the approach described, for example, by Choi, Mathur, and Pattison [15]. Their system, called PMothra, is based on the Mothra mutation system, but uses a 128-node Ncube/7 hypercube MIMD multiprocessor. The hardware environment of the system also includes the host (Sun workstation). While running on the host the scheduler schedules the mutants on the next available node of the hypercube. The idea of this is to run as many mutants as possible on the different nodes of the multiprocessor.

Although appealing for its simplicity and potential speed-up, it is not necessarily the most cost effective solution. MIMD machines, and the Hypercube in particular, are very expensive because their architecture includes either a large shared memory and complex synchronization hardware, or a collection of separate processor-memory pairs connected by a complex interconnection network. Running a large number of mutants in parallel on this architecture makes no significant use of either the interconnection network or the shared memory. Given that these architectures are costly to build primarily because of these features, they are a poor choice for mutation analysis. Simply put, even though a significant speed-up is possible on an MIMD machine, the architecture is not well utilized and the speed-up per unit cost may not be optimal.

An alternative is to capitalize on the fact that mutants are nearly identical programs, differing by a single token, and that therefore much of their collective execution involves repeatedly executing the same code sections with different data. This suggests that a

vector processor might be suitable for high performance mutation analysis, which is the approach reported in [37]. There the authors achieve large speed-ups for a few small mutated programs, but our analysis shows that their method cannot in general provide significant speed-up for programs of moderate or large size. The problem, as was noted in [37], is that once the mutated statement is executed, the mutants' execution paths may not proceed in unison or even close to it; the basic block to be executed will then be different for most mutants, and the processing elements (PEs) will spend too much time in an idle state. In short, the potential concurrency of many mutants executing in parallel may not be very great for realistic programs. We discuss this in Section 2.5.

Wilsey et al [67] arrived at the same conclusion and developed another mutation system for SIMD computers. They used the MasPar MP-1 processor and a non-optimizing FORTRAN compiler. The main approach of their work is to place compiled programs into the local memory of each PE. A local program counter is also initialized within each PE and the instructions are interpreted in parallel across all the PEs by control signals from the central control units. Their analysis shows that the speed-up should increase linearly with the number of PEs. This is a promising result, indeed. However, we should mention that it is quite unusual for processing elements in a SIMD architecture to have enough local memory to store compiled code. In fact, this lack of memory is one of the main reasons to have a host, the center processor that has a compiled program in its memory and transmits the instructions to the PEs.

2.4. Strong Versus Weak Mutation

Many researchers have investigated a weaker notion of mutation adequacy

appropriately called *weak mutation analysis*. [32, 35]. Whereas in ordinary mutation analysis, which is also called *strong mutation analysis*, a mutant M is considered killed by a test t only if the output $M(t)$ is different from $P(t)$, in weak mutation analysis, a mutant is considered killed by t if the program state of M after some execution of the mutated statement is different from the program state of P at that same point.

It stands to reason that weak mutation adequacy is easier to check and is probably less costly to use than strong mutation adequacy, since it is not necessary to execute a mutant completely, but only to the first point at which its internal state differs from that of the original program. In fact, because the states of a mutant and the original program can diverge only at the mutated statement, a more efficient approach is to execute the original program alone, saving its state each time it reaches the mutated statement, and executing each mutant of that statement in that state only. This idea is not new; Howden suggested something like this in his seminal paper on the subject [35] and Offutt and Lee [51] also mentioned the possibility of implementing weak mutation using this idea, borrowing the term "split-stream execution" from [36]. It has also been suggested that constructing mutants is unnecessary in weak mutation analysis because in principle the constraints on the inputs necessary to kill mutants weakly can be derived automatically [29, 51]. Some researchers have investigated systems that try to derive such constraints [59] and others have investigated theoretical issues related to these constraints [47].

Several weak mutation systems have been implemented [29, 32, 46, 51] although not all have used the more efficient methods mentioned above. For example, *Leonardo* [51] performs weak mutation by executing each mutant separately from its initial statement until a strategic "compare" point at which the mutant and program states are

compared. We refer to this method as the *naive* weak mutation algorithm, in contrast to the more efficient implementations such as GCT [46], which are closer in concept to the idea described by Howden and Offutt and Lee.

It would be ideal if weak mutation were as effective as strong mutation in exposing faults in software, since it is so much easier to implement weak mutation. Questions about how effective weak mutation analysis is, and whether or not there is a correlation between a test set's weak mutation adequacy and its strong mutation adequacy have interested many researchers [30, 34, 45, 51]. The *weak mutation hypothesis* [44, 45] is that a test case that weakly kills a mutant strongly kills it also. Marick provides reasonably strong statistical evidence that the weak mutation hypothesis is often true, and Horgan and Mathur use probabilistic arguments to justify it [34]. Further studies must be done however before any conclusions can be made about the correctness of this hypothesis.

2.5. Using SIMD Machines for Mutation Analysis

An interesting attempt to decrease the cost of mutation analysis by using multiprocessors was made by Krauser, Mathur, and Rego [37]. In their work, the possibility of high performance mutation analysis was evaluated on an abstract SIMD machine that was intended to model commercially available non-partitionable SIMD machines such as the Connection Machine [31]. The basic architecture of these vector-multiprocessors is as follows: Many identical processing elements are controlled by a single control unit referred to as the *host*. Each PE has a local memory which is used to hold the data on which the PE operates. The program to be executed resides in the memory associated

with and accessible only to the host. The host broadcasts the instruction to be executed to all PEs simultaneously but each PE executes it with its own local operands. Each PE executes its own mutant. It can be in one of two possible states at any time: *enabled* and *disabled*. In the enabled state, a PE executes the instruction sent by the host. In the disabled state, a PE is prevented from executing any instructions of the mutated program.

Initially, the host broadcasts instructions of the first block of the program to all PEs, each of which computes with its own local data. When a block is completed, the successor blocks are computed for all PEs. Until a mutated statement is executed, the state of all PEs is the same and the PEs execute with maximal concurrency. However, after the mutated statements are executed the states of the PEs may be different, and the PEs in general require different successor blocks. It is then up to the host to choose which successor block to execute next. Any PE whose computed successor disagrees with the one chosen by the host disables itself until the host is ready to execute a new block. Thus, at any given time, only a subset of the PEs is actually executing.

The basic block containing the mutated statement is treated as three separate blocks: (1) the *prefix* of the block preceding the mutated statement, (2) the mutated statement itself, and (3) the *suffix* of the block following the mutated statement. The mutated statement is special because it must be executed serially.

From the preceding description it is evident that a significant fraction of the PEs may be idle at a given time. In the absence of a particular strategy for the host to use to select the next block for execution, we may suppose that the average number of enabled PEs is M/K, where M is the number of PEs in the vector-processor and K is the number of basic blocks in the program. The average number of disabled processing elements is

$M(1-1/K)$. Therefore, when K is very large, as is the case for most production quality programs, the absolute majority of PEs are disabled. In other words, a simple analysis suggests that the hardware may be used very inefficiently for large programs.

In the same paper [37] the authors report a good speed-up for weak mutation analysis, but they used a very poor serial weak mutation algorithm for comparison to obtain their speed-up. They just run every mutant on each test case until the mutant is weakly killed or terminated. It is easy to see that every mutant follows the same trace as the original program on the same input except for the last state, in the case when mutant is weakly killed. It is obvious that whole execution except for executions of the mutated statement is redundant, but the executions of the mutated statements are done serially in described parallel system[37] .

3. Improved Algorithms for Mutation Analysis and Their Complexity

In exploring the literature we did not find any research whose primary goal was to define fast serial algorithms for mutation analysis. We found mutation systems that provide compiler support for mutation by improving compilation, load, and even execution time of mutants [21, 22], but these do not use different algorithms for checking mutation adequacy. Furthermore, we did not find any research in which the run time complexity of any known mutation algorithm was analyzed.

Thus, the goal of this section is twofold: (1) to develop faster serial algorithms for mutation analysis, and (2) to analyze the time complexity of these algorithms.

3.1. Description of the Data Structure

Before the mutation analysis algorithm can be applied to the subject program, for each test case in the test set it is necessary to construct a data structure that captures all important information about the execution of the program on that test case. We call this data structure a *Mutant-DS*, or an MDS for short. The MDS for an arbitrary input t_i (see Figure 3.1) consists of two parts: an array $\boldsymbol{I}$ representing the program and a list of program states $\boldsymbol{S}$ representing the trace of the program on the given input. The array $\boldsymbol{I}$ contains an element for each statement of the program. Whereas the term ''statement of a program'' can be used to mean a structured statement in a high-level language, such as a while-statement, we mean a lower level execution unit. We conceptualize the program as having been translated into intermediate code, with a ''statement'' as an instruction at this level of abstraction. The elements of the array $\boldsymbol{I}$ are in the same order as the corresponding instructions of the program.

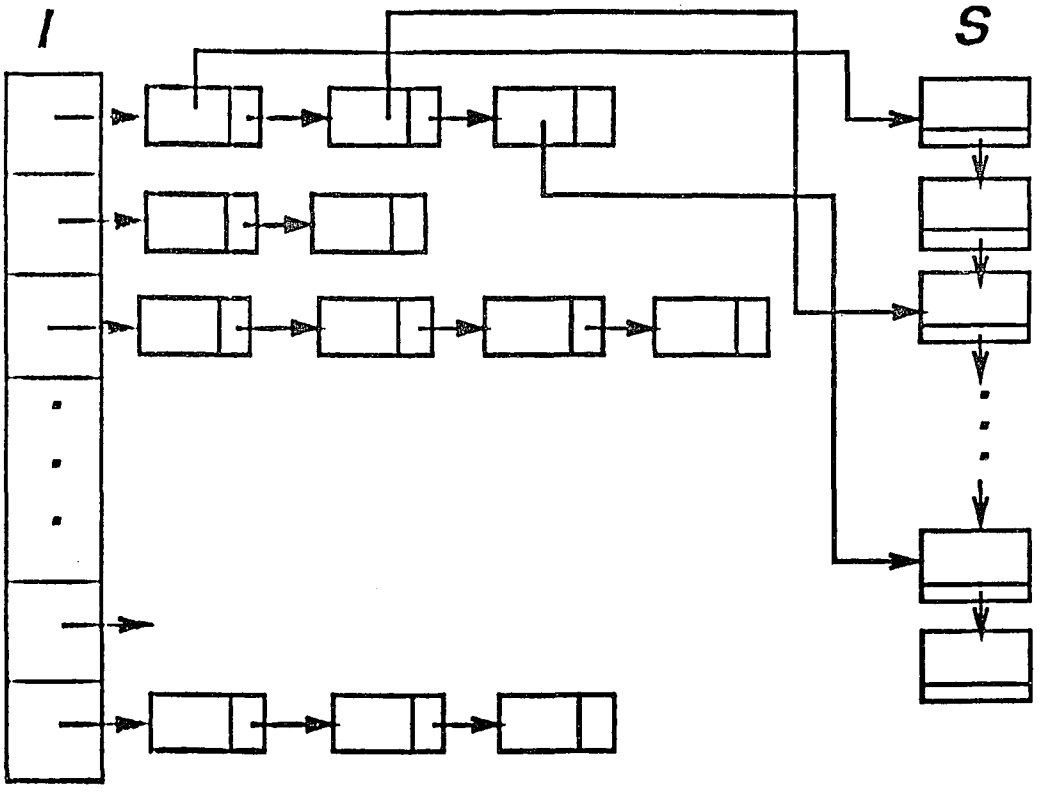

Figure 3.1. The Data Structure.

The trace S is a list, $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_N$, of the execution states visited during execution of the program. A "state" in this model is at the same level of abstraction as a program statement, in the sense that, if the program is in state σ_j and the instruction about to be executed in state σ_j is I_k, then the successor state $\sigma_{(j+1)}$ of σ_j in the trace S is the one that would be obtained by executing instruction I_k in state σ_j. A program state contains information about the values of variables at that point in the execution. Such a list can take up a large amount of storage. In Section 3.4. we describe modifications of the data structure that decrease space consumption substantially while increasing running time only slightly.

Each element of the array I contains a pointer to a linked list of references of the corresponding statement in trace S. Each time that the kth instruction I_k is reached during execution of the program on the given test case, a node is appended to the end of the linked list attached to array element $I[k]$. This node is a pointer to the state in the trace in which statement I_k is about to be executed. If I_k is never reached during execution of the program on this test case, then $I[k]$ will be *NIL*. The following observations follow readily:

1. The length of the list of nodes attached to $I[k]$ is the number of times that statement I_k was executed in the trace.
2. The order of the nodes in this list is the order in which the respective executions of I_k took place during execution of the program on the test case.

It is important to realize that creating this data structure is a trivial problem that does not add much to the running time of mutation analysis. In order to generate MDSs

the program has to be instrumented as follows:

1. At the very beginning of the program prior to execution of any instructions, a routine that generates an empty data structure is called. This routine puts NIL in every entry of array $\boldsymbol{I}$ and sets trace $\boldsymbol{S}$ to NIL also.

2. Each instruction of the program has to be replaced by a block consisting of the following actions:

   - make a copy of the current state of the program and put it at the end of the trace $\boldsymbol{S}$;
   - append a node to the end of the list attached to $\boldsymbol{I}[k]$;
   - make the pointer in this node point to the last state in the trace $\boldsymbol{S}$;
   - execute instruction I_k.

The program has to be instrumented once for the entire test set, and then for each test the data structure is generated automatically by running an instrumented version of the program on this test. The time needed to create MDSs, neglecting the cost of instrumentation of the program (which should be no greater than the time of a compilation), is just a constant multiple of the program running time on the entire test set. Because this time is very small in comparison to the time needed for mutation analysis (each mutant might be run on the entire test set, so one extra run is negligible), and because the program must be run on each test anyway, we discount it in the run time complexity analyses in the sequel.

Henceforth we write MDS[i] to mean the MDS of the test t_i. The trace of this MDS is denoted MDS[i].$\boldsymbol{S}$ and its instruction array is denoted MDS[i].$\boldsymbol{I}$. Thus, MDS[i].$\boldsymbol{I}$[j] is a pointer to the linked list attached to the jth entry of the $\boldsymbol{I}$ array of the MDS of test t_i.

Space Complexity

If $|S|$ represents the length of trace S and $|I|$ is the number of instructions in the program, then the space consumption of each MDS is $O(|S|+|I|)$, since the sum of the lengths of all linked lists attached to the array I is the same as the length of S.

3.2. Algorithms for Mutation Analysis

The main purpose of the MDS is to facilitate deciding whether or not a test set satisfies strong mutation adequacy for a program. It can also be used to decide whether the test set satisfies weak mutation adequacy. Because a mutant is strongly killed only if it is weakly killed, and is weakly killed only if the mutated statement has been executed by the test data (called the *reachability condition*), and because it is generally easier to determine whether the weaker statement is true than the stronger one, our general approach to deciding whether a given level of adequacy is achieved is to check whether the lower level is achieved. If so, we use information obtained in making that decision to solve the harder problem. For example, to decide whether or not a mutant is strongly killed by a test, we first check whether it is weakly killed. If so, we then check whether it is strongly killed.

The MDS can also be used to answer other questions relating to test data coverage. For example, from the two observations in Section 3.1 it follows that the MDS can be used to decide whether or not a particular statement or basic block has been executed by test data. Algorithms to do this are straightforward and are not presented here.

This section consists of five subsections. The first contains preliminary definitions, notation, and analysis, as well as the basic assumptions underlying this analysis. The

next summarizes the interrelationships among the algorithms, and the last three sections are descriptions of the mutation analysis algorithms. Each algorithm is followed immediately by a performance analysis of that algorithm.

3.2.1. Preliminaries

For the analysis of time complexity that follow we need to make a few assumptions. For one, we assume that the execution time of each instruction of the program is the same, and we treat that amount of time as a single time unit in the analysis. Under this assumption, the running time of the program on a test case t is the length of the trace $\boldsymbol{S}$ of t. To simplify the performance analysis, we further assume that the average running time of all mutants of a given program is the same as that of the program itself. We do not attempt to justify this assumption here; clearly one can find programs and test sets for which the average mutant running time (over the tests in the test set) is greater than that of the program, and one can find programs and test sets for which it is less. In the absence of further information about the subject program, this is a reasonable assumption to make, and it allows us to obtain a rough estimate of the run-time complexity of the algorithm.

3.2.1.1. Definitions and Notation

We let $l(P)$ denote the number of instructions in P. We let $r(P,t)$ denote the running time of P on test t, and $r(P,T)$ denote the *average* running time of all of the tests in T. Since each test executes some subset of the instructions of P, for test t, we let $\rho(P,t)$ denote the number of distinct instructions of P executed by t, and for test set T, we let $\rho(P,T)$ be the *average* number of distinct instructions executed in one run of a

test in T.

When we say that "a test reaches a mutant" we mean that the statement by which the mutant differs from the original program is executed at least once during execution of the program on the test.

For each mutant M, $SK(M,T)$ is the number of test cases in T that strongly kill M. We define $P_{strong}(M,T)$ to be the fraction $SK(M,T)/|T|$. $P_{strong}(M,T)$ is the frequency with which tests in T strongly kill M. We let $P_{strong}(P,T)$ denote the average value of $P_{strong}(M,T)$ over all mutants $M \in \mathbf{M}(P)$. Imagine a binary matrix $S(i,j)$ with a column for each test in T and a row for each mutant M, such that $S(i,j) = 1$ if and only if test t_j strongly kills mutant $M_i(P)$, and $S(i,j) = 0$ otherwise. Then

$$P_{strong}(P,T) = \frac{\sum_{i=1}^{|M(P)|} \sum_{j=1}^{|T|} S(i,j)}{|M(P)|\,|T|}$$

is another definition for $P_{strong}(P,T)$. This definition is equivalent to the previous one. For ease of notation we write P_{strong} instead of $P_{strong}(P,T)$ omitting the parameters.

For the purpose of our performance analysis, we use P_{strong} as an estimate of the probability that an arbitrary test in T kills an arbitrary mutant of P strongly. This is reasonable because we assume no particular strategy for selecting the next test to evaluate and because all mutants must be analyzed to decide the adequacy of the test set. In fact, regardless of the strategy used for selecting the next test to evaluate, our strong mutation algorithm offers some improvement over the naive method. Informally, this is because if the tests are ordered so that many mutants are killed "quickly" in the naive method,

then under this same ordering, the same mutants are killed quickly in our method, with less work. The point is that the speed-up of our algorithm is independent of the mutant evaluation strategy and that the reason we use P_{strong} as an estimate of the probability of killing a mutant is to facilitate the analysis of this speed-up.

We similarly define P_{weak} and P_{reach} to be the average "probabilities" that an arbitrary test weakly kills and reaches a mutant respectively. We can make arguments similar to those above regarding the appropriateness of these definitions.

We use $P_{weak \mid reach}$ and $P_{strong \mid weak}$ to denote the conditional probabilities that a mutant is weakly killed given that it is reached, and is strongly killed given that it is weakly killed, respectively.

Let $T_{strong}(P,T)$ or simply T_{strong} be an average over all mutants $M(P)$ of the number of tests in T that strongly kill a mutant. Similarly we define T_{weak} and T_{reach}. It is trivial to see from the matrix definition of probability above that $T_{goal} = |T| P_{goal}$, where $goal$ can be replaced by each of $reach$, $weak$, and $strong$. Directly from the definition of $P_{weak \mid reach}$ follows that

$$P_{weak \mid reach} = T_{weak} / T_{reach} = P_{weak} / P_{reach}, \tag{3.1}$$

analogously

$$P_{strong \mid weak} = P_{strong} / P_{weak}. \tag{3.2}$$

For a given test set T and program P, we define $\varepsilon_{strong}(P,T)$ to be the probability that an arbitrary mutant of P is *not* strongly killed by any tests in T. We similarly define $\varepsilon_{weak}(P,T)$ and $\varepsilon_{reach}(P,T)$ to be the probabilities that a mutant is not weakly killed or reached, respectively, by any tests in T. For ease of notation, we drop the arguments

"(P,T)", using instead ε_{strong}, ε_{weak}, and ε_{reach}. From the preceding definitions and assumptions it follows that

$$\varepsilon_{goal} = (1 - P_{goal})^{|T|} \tag{3.3}$$

Finally, let α be the average fraction of a trace to be executed once it is discovered that a mutant is weakly killed.

3.2.1.2. Initial Analysis

We establish a few facts that are used in the more detailed performance analyses that follow the descriptions of the algorithms.

Imagine a binary matrix $R(i,j)$ with a column for each test in T and a row for each instruction of P, such that $R(i,j) = 1$ if and only if test t_j reaches statement I_i of P and $R(i,j) = 0$ otherwise. Then $\rho(P,t_j)$, the number of distinct instructions in P executed by t_j, is the sum of the entries in the jth column of $R(i,j)$. The row sum of the ith row is the number of tests in T that reach I_i, which we denote $\rho(I_i,T)$. Since the sum of the row sums equals the sum of the column sums, we have

$$\sum_{j=1}^{|T|} \rho(P,t_j) = \sum_{i=1}^{l(P)} \rho(I_i,T) = R(P,T)$$

where $R(P,T)$ is the total number of 1s in the $R(i,j)$ matrix. By definition,

$$\rho(P,T) = \frac{1}{|T|} \sum_{j=1}^{|T|} \rho(P,t_j) = \frac{R(P,T)}{|T|} \tag{3.4}$$

Also by definition,

$$P_{reach} = \frac{1}{l(P)} \sum_{i=1}^{l(P)} \frac{\rho(I_i,T)}{|T|} = \frac{R(P,T)}{l(P)|T|} \tag{3.5}$$

From Equations 3.4 and 3.5 it follows that

$$P_{reach} = \frac{\rho(P,T)}{l(P)} \tag{3.6}$$

which means that the average probability that a statement is reached in P is the average number of statements executed in P by tests of T divided by the number of statements in P.

Let ζ_{reach}, ζ_{weak}, and ζ_{strong} denote the average numbers of tests needed to reach, weakly kill, and strongly kill a mutant respectively. Then, letting "goal" stand in for each of "reach", "weak", and "strong" respectively,

$$\zeta_{goal} = 1P_{goal} + 2P_{goal}(1 - P_{goal}) + 3P_{goal}(1 - P_{goal})^2 + \cdots$$

$$+ (|T|-1)P_{goal}(1 - P_{goal})^{|T|-2} + |T|(1 - P_{goal})^{|T|-1}$$

Note that the last term of this expression is not multiplied by the factor P_{goal} because the probability that exactly $|T|$ tests are needed is the probability that all preceding $(|T| - 1)$ tests did not satisfy the goal, regardless of the outcome of the $|T|th$ test. We prove in Appendix A that if $P_{goal} > 0$ then

$$\zeta_{goal} = \frac{1 - (1 - P_{goal})^{|T|}}{P_{goal}} \tag{3.7}$$

and combined with Equation (3.3) we obtain

$$\zeta_{goal} = \frac{1 - \varepsilon_{goal}}{P_{goal}} \tag{3.8}$$

It is obvious that $\varepsilon_{reach} \leq \varepsilon_{weak} \leq \varepsilon_{strong}$. When mutation analysis is being used as an adequacy criterion it is fair to assume that the test sets that would be evaluated are

reasonably close to adequate, and so ε_{strong} is close to 0; we can safely say that $\varepsilon_{strong} < 0.2$. Therefore, the factor $(1 - \varepsilon_{goal})$ is close to 1.0, i.e.,$(1 - \varepsilon_{goal}) > 0.8$. For this reason and because $(1-\varepsilon_{goal})$ is a factor in the run time complexities of all algorithms that we compare, we assume that

$$\zeta_{goal} = \frac{1}{P_{goal}} \tag{3.9}$$

In other words, the average number of tests needed to check whether an arbitrary mutant is reached, weakly killed, or strongly killed using the given test set is roughly the inverse of the probability that a test reaches, weakly kills, or strongly kills an arbitrary mutant. We point out that since $(1 - \varepsilon_{goal}) < 1$, this estimate of ζ_{goal} is a conservative one - it overestimates the number of tests needed, and consequently, the running time required by the algorithms.

3.2.2. General Description

We describe three functions, `Reachable`, `WeaklyKilled`, and `StronglyKilled` , and a driver routine that calls one or more of them depending upon whether weak or strong mutation is the ultimate goal. These functions work with each other, using the MDS to decide whether a particular level of coverage is achieved. Their interaction is fairly complex because they have been designed to eliminate all redundancy in the execution of mutants.

The general idea underlying the algorithms is to avoid doing any work that is not absolutely necessary at the time. Thus, `StronglyKilled` attempts to check only those tests that weakly kill, `WeaklyKilled` checks only those that reach the mutated

statement, and `Reachable` checks only reachability of a test when it is called upon to do so, and even then it is designed to retain information about past requests so as not to duplicate previous work.

The algorithms assume that the tests in T are organized into an ordered list, with t_i preceding $t_{(i+1)}$. They further assume that the mutants are organized into $|I|$ lists, one for each statement, each of which contains only mutants of that statement. We write M_j^k to denote the kth mutant in the list of mutants of the jth statement of P. For each of these lists we maintain two pointers: one that points to the last mutant checked so far in the list, and another that points to the last mutant of the list. Each time that a mutant is checked by the driver, if it is discovered that it has been killed, either strongly or weakly depending upon the goal, it is deleted from the list. Because we maintain a pointer to the mutant just checked, the deletion takes constant time. In any case, after checking a mutant, the pointer is advanced to the next mutant in the list, unless it is already at the end of the list.

Each time a new mutant is created, it is appended to the end of the list of mutants of its mutated statement. We do not assume that these lists are constructed prior to the start of analysis; we allow for the possibility that mutants could be created and appended to the lists while mutation analysis is taking place. Our algorithms are designed to allow such parallel activity to take place.

```
function Reachable(j:StatementIndex; p:ReachPtr):ReachPtr;
begin
        Found := False;
        if p = NIL then
          i := 1
        else
          if p↑.next ≠ NIL then
            begin Reachable := p↑.next; Found := True end
          else
            i := p↑.t + 1;
        while not Found and (i ≤ |T|) do
              if MDS[i].I[j] = NIL then
                i := i + 1
              else begin
                Found := True;
                q := newreachNode;
                q↑.t := i;
                q↑.next := NIL
                if p ≠ NIL then
                  p↑.next := q;
                Reachable := q
                end;
        if not Found then
              Reachable := NIL
end {Reachable}
```

Figure 3.2. Reachability Function.

3.2.3. The Reachability Function

The function `Reachable` is designed so that, given the index j of a statement I_j, it makes the amortized cost of deciding the reachability of statement I_j over all mutants M_j^k as small as possible. `Reachable` returns a pointer to the next test that reaches statement I_j, if one exists, and returns NIL otherwise. Since many mutants are obtained by mutating the same statement, it is wasteful to have to check each time we start a new mutant whether a given test reaches the mutated statement of that mutant, since we might have already checked. This suggests that for each mutated statement we make one preprocessing pass over all tests, making a list of those tests that reach this statement. But this too is wasteful, since we might never need to look at, say, the $100th$ test if the first one reaches and kills all mutants. Therefore, `Reachable` constructs this list on a "demand" basis, extending the list only as those reaching tests currently in the list turn out to be insufficient to kill the current mutant.

Each time the driver starts to analyze a new mutant M_j^k, it initializes a pointer `p` to the start of the reachable list for I_j. The pointer `p` points to the next place in the reachable list from which to start searching for a test that reaches I_j. During the analysis, `Reachable(j,p)` is called whenever a test that reaches I_j is needed. Not only does `Reachable` return a pointer to the next reachable test that this mutant has not yet seen, if one exists, but it also appends to the reachable list for statement I_j, a node that points to this test in case it is not yet on the list. When processing all mutants of a given statement is finished the space allocated to the reachable list is released, to be reused as needed by the reachable list for the next statement. In the algorithm in Figure 3.2, if `p` is a pointer to a node in the reachable list, then `p^.t` denotes the *index* of the test

represented by that node.

Analysis

Because we process all mutants of a given statement before those of another statement, we need to maintain only one reachable list at a time, so that the space complexity is at most $|T|$. It follows directly from the definition of ζ_{reach} and Equation (3.9) in Section 3.2.1 above that the amortized time complexity of deciding whether there is a test case in T that reaches the mutated statement of a given mutant is $1 / P_{reach}$, which, by Equation (3.6) is $l(P) / \rho(P,T)$. The time complexity of reachability analysis is roughly the size of the program divided by the average number of instructions executed by an arbitrary test.

3.2.4. The WeaklyKilled Function

We define a function called `WeaklyKilled`, which receives a mutant M_j^k and a pointer `p` to a test t_i that is known to reach the mutated statement, I_j. It checks to see whether any execution of I_j on this test causes M_j^k's state and P's state to differ. If so, it returns "True" as the function result, and transmits in a parameter a pointer to the first state in which they differ. If t_i does not weakly kill M_j^k then the `Reachable(j,p)` function is called to find the next test that reaches I_j, and `WeaklyKilled` once again checks whether this weakly kills the mutant. It repeatedly does this until either it finds a test that weakly kills M_j^k or the `Reachable` function returns a NIL pointer, indicating that the test set has been exhausted. If no test is found that weakly kills M_j^k then "False" is returned. The algorithm is shown in Figure 3.3.

```
function WeaklyKilled(M_j^k:mutant; var p:ReachPtr; var l:ListPtr):Boolean;
begin
      WeaklyKilled := False;
      while (not WeaklyKilled) and (p ≠ NIL) do begin
            nextState := MDS[p↑.t].I[j];
            while (nextState ≠ NIL) and (not WeaklyKilled) do begin
                  let σ_i be the trace element to which nextState points
                      and let σ_{i+1} be its successor;
                  execute the mutated I_j in the state σ_i;
                  if the resulting state is different than σ_{i+1} then
                        WeaklyKilled := True;
                        l := nextState {M_j^k is weakly killed in this state}
                  else
                        nextState := nextState↑.next
            end; {while}
            if not WeaklyKilled then
              p := Reachable(j,p)
            end {while}
end {WeaklyKilled}
```

Figure 3.3. WeaklyKilled Function.

Analysis

For a single test the worst case time complexity of this algorithm is the number of times that the mutated statement is executed on t_i, which is the length of the linked list attached to MDS[i][j]. This length in the worst case can be as long as the trace itself, $r(P,t)$. On average, this length is the average number of times that a single instruction is executed by an arbitrary test, which is the total number of instructions executed by an arbitrary test in T divided by the average number of distinct instructions in P executed by the tests in T:

$$\frac{r(P,T)}{\rho(P,T)}$$

From Equation (3.6) it follows that this length is on average

$$\frac{r(P,T)}{l(P)P_{reach}} \tag{3.10}$$

The number of tests that might need to be tried before finding one that weakly kills the mutant is, on the average, P_{reach} / P_{weak} . This is obtained by an argument similar to the one used to derive the estimate of ζ_{goal} in Equation (3.9); we can show that the expected number of tests on which `WeaklyKilled` is run is $1/P_{weak\,|reach}$, or using Equation (3.1) P_{reach} / P_{weak}.

The average time complexity to check whether a single mutant is killed is the product of list length and number of tests, and is thus, after simplification,

$$\frac{r(P,T)}{l(P)P_{weak}} \tag{3.11}$$

We believe, but have not proved, that this result is optimal for serial weak mutation, in the sense that it cannot be improved by more than a constant factor without a particular strategy for choosing tests that are more likely to kill weakly the mutant.

The speed-up of this method over the naive one for weak mutation is roughly $l(P)$, the size of the program, a substantial improvement, and as we detail below, the speed-up over the more efficient weak mutation systems described in Section 2.4 can also be quite significant. This serial algorithm is also faster than some weak mutation algorithms running on SIMD machines [37].

3.2.5. The StronglyKilled Function

The function `StronglyKilled` receives a mutant M_j^k, a pointer `p` to a test t_i that is known to kill weakly the mutated statement I_j, and a pointer to the state in the trace just before the mutant's and program's states differ. It checks whether this test strongly kills M_j^k by running the mutant to termination from that state and comparing outputs. If they differ it returns "True", otherwise it repeatedly calls the `WeaklyKilled` function to obtain the next test that weakly kills M_j^k until either it has exhausted all tests or it finds one that weakly kills it. The algorithm is in Figure 3.4.

We defer analysis of the complexity of this algorithm until explaining the driver routine that determines whether a test set satisfies weak or strong mutation. That driver is in Figure 3.5. In the line marked with a (*), the "X" should be replaced by "Strong" or "Weak", depending upon whether the goal is strong or weak mutation.

Initially, a newly created mutant is considered *new* by the driver algorithm. This means that it has been appended to the list of mutants of the corresponding statement but

has not yet been checked. After it has been processed by the driver, it is no longer considered new; this prevents a mutant from being checked twice in the case that the driver is being used during incremental mutant generation, which we describe below. In essence, all mutants preceding the pointer that points to the last mutant checked are considered old and all those following it are considered new.

To do strong mutation analysis on the entire set of mutants, we process them in groups ordered by mutated statement, and for each group, we check each mutant in turn, calling `StronglyKilled`. Prior to the first call, we find the first test that reaches the statement. After the driver has been run on all mutants, we check whether all remaining mutants in the lists are equivalent to the original program. If so, then the test data was strong mutation adequate. Weak mutation analysis is done analogously.

Analysis

The average time complexity of strong mutation using this method is the product of two factors. One is the expected number of tests on which the `StronglyKilled` function has to run a mutant, and the other is the time cost of finding such tests and running the mutants on them until they terminate. Again using an argument similar to the one used to derive the estimates of $\zeta_{reach}, \zeta_{weak}$, and ζ_{strong} in Equation (3.9) we can show that the expected number of tests on which the `StronglyKilled` function has to run a mutant, given that the mutant is weakly killed by each test, is $1 \,/\, P_{strong \mid weak}$, or using Equation (3.2) simply $P_{weak} \,/\, P_{strong}$. The second factor is the sum of the average time complexity of weak mutation, Equation (11), plus the term $\alpha r(P,T)$, The complexity is thus

```
function StronglyKilled(M_j^k:mutant; var p:ReachPtr; var l:ListPtr):Boolean;
begin
        StronglyKilled := False;
        while (WeaklyKilled(M_j^k,p,l) and not StronglyKilled do begin
                run M_j^k starting from the state in MDS[p ↑ .t].S where
                    M_j^k was discovered to be weakly killed;
                if output of M_j^k differs from the output of P then
                  StronglyKilled := True
                else
                  p := Reachable(j,p)
                end {while}
end {StronglyKilled}
```

Figure 3.4. StronglyKilled Function.

```
procedure MutationDriver;
begin
        for each statement I_j do begin
            ReachList := Reachable(j, NIL);
            if ReachList ≠ NIL then
              for each new mutant M_j^k of I_j do begin
                  p := ReachList;
                  if XlyKilled(M_j^k,p,l) then (*)
                    remove M_j^k from the jth list
                  end
end {MutationDriver}
```

Figure 3.5. Mutation Analysis Driver.

$$\frac{P_{weak}}{P_{strong}}\left(\frac{r(P,T)}{l(P)P_{weak}} + \alpha r(P,T)\right) \tag{3.12}$$

It should be pointed out that Equation (12) is an upper bound on the average time complexity to check whether a mutant is strongly killed. This is because $r(P,T) \,/\, (l(P)P_{weak})$ is actually an upper bound on the cost of repeatedly calling the `WeaklyKilled` function; it is really the average cost of the *first* call to `WeaklyKilled`. The amortized cost of repeated calls to `WeaklyKilled` is strictly less than $r(P,T) \,/\, (l(P)P_{weak})$ because each time that it is called, intuitively, the maximum number of tests that it checks in subsequent calls is diminished, implying that the average cost of subsequent calls diminishes or decreases.

Simplifying (12) the run time of complexity of strong mutation becomes

$$\frac{r(P,T)}{P_{strong}}\left(\frac{1}{l(P)} + \alpha P_{weak}\right)$$

which, if $P_{weak} \ll 1/l(P)$, implies that the complexity is roughly

$$\frac{r(P,T)}{P_{strong}\, l(P)}$$

This would mean in turn that the speed-up in comparison with the naive algorithm is proportional to the length of the program. If P_{weak} and P_{strong} are close in value, as would be implied by the weak mutation hypothesis, then we could perform strong mutation for almost the cost of weak mutation.

If $P_{weak} \gg 1 \,/\, l(P)$ then the complexity is roughly

$$\frac{r(P,T)}{P_{strong}} \cdot \alpha P_{weak}$$

This is just the average time to run repeatedly a mutant to termination from the state at which it first differs until it is strongly killed. In this case the speed-up is the factor $1 / (\alpha \cdot P_{weak})$. If we assume that on average the first point at which the mutant's state differs is the halfway point of the trace, then $\alpha = 1/2$ and the speed-up is at least $2 / P_{weak}$.

It is very difficult to predict the value of α. It is obvious that some fraction of mutants may do something "really stupid", such as dividing by 0, and therefore, terminate abnormally just after the point in the trace where they are weakly killed. If the majority of the mutants behave this way, then α is close to 0. On the contrary some fraction of mutants may enter infinite loops and have to be killed by the timer. These mutants contribute to increasing the value of α and if the majority of mutants behave this way α may be close to 1. It should be noted that the value of α depends on both the program and the test set. Therefore, the use of $\alpha(P,T)$ is more appropriate, but for simplicity we omit the parameters. Also, the preceding analysis is very general, in that it captures the potentially different behaviors of a program's mutants in a general way through use of the parameter α.

3.3. Further Performance Results

The preceding performance results are summarized in Table 1. There we compare the average time complexities of applying the naive methods of mutation analysis and our algorithms to a single mutant. In the table, naive reachability refers to the method in which the mutated statement is replaced by a trap statement and naive strong mutation is performed [18].

We used the assumption from 3.2.1 that the majority of mutants are killed by some test in the test set. If we remove this assumption, then the time complexities shown for both the naive method and our method of strong mutation need to be multiplied by the factor $(1 - \varepsilon_{strong})$. Recall from Section 3.2.1 that $\varepsilon_{strong} = (1 - P_{strong})^{|T|}$ is roughly the fraction of mutants that are *not* strongly killed by any tests in T. Similarly, the time complexities shown for both the naive method and our method of weak mutation and reachability analysis need to be multiplied by the factors $(1 - \varepsilon_{weak})$ and $(1 - \varepsilon_{reach})$ respectively. Because the time complexities of the methods being compared are multiplied by the same factor, the speed-ups shown do not change. In this sense our analysis is very general.

3.3.1. Comparison of Weak Mutation Algorithms

So far we have compared our weak mutation algorithm to what we call the naive one, exemplified by systems such as *Leonardo*. It is natural for the reader to wonder how our algorithm compares to the more efficient method alluded to by Howden [35]. Since there are no details available that describe this method, we presuppose that it might work as follows: The original program is instrumented so that during execution of a single test, each time it reaches a mutated statement it stops and compares its state to the states of each possible mutant of that statement, and if a mutant is killed, it records that it is killed. As we noted earlier, GCT is a working system that uses this basic idea.

The average time complexity of this method of weak mutation is proportional to the product of the average number of mutants per statement of the program, the number of tests on which the program is executed, and the average running time of each test. The

average number of mutants per program statement is $\theta(P)=\frac{|\boldsymbol{M}(P)|}{l(P)}$. The product $r(P,T)\cdot\theta(P)$ is the average cost of checking a single test executing each mutant of each statement of the program when execution reaches that statement, since each test reaches $r(P,T)$ statements on average during its execution and must check $\theta(P)$ mutants at each one. Since the program is executed on all $|T|$ tests, the average time complexity is

$$r(P,T)\cdot\theta(P)\cdot|T| \tag{3.13}$$

and the per-mutant complexity is therefore

$$\frac{r(P,T)\cdot\theta(P)\cdot|T|}{|\boldsymbol{M}(P)|}=\frac{r(P,T)|T|}{l(P)} \tag{3.14}$$

This analysis assumes that systems such as GCT execute the instrumented program on the entire test set.

The average time complexity of our weak mutation algorithm is less than or equal to

$$\frac{r(P,T)}{l(P)P_{weak}}\cdot(1-\varepsilon_{weak}) \tag{3.15}$$

and strictly greater than

$$\frac{r(P,T)}{l(P)P_{weak}}\cdot(1-\varepsilon_{weak})(1-P_{weak}) \tag{3.16}$$

The first thing to observe about Formulas (3.15) and (3.16) is that the factor $(1-\varepsilon_{weak})$ is present in both of them but not in Table 3.1. Recall that we intentionally omitted this factor in the expressions in the table because the factor was present in the complexities of all methods being compared there, and its omission simplified the formulas.

Formula (3.15) is an upper bound; it represents the case in which, for each mutant, the entire linked list for the mutated statement must be traversed to check whether the mutant is weakly killed by the particular test case that kills it. Since for each of the tests that do not weakly kill the mutant, the entire list must also be traversed, this implies that for each mutant, the amount of work performed to check whether it is killed is the same for each test checked.

What is more likely is that the entire linked list does not need to be traversed when checking the test case that actually weakly kills the mutant, but only some fraction of it. Formula (3.16) represents the case in which, for each mutant, the test case that weakly kills the mutant is discovered without having to traverse the list beyond the first node. Thus, although all previous non-killing tests require that the entire linked list be traversed, the killing one requires the least work possible. Formula (3.16) is therefore a lower bound on the amount of work involved in this case; it is derived in Appendix B.

A comparison of Formula (3.14) with Formulas (3.15) and (3.16) shows that the speed-up of our weak mutation algorithm over the other one lies between

$$\frac{|T|P_{weak}}{1-\varepsilon_{weak}} \tag{3.17}$$

and

$$\frac{|T|P_{weak}}{(1-\varepsilon_{weak})(1-P_{weak})} \tag{3.18}$$

If P_{weak} is greater than 0.5, then the speed-up is roughly the size of the test set multiplied by a factor close to 1.0. For example, if $P_{weak} = 0.5$, then, using the conservative

Mutation Problem	Naive Method	Our Method	Speed-up
reachability	$r(P,T)\cdot(\frac{1}{P_{reach}}-\alpha)$	$\frac{1}{P_{reach}}$	$r(P,T)(1-\alpha P_{reach})$
weak mutation	$r(P,T)\cdot(\frac{1}{P_{weak}}-\alpha)$	$\frac{r(P,T)}{l(P)P_{weak}}$	$l(P)(1-\alpha P_{weak})$
strong mutation	$\frac{r(P,T)}{P_{strong}}$	$\frac{r(P,T)}{P_{strong}}(\frac{1}{l(P)}+\alpha P_{weak})$	$\sim \min(l(P),\frac{1}{\alpha P_{weak}})$

Table 3.1. Comparison of Time Complexities.

ε_{weak}	Speed-up
0.50000	1.4
0.10000	2.6
0.05000	3.2
0.01000	4.7
0.00500	5.3
0.00100	6.9
0.00005	9.9
0.00001	11.5

Table 3.2. Sample Values of ε_{weak} versus Speed-up.

estimate that $1 - \varepsilon_{weak} = 1.0$ in Formulas (3.17) and (3.18), we obtain that the speed-up is at least $|T|/2$ and at most $|T|$. Thus, even for very small test sets, our method offers a great improvement over the other.

In the case that P_{weak} is less than 0.5, we use (3.17), the more pessimistic estimate of the speed-up, to obtain a more accurate estimate. Consider the definition of ε_{weak}:

$$\varepsilon_{weak} = (1 - P_{weak})^{|T|} \tag{3.19}$$

$$= \left((1 - P_{weak})^{\frac{1}{P_{weak}}} \right)^{P_{weak}|T|}$$

$$= e^{P_{weak}|T| \ln(1-P_{weak})^{\frac{1}{P_{weak}}}}$$

From Equation(3.19) it follows that

$$P_{weak}|T| = \frac{\ln \varepsilon_{weak}}{\ln(1-P_{weak})^{\frac{1}{P_{weak}}}} \tag{3.20}$$

Since

$$\lim_{P_{weak} \to 0} (1 - P_{weak})^{\frac{1}{P_{weak}}} = \frac{1}{e}$$

we have that

$$\lim_{P_{weak} \to 0} \ln(1 - P_{weak})^{\frac{1}{P_{weak}}} = -1$$

To give the reader an idea about the rate of this convergence, for $P_{weak} = 0.5$, 0.25 and 0.125,

$$\ln(1-P_{weak})^{\frac{1}{P_{weak}}}$$

is about -1.39, -1.15, and -1.07 respectively. Thus,

$$P_{weak} \cdot |T| \approx -\ln \varepsilon_{weak} = \ln \frac{1}{\varepsilon_{weak}}$$

is a good approximation for small values of P_{weak}. This results in the following expression for the speed-up:

$$\frac{-\ln \varepsilon_{weak}}{1-\varepsilon_{weak}} \tag{3.21}$$

As ε_{weak} approaches 1, (3.21) approaches 1. This implies that our weak mutation algorithm offers little speed-up over existing efficient techniques when the data sets are of extremely poor quality, meaning that few, if any, mutants are weakly killed by the data. For very small values of ε_{weak}, (3.21) is approximately $\ln(1/\varepsilon_{weak})$.

Table 3.2 shows values of the speed-up for various values of $\varepsilon_{weak} \leq 0.5$. From the table we can conclude that when we run our algorithm on test sets that weakly kill almost all mutants, there is a big improvement in performance.

3.4. Modifications to the Data Structure

As we noted earlier, the data structure used by these algorithms can take up an extremely large amount of storage. However, a careful examination of the algorithms shows that for most of their execution time, only small portions of the data structure are actually needed. In this section we take advantage of this observation and propose two different and complementary methods of reducing the space requirements of our

algorithms.

3.4.1. Modification 1: Reducing the Number of Saved States

A sensible way to reduce the space requirements is to pare down the data structure so that it contains only information that is actually used a large fraction of the running time, and to construct the remaining pieces of it on a demand basis. As we justify shortly, this "lazy" approach to the maintenance of the data structure can save considerable space, at the cost of sometimes having to execute the same code twice.

The original MDS for test case t_i has two components: the trace MDS[i].S of the program on input t_i, and an array MDS[i].I indexed by the statements of the program. Recall that MDS[i].I[j] points to a linked list of nodes in which the kth node in the list points to the state in the trace immediately prior to the kth execution of statement I_j. (See Figure 3.1). The modified MDS, or MMDS for short, takes advantage of the decomposition of the program into basic blocks. In the MMDS, rather than storing each program state of the trace, we store only those states that occur prior to the executions of the first statements of the basic blocks of the program. We make an analogous change to the instruction array; the components of the modified instruction array, MMDS[i].I, correspond to the first statements of the basic blocks of the program. The components corresponding to all other statements are deleted. The linked list attached to MMDS[i].I[j] is a sequence of nodes such that the kth node in the list points to the state in the trace in which the first statement of the jth basic block is about to be executed for the kth time.

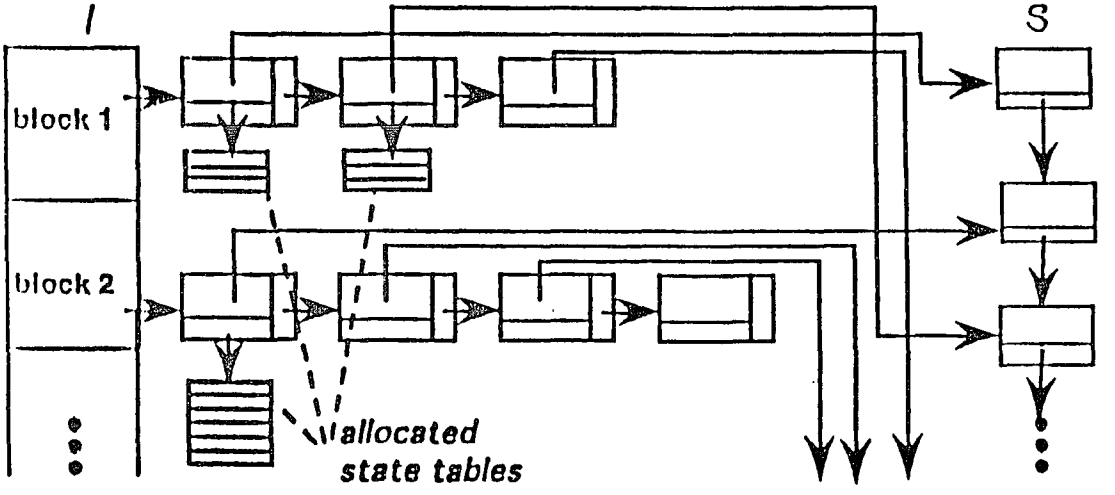

Figure 3.6. The Modified MDS.

In addition, each of the nodes in the list MMDS[i].I[j] contains a pointer, initially *NIL*, that can point to a dynamically allocated table of states, one state for each statement in the jth basic block. If the jth basic block contains m_j statements, then there can be as many as m_j states in this table, although there may be fewer at different times. The rth state in this table, $1 \leq r \leq m_j$, is the program state *after* execution of the rth statement of the jth block. The table attached to the kth node in the linked list contains the states after execution of the statements of the block the kth time the block has been executed. Thus, the rth state in the table attached to the kth node in the list MMDS[i].I[j] is the state of the program after executing the rth statement in the jth basic block for the kth time. The table does not need to exist at all times. It is allocated only when it is needed, and deallocated once we know for sure that it is no longer be needed again. Below we describe exactly when it is allocated and deallocated. Figure 3.6 depicts how a portion of the MMDS might look during mutation analysis.

The MMDS is created in the same way as the original MDS, by executing an instrumented version of the original program. But instead of saving the program state prior to each statement, the instrumentation is designed to save state information only upon entering a basic block. At the time the MMDS is created, none of the state tables is created. When a state table does need to be created, it is a trivial matter to do so, since it is sufficient to execute the basic block of the original program, starting in the state saved in the trace immediately prior to entering it, saving the state after execution of each of its statements.

The only change to the mutation analysis algorithms mandated by the MMDS is in the `WeaklyKilled` function. Reachability analysis does not change because a

statement is reached if and only if the basic block in which it occurs is reached, so to decide whether a particular statement is reached by a test, it is sufficient to check whether the list corresponding to the basic block in which it occurs is empty, and the structure of the lists in the MMDS is the same as in the original MDS. Strong mutation analysis does not change because, other than making repeated calls to the `WeaklyKilled` function, the `StronglyKilled` function just runs the actual mutants from the points at which they are weakly killed until they terminate, and then compares outputs. This activity does not require the MDS.

Let us denote the block in which the jth statement of the program occurs as $block(j)$. Suppose that the jth statement of the program is the rth statement in $block(j)$. To check whether mutant M_j^k is weakly killed by test t_i, assuming that t_i reaches the block, it is necessary to compare the states of the mutant and the original program after each execution of this statement or until we discover that these states are different. If $r > 1$, then state tables have to be constructed, if they do not yet exist. Starting with the first node in the linked list for $block(j)$ in MMDS[i], a state table that can store r program states is allocated and filled with the states immediately following executions of the first r statements of $block(j)$. This table is constructed by executing an instrumented version of the original program in the state just prior to the current execution of $block(j)$, from the first statement of the block to the rth statement. The mutated statement is then executed in the state prior to the mutated statement (the $(r-1)th$ state), and the resulting mutant state is compared to that of the original program, which is the rth state in the table. If the two states differ, the mutant is weakly killed by this test and there is no need to traverse the list further. If not, then a state table has to be constructed

for the next node in the list, and so the preceding steps are repeated. This process of attaching state tables to nodes in the list continues until either M_j^k is weakly killed by test t_i or we reach the end of the list. It follows that for each MMDS checked except the one representing the test that kills the mutant, every node in the linked list attached to $block(j)$ has a state table allocated and in the one that does weakly kill the mutant, if any, some nodes in the linked list do not have state tables allocated.

The collection of MMDSs is probably utilized most efficiently if the mutants are evaluated in a particular order. To illustrate, suppose that a given block has 10 statements. The mutants should be arranged so that we first check whether all mutants of the $10th$ statement are killed, then whether all mutants of the $9th$ statement are killed, then those of the $8th$, and so on, until we check the mutants of the first statement. To see why this order is the most efficient, consider the way the decision is made as to whether the mutant is weakly killed. In the preceding paragraph, we pointed out that we need to execute the mutant and the program in the state just prior to execution of the mutated statement, and that we therefore need to access the state table to obtain that state. By first checking all mutants of the $10th$ statement of the block, we force *complete* state tables to be created for each node in the linked lists associated to that block in each MMDS examined, up to the node in which we discover that the mutant has been weakly killed, if at all. In other words, by starting with the $10th$ statement, we force each allocated state table to contain all states up to and including the $10th$ state of the table. If the mutant is not weakly killed by a particular test case, then a complete state table is allocated for each node in the linked list of the associated MMDS. If it is weakly killed by the test case in, say, the kth execution of the mutated statement, then complete state tables have

been allocated for the first k nodes in the list of that MMDS.

When we have checked all mutants of the $10th$ statement, we proceed to those of the $9th$ statement. Suppose that complete state tables have been constructed for the first s nodes in the linked list of a particular MMDS, say MMDS[i]. The first time that we check whether a mutant of the $9th$ statement is weakly killed by test case t_i there are three possibilities:

1. It is weakly killed in the kth execution of this statement, where $k \leq s$. In this case, no additional state tables need to be constructed, because $k \leq s$ and the first s state tables exist, each of which contains the state immediately after the mutated statement.
2. It is weakly killed in the kth execution of this statement, where $k > s$. In this case, $k-s$ state tables need to be constructed, but each has one fewer state than the first s tables.
3. It is not weakly killed. In this case, state tables each with one fewer state have to be constructed for all nodes in the linked list after the sth node.

Thus, the most new work that needs to be performed in MMDS[i] is that we have to create some new state tables containing 9 states, but we have to create these anyway to check this mutant. It is worth observing at this point that if any of the mutants of the $10th$ statement were not weakly killed by any test case, then complete state tables have been allocated to all nodes in all linked lists of all MMDSs. In this situation, which might be very common, after all mutants of the $10th$ statement have been checked, no further allocation would have to be done.

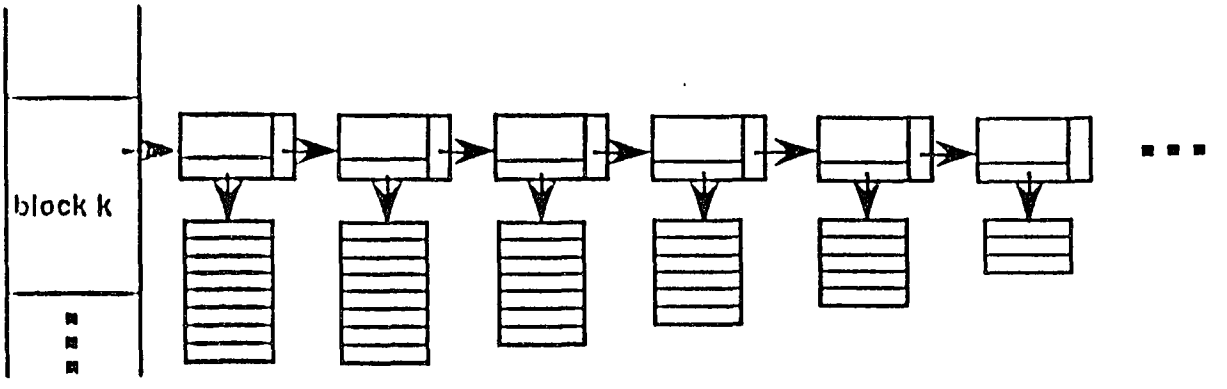

Figure 3.7. Link List and State Tables.

Suppose instead that we reversed the order in which we checked the mutants, starting with those of the first statement, followed by those of the second, the third, and so on. Each time we check mutants of the next statement, we need to obtain a state not yet constructed, regardless of how much of the linked list was already traversed during the analysis of mutants of previous statements. To construct this state requires stopping the algorithm and restarting the program in the previous state. We assume that the overhead involved in restarting the program each time we need to compute a new state is greater than the actual execution time of an average block, and so this is a very poor order in which to check mutants.

Figure 3.7 illustrates how the linked list attached to a given block in the MMDS looks after processing many mutants of statements of the block. Each time we start checking mutants of a lower numbered statement, we have to allocate more state tables, but the new state tables, which occur after the old ones in the list, are smaller by at least one state. The effect is that the sizes of the state tables allocated to the nodes in a given linked list diminish as the list is traversed from start to end, as the figure shows.

When all of the mutants of statements in a given block have been checked by the `WeaklyKilled` function, the state tables attached to the nodes in the block's list are deallocated. This implies that at any time the only state tables that are allocated are those of the nodes of a single basic block of the program.

Analysis

The key question is how much space can be saved, on average, as a result of this modification of the MDS. To answer this question we make one simplifying assumption,

that there is no correlation between the number of statements in a basic block and the frequency with which that block is entered during execution. This being the case, the analysis is relatively simple.

Let $\beta(P)$ denote the number of basic blocks in program P, and define $b(P) = l(P)/\beta(P)$. Since $l(P)$ is the number of statements in P, $b(P)$ is the average number of statements per block. For convenience, we take as the unit of memory the size of a program state of P. Assuming that the trace portion of the original MDS dominates the total memory used by the MDS, which is the case of interest, the amount of memory used by the original MDS is $r(P,t)$, the length of the trace.

We separate the memory consumption of the MMDS into two portions: the statically allocated memory and the dynamically allocated memory. The size of the statically allocated memory, SM, is the number of states permanently allocated to the trace, which is the length of the trace divided by the number of statements per block:

$$SM = \frac{r(P,t)}{b(P)} = \frac{r(P,t)\beta(P)}{l(P)}$$

The size of the dynamically allocated memory, DM, is a function of the average length of the linked lists attached to the instruction array. The average length of the linked lists attached to the MMDS for test case t is

$$LL = \frac{r(P,t)}{\rho(P,t)}$$

Since in the worst case, each node in the linked list may have a state table attached and this state table may have as many as $b(P)$ states, the average amount of memory used by the dynamic portion of the MMDS is at most

$$DM = b(P) \cdot LL = \frac{b(P)r(P,t)}{\rho(P,t)}$$

Thus, the total average memory consumption of the MMDS is at most

$$MM = SM + DM = r(P,t)(\frac{1}{b(P)} + \frac{b(P)}{\rho(P,t)})$$

and so the space savings is

$$s = \frac{r(P,t)}{MM} = \frac{1}{\frac{1}{b(P)} + \frac{b(P)}{\rho(P,t)}} = \frac{l(P)}{\beta(P) + \frac{b(P)l(P)}{\rho(P,t)}} \qquad (3.22)$$

Since $\rho(P,T)$ is the average number of distinct statements executed by the tests in T, we replace $\rho(P,t)$ by $\rho(P,T)$ in (3.22) to obtain

$$s = \frac{l(P)}{\beta(P) + \frac{b(P)l(P)}{\rho(P,T)}}$$

and, using Equation(3.6),

$$s = \frac{l(P)}{\beta(P) + \frac{b(P)}{P_{reach}}} \qquad (3.23)$$

If the number of blocks in P is much smaller than $b(P)/P_{reach}$ then Equation(3.23) becomes

$$s \approx \frac{l(P)}{b(P)} = \beta(P)P_{reach}$$

This is very unlikely. For most test sets being evaluated for mutation adequacy, P_{reach} should not be a very small fraction -- it is safe to assume that at least one third of the statements in the program would be reached by the test data, and so $1/P_{reach}$ is at most

three. Even if only one fifth of the statements were reached, $1/P_{reach}$ is only five. Thus, it is more likely that the number of blocks in the program, $\beta(P)$, is much larger than $b(P)/P_{reach}$. In this case,

$$s \approx \frac{l(P)}{\beta(P)} = b(P)$$

In other words, the size of the MMDS is a fraction of the size of the original MDS, and this fraction is proportional to the average number of statements per block of the program, which is the "intuitive" expected savings. This is a substantial reduction in the space requirements of our algorithm.

3.4.2. Modification 2: Reducing the Size of Saved States

A different and complementary way to reduce the space requirements of the MDS is to reduce the amount of information contained in the states. A program state is ordinarily treated as a vector containing the values of all program variables. But most program statements change the values of only a few variables. Therefore, instead of storing the values of all variables, the state could contain the values of only those variables that have changed since some fixed checkpoint in the program, together with a link to the state just prior to this checkpoint. This *checkpoint state* would be a complete state, containing the values of all variables, but all states following this one, up to but not including the next checkpoint state, would be *partial states*, with links back to the checkpoint. The partial states would contain a list of the variables that had changed since the last checkpoint state, together with their new values. The link in a partial state would be followed just in case we needed the value of a variable that was *not* in the list of changed variables.

To perform mutation analysis using this modified state representation is straightforward. To execute the mutated statement, we first determine which variables are needed to execute the statement. If all such variables' values are contained in the immediately preceding state, then we execute the mutant using this state. If not, we follow the state's link back to the previous checkpoint state to obtain the values of the variables not in the partial state and execute the mutant using the values obtained. To check whether the mutant is weakly killed, we first determine which variables were changed in the original program as a result of executing the statement that is mutated in the mutant, and compare the values of these variables with those obtained by executing the mutated statement.

The increase in execution time caused by having to follow links back to checkpoint states is relatively insignificant. The only time that links are followed is after we have found a test that reaches the mutated statement and need to check whether the mutant is weakly killed by this test. Links are not followed after we have discovered that the mutant is weakly killed and want to know whether it is strongly killed, since from that point forward the mutant is executed normally and its output compared to that of the original program. Under the assumption that the number of variables that need to be checked is usually small, the increase in execution time is a small constant multiple of the total execution time. (It is proportional to the average number of variables checked during state comparisons.)

Of much more importance are the questions of how much space can be saved using this approach, and how often checkpoint states should be created to obtain the optimal space reduction. We answer these questions here. Let S_0 be the size of an arbitrary checkpoint state, and let $S_1, S_2, \ldots, S_n$ be the sizes of the partial states starting from the

state immediately after the checkpoint and ending with the one just prior to the next checkpoint state. Then the problem of maximizing space reduction can be formulated as the problem of finding a value of n that makes the average space used by this portion of the trace minimal:

$$Find \quad n \geq 0 \quad such \quad that \quad \bar{S} = \frac{\sum_{k=0}^{n} S_k}{n+1} \quad is \quad minimum.$$

To solve this problem we seek a value of n such that $\bar{S}$ does not change regardless of whether the next state is partial or checkpoint, i.e., we seek n such that

$$\frac{\sum_{k=0}^{n} S_k}{n+1} = \frac{\sum_{k=0}^{n-1} S_k}{n} \tag{3.24}$$

Rewriting Equation(3.24), we get

$$n \sum_{k=0}^{n} S_k = (n+1) \sum_{k=0}^{n-1} S_k$$

which implies that

$$nS_n = (n+1) \sum_{k=0}^{n-1} S_k - n \sum_{k=0}^{n-1} S_k = \sum_{k=0}^{n-1} S_k \tag{3.25}$$

This is interpreted as follows. If $nS_n < \sum_{k=0}^{n-1} S_k$ then the nth state should not be a checkpoint state, since the left side of Equation(3.24) is smaller than the right side. On the other hand, if $nS_n > \sum_{k=0}^{n-1} S_k$ then it is time to create a checkpoint. This leads to the algorithm shown in Figure 3.8 for creating optimal checkpoints.

```
Create a checkpoint at State 0;
Sum := Size(State 0);
n := 0;
for k := 1 to r(P,t) do
      begin
      n := n + 1;
      let S := size of the partial state that would be created at State k;
      if n * S > Sum then
        begin
        create a new checkpoint at State k;
        n := 0;
        Sum := Size(State k)
        end
      else begin
        create the partial state at State k;
        Sum := Sum + Size(State k)
        end
      end
```

Figure 3.8. Algorithm to Optimize Checkpointing.

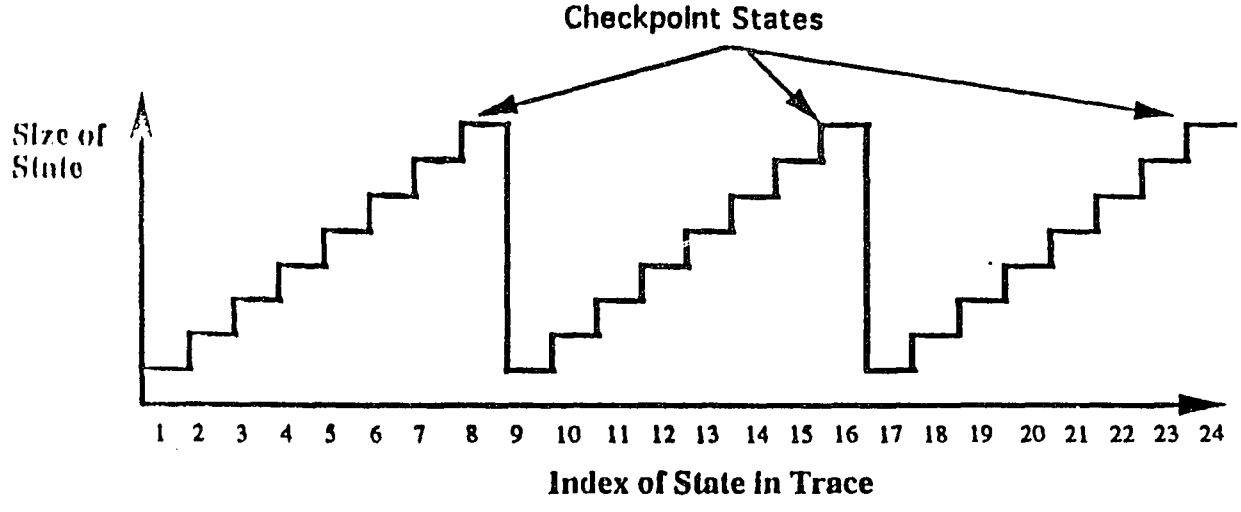

Figure 3.9. Graph of State Size.

3.4.2.1. An Illustration

We illustrate approximately how much memory can be saved using this modification of the MDS. For simplicity, assume that a unit of memory is the average amount of memory that would be modified as a result of executing a single program statement. Further assume that each statement's execution adds one such unit of memory to the size of the partial state it creates from the previous partial state. In other words, we assume that each time a statement is executed, the list of variables changed since the last checkpoint increases, and the memory used by this list increases by a unit of memory. We note that this is a very pessimistic estimate of the increase in memory because:

1. the statement may modify a variable that has already been modified since the last checkpoint and is therefore already in the list; in this case the list would not grow; or
2. the statement might modify a variable in such a way that the variable receives the same value as it had in the checkpoint state, implying that if the variable were in the previous partial state, then the new partial state could be reduced by this variable.

Thus, the partial state might even decrease in size as a result of executing a statement, but we assume the worst, that it always increases in size. Under these assumptions, the amount of memory used by the trace in the MDS is as illustrated in Figure 3.9 and for each $k, 0 \le k \le n$, $S_k = k$. (In the figure we illustrate the case that n=8.) Thus, Equation(3.25) becomes

$$n^2 = S_0 + \sum_{k=0}^{n-1} k = S_0 + \frac{n(n-1)}{2} \approx S_0 + \frac{n^2}{2} \qquad (3.26)$$

and solving for $n \geq 0$,

$$n = \sqrt{2S_0} \tag{3.27}$$

In other words, the number of partial states between checkpoints is roughly the square root of twice the size of a checkpoint state, measured in the memory units described above. The average amount of space used per trace element is therefore also $\sqrt{2S_0}$, as compared to S_0 per trace element in the original MDS. Thus for each trace element we reduce space consumption by a factor of $\sqrt{S_0/2}$. This method of space reduction can therefore result in a significant shrinking of the size of the MDS.

4. HiTest: an Architecture for Highly Parallel Software Testing

In this section we present a special purpose, parallel machine architecture that can substantially reduce the total time needed for testing. Although it can improve the performance of any testing method, it is motivated in particular by the requirements of mutation analysis. Our work differs from other recent attempts to use parallel architectures to reduce testing effort [15, 37, 67] because it uses a dynamically reconfigurable multiple-SIMD (MSIMD) model. Our model, called HiTest, is unlike other reconfigurable MSIMD systems (e.g., PASM [60], MAP [49], and PM4 [6]) in many respects, and is designed for the express purpose of software testing.

4.1. Description of the Architecture

We have designed a machine model that is neither pure SIMD nor pure MIMD, but a hybrid of the two. Figure 4.1 illustrates its structure. Like an SIMD machine, it has M identical PEs with local memory units. Unlike an SIMD machine, it has not just 1, but N hosts. If $N = 1$, then our machine reduces to an SIMD model. But in general, all N hosts of our machine can control any subset of the PEs, except that no two hosts can control the same PE. In other words, the PEs can be partitioned into disjoint subsets allocated to distinct hosts. This allocation can be changed at any instruction cycle of the system.

While other multiple SIMD systems do exist, they do not have the functionality needed to support highly parallel software testing. PASM [60] allows each host to control 2^n PEs for variable n; the mapping of PEs to hosts is permanent even though n can be adjusted as required. This prevents it from being used optimally for testing.

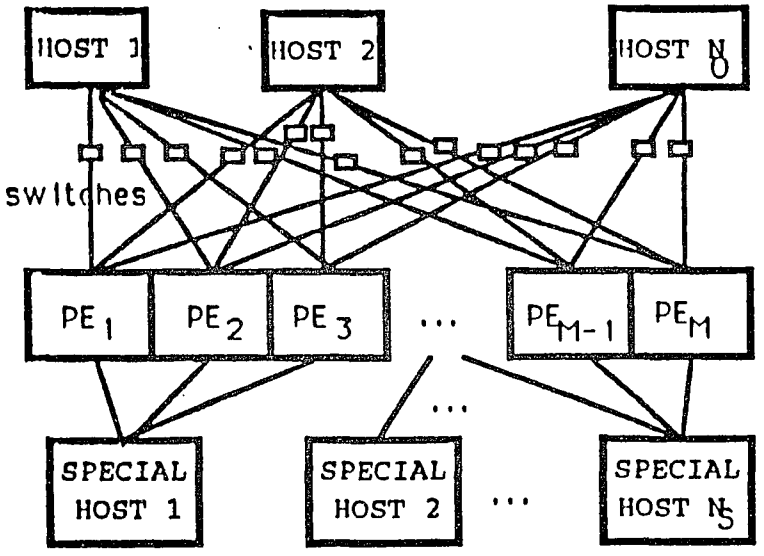

Figure 4.1. Structure of HiTest.

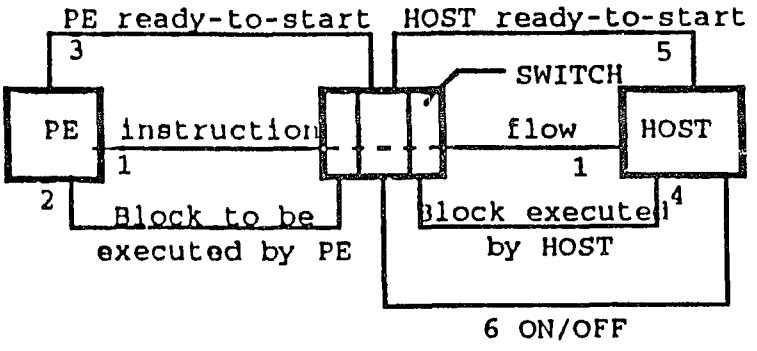

Figure 4.2. Schematic Wiring of the Switch.

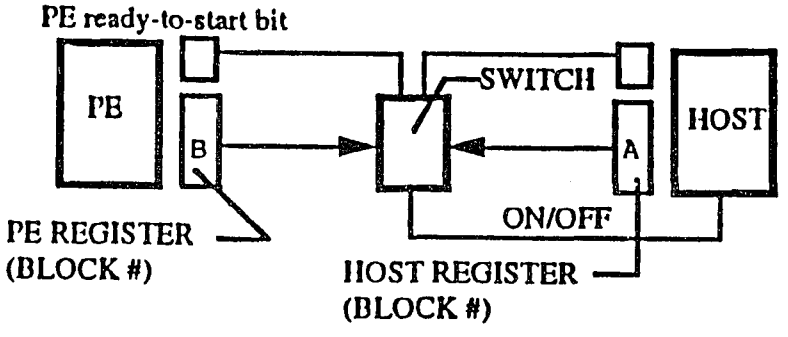

Figure 4.3. Operation of the Switch.

```
repeat
  Turn on own ready-to-start bit;
  Delay /* allow switches to turn on */;
  Turn off own ready-to-start bit;
  Give instruction  to PEs to turn off
    their own ready-to-start bits;

  EXECUTE BODY OF BASIC BLOCK;

  Give instruction to PEs to enter the
    next basic block number into their
    register B;
  Give instruction to PEs to turn on
    their ready-to-start bits;
  Turn off all switches;
forever
```

Figure 4.4. Operation of an Ordinary Host.

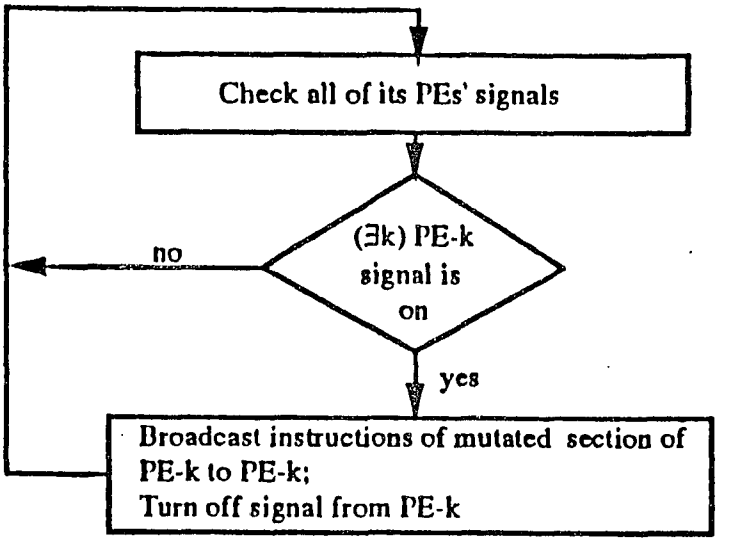

Figure 4.5. Operation of a Special Host.

Both MAP [49] and PM4 [6] allow each host to control groups of PEs using a crossbar switch, and both allow dynamic mapping, but they lack the special purpose interconnections we describe below that are so suitable for mutation analysis, and they have complex interprocessor and processor-to-memory networks that are not needed for testing.

In our system, which we call HiTest, the allocation of PEs to hosts can change dynamically as a program executes, and it depends upon the state of a system of $N \cdot M$ switches. A switch is interposed between each possible host-PE pair. Figure 4.2 illustrates the interconnections of the switch, a PE, and a host. In the figure, the line labeled "1" represents the flow of instructions, and lines 2 and 3 represent the demands on the PE. One line stands for "ready-to-start" and the other gives the number of the basic block to be executed next. Lines 4 and 5 represent signals from the host. One line indicates that the host is ready-to-start broadcasting and the other one gives the number of the basic block that the host will execute. The switch checks in parallel if both the PE and the host are "ready-to-start" and if the number of the basic block for execution is the same for both of them. If both conditions are true then it turns on the connection, making it possible for the host to broadcast to the PE.

Figures 4.3 and 4.4 represent the function of the switch and the operation of a host. In Figure 4.3, register A stores the number of the block the host controls and register B stores the number of the block the PE "has decided" to execute next. If the contents of registers A and B match and both ready-to-start bits are on, the switch opens the flow of instructions. Immediately afterwards, the host turns its ready-to-start bit off and transmits the instruction to turn off the ready-to-start bits of all of the PEs that are now controlled by this host. The host then begins broadcasting the block's instructions to the

PEs. When all instructions of the block are completed, the host broadcasts an instruction to each of these PEs to store the number of the block each is to execute next into its own B register and to turn on its ready-to-start bit. After that, the host turns off all of its switches (through line 6 of Figure 4.2).

Mutual exclusion, i.e., preventing the same PE from being controlled by more than one host, can be provided by one of the three following ways:

(1) *By ensuring that no two hosts execute the same basic block.* This may be done by guaranteeing that the number of blocks exceeds the number of hosts. For large enough programs, this is automatic; for small programs, one can break the largest blocks into smaller blocks until the total number equals the number of hosts.

(2) *By adding timing constraints to synchronize the hosts' executions.* In other words, we could force the hosts to execute so that no two could be starting the execution of the same block at the same time. (If there are two hosts that begin execution of the same block at the same time, one of them is redundant anyway.)

(3) *Through the use of special test-and-set-lock style hardware.* The ready-to-start bit of the PE can be treated as a lock variable; the first host to successfully match register A with the PE's register B would test the ready-to-start bit of the PE and turn it off simultaneously were it found to be on. We are treating "off" as set value and "on" as unset value. This is the most costly option.

The operation depicted in Figure 4.4 assumes that mutual exclusion is provided by either method (1) or (2) above. In the case that mutual exclusion is provided using special hardware, the only difference is that the emboldened box would be deleted from Figure

4.4, because at that point the ready-to-start bit would have been turned off by the hardware.

4.2. Software Testing on HiTest

HiTest can not only decrease the cost of mutation analysis, but it can also significantly decrease the total time for software testing in general. Below is a simplification of how this may be done.

4.2.1. Parallel Software Testing: A Simple Case

The way in which HiTest may be used for parallel software testing is illustrated by the following unrealistic but very illustrative example.

Assume that a program has K basic blocks, each of which requires the same execution time. Suppose our architecture is used for running this program on M different test inputs, where M is the number of PEs, so that each PE has its own input data. We consider how HiTest performs in the simplest case, when the number of hosts equals the number of basic blocks of the program. Each basic block is assigned to a unique host, which is responsible for executing it. All hosts begin to work on their basic blocks at the same time. Because all blocks need the same amount of execution time, all hosts finish controlling their subset of PEs at the same time, after which the host-PE relationships change and the process is repeated. Clearly all PEs are working all of the time because all K blocks of the program are executed in each cycle. The utilization factor in this synchronized operation mode is 1.

4.2.2. Parallel Execution of Mutants

Our machine can execute as many as M mutants in parallel. If the number of PEs is much larger than the number of mutants obtained from applying all of the mutation operators to a single statement or expression, then we can repeatedly apply the mutation operators to adjacent statements in the enclosing basic block until we have generated as close to M many mutants as possible, in order to take full advantage of the hardware. In this case, when we refer to a *mutated code section*, we do not necessarily mean a single statement, but the smallest sequence of statements that contains all of the mutated statements in the program. The only restriction is that a mutated section must be a subsequence of a basic block, i.e., it cannot extend across two blocks.

We decompose the basic block containing the mutated section into three blocks: the prefix, mutated section, and suffix. The first and third are regular blocks, executed as described above. The mutated section is assigned to and executed by *special hosts*, which, unlike ordinary hosts, are permanently assigned to a fixed number of PEs. When a PE needs to execute a mutated section, its special host takes over. During execution of the mutated code, this host controls the PE exclusively, although many mutated sections might be executed in parallel by different special hosts. The operation of the special hosts is shown in Figure 4.5. When a PE computes its successor block and finds that it is a mutated section, a signal is sent to the special host that is assigned to it. The host searches a map in its memory for an entry for this PE to determine which particular mutation of the mutated section has been assigned to the PE, and then transmits the appropriate instructions to the PE. When the mutated section is finished, the host checks all of its PEs' signals. If no signal is on, the host checks the PEs' signals again (a busy

form of waiting), otherwise it picks up any one of the PEs whose signal is on and repeats the preceding process. As we show below the optimal size of the queue of PEs waiting for the special host to be served is zero. This means that if the system parameters have been chosen to achieve optimal performance, then there should rarely be more than one PE waiting for service by the special host assigned to it. In case there is, we can use a method that is the simplest to implement, such as by choosing the PE whose "id" is smallest.

Assume again that a program has K basic blocks, each requiring the same amount of execution time, and that one of these blocks is a mutated section. By the above reasoning, we may assume that the number of mutants equals the number of PEs, and that each PE executes its own mutant. The special hosts are responsible for controlling execution of the mutated section, and all of the PEs are divided among these hosts. The remaining $K-1$ blocks are assigned to $K-1$ ordinary hosts. Because all mutants execute the same instruction stream on the same data, until the first point at which the mutated statement is executed, there is no advantage to scheduling the mutants among the PEs. Instead, we "preprocess" the program by executing it serially on a single processor until the mutated section is reached, and we then start each mutant on its own PE, with its initial state being the same, namely the state of the program just prior to execution of the mutated section. A PE reaching the mutated section sends a signal to its assigned special host and this host then assumes control of the PE, sending it the instructions of the mutated section. When all instructions of the mutated block are completed, the special host broadcasts an instruction to the PE to store the number of the block to be executed next into its register B (see Figure 4.3) and turn on its ready-to-start bit. This

allows the PE to resume execution of ordinary blocks concurrently with other PEs controlled by the ordinary hosts. While executing ordinary blocks, all PEs work as described in Section 4.2.1. If there are enough special hosts, then no PE ever has to wait for work to do, and so all PEs are busy almost always, either working on mutated sections or on ordinary blocks, for which no waiting is involved.

4.3. Performance

The model of the program in which every basic block has exactly the same execution time as any other, that was used in 4.2.1 and 4.2.2, is very unrealistic. Further more, the number of the ordinary hosts of HiTest may be not equal to or even not close to the number of the basic blocks of the program. The problem of tuning the parameters of HiTest to optimize the performance with realistic programs is the problem considered in this chapter. Our goals also are to predict the potential speed-up of mutation analysis and ordinary software testing using HiTest, and to determine the optimal parameters of the architecture, such as the ratio of special to ordinary hosts.

4.3.1. Preliminaries

We assume that a program has K blocks, $B_1, \ldots, B_K$, and associated to these blocks is a vector of stationary probabilities $(p_1, p_2, \ldots, p_K)$ such that p_j is the probability that the program is executing block B_j at an arbitrary time. We let $T_1, \ldots, T_K$ denote the amounts of time needed to execute blocks $B_1, \ldots, B_K$, respectively.

We assume that the size of the mutated section is much smaller than that of the program. We let $N = N_O + N_S$ where N_O and N_S are the numbers of ordinary and special hosts respectively, and we assume that the number of PEs, M, is orders of magnitude

larger than N (for instance, $M > N^{1.5}$).

At any given time there are zero or more PEs executing a particular block. We let M_{it} denote the number of PEs execution B_i at time t, and M_i, the average number of PEs that execute B_i over the course of time that the system executes the program.

$$M_i = \lim_{t \to \infty} \frac{\int_0^t M_{it} dt}{t}$$

We assume that the number of PEs that reach the point at which they are ready to execute a given block, say B_j, is constant over time. Thus, as time progresses, the number waiting for a host to execute B_j increases linearly as function of time until it reaches the point when some host "picks up" B_j and executes it, at which time the number of waiting PEs becomes zero again (see Figure 4.6).

Each PE is either busy executing instructions of a mutant, or it is in a waiting state. The total amount of time that all PEs spend waiting is denoted T_{wait}. The sum of their total waiting time and total execution time is denoted T_{total}. We let τ denote the overhead of the system per unit time, i.e., $\tau = \frac{T_{wait}}{T_{total}}$. The *utilization factor* $\hat{\mu}$ of the collection of PEs is the fraction of PEs busy at a given time. At any given time the number of the PEs executing mutated sections (or waiting for its execution) is C. We define μ to be the fraction of the remaining $M - C$ PEs that are executing instructions (as opposed to waiting for a host). Because we assume that the size of the mutated section is much smaller than that of the program, C is much smaller than M, and μ and $\hat{\mu}$ are very close in value, but we can calculate μ without making any assumptions about the number of

the special hosts N_S. Therefore, in our analysis we calculate μ rather than $\hat{\mu}$ and treat μ as the utilization factor. It follows that $M_i \approx \mu p_i M$.

4.3.2. Synchronous Mode

All of the examples in Section 4.2. are based upon *synchronous mode* operation, in which all N ordinary hosts begin the execution of their blocks simultaneously. This method of operation is suitable only in the case that each block requires the same amount of execution time. Even if the blocks in a program are of different sizes, a program can be decomposed into code sections to be assigned to hosts such that each code section takes roughly the same amount of execution time. We consider three cases.

Case 1. If $K \ll N_O$ then each basic block of the program is divided into equal sections which are assigned to unique hosts in such a way that the execution of each section requires roughly the same amount of time. This division of the blocks into sections makes the problem of finding the overhead analogous to the problem of internal memory fragmentation when paging is used. Only the last section of each block may require less time than the others. On average, we can assume that the last section requires half the time of the other section. Another way to state this is that there are K final sections assigned to K hosts and, at any given time, half of these K hosts may be idle, each having already completed its execution. Thus, $K/2$ hosts out of N_O are idle at any time. Since there is no dependence between PEs and the hosts that control them, we assume that the fraction of PEs that are idle is the same as the fraction of idle hosts, i.e.,

$$\tau = \frac{K}{2N_O}.$$

Since the utilization factor μ is $1 - \tau$ we have the following:

$$\mu = \frac{2N_O - K}{2N_O}. \tag{4.1}$$

Case 2. If $K \gg N_O$ then each code section consists of several basic blocks. In this case, a PE may have to wait for a host when it is ready to execute a new block, because the host is scheduled to execute a different block. Since we have N_O hosts we can execute N_O out of K different blocks at any time. It is natural to assume that the utilization factor μ is just N_O/K, but this is false. Since we assume that the number of PEs waiting to execute a block increases linearly with time (see Section 4.3.1) until it reaches a maximum at the time a host begins to execute the block, the utilization factor is about twice as great, i.e.,

$$\mu \approx \frac{2N_O}{K}.$$

More precise analysis is presented in Appendix E. We calculated there that

$$\mu = \frac{2N_O}{2N_O + K}. \tag{4.2}$$

For an architecture with one host such as the SIMD model used in [37], $\mu \approx 2/K$, which is twice as great as the expression we arrived at informally in Section 2.5 above. This result is higher than the predicted one because when we derived Equation 4.2 we used a method of scheduling the blocks' executions that is close to the optimal one, namely that blocks are distributed among hosts so that the total execution time of the blocks assigned to a host is the same for all hosts, and each host executes its blocks in round-robin order. It is worth noting that (4.2) is also a good approximation to (4.1) when $K \ll N_O$ (i.e., in

Case 1).

Case 3. When N_O and K are close in value, the methods of cases 1 and 2 may be used, but we have not proved that Equation 4.2 is a good approximation. However, as we show below the utilization factor of the Equation 4.2 can be reached using the asynchronous mode also. And since the asynchronous mode is easier to implement it is more likely be used. All these make the results for synchronous mode the subject of least importance.

4.3.3. Asynchronous Mode

In *asynchronous mode*, hosts control PEs without any communication or synchronization with each other. Our objective is to determine, for this mode of operation, the optimal assignment of blocks to hosts and hosts to blocks, the way the system should operate, and finally, the utilization factor.

Case 1. $K \ll N_O$. In this case, general speaking, several hosts are responsible for the same block. We let n_i denote the number of ordinary hosts assigned to execute block B_i. Thus $N_O = \sum_{i=1}^{K} n_i$. By the overhead of a block B_j, we mean the sum of the amounts of time that all PEs spend waiting to execute B_j per unit of time. We let τ_j denote the overhead of B_j. The following theorem characterizes the method of achieving optimal performance:

Theorem 1.

If block B_j requires T_j units of execution time, and n_j hosts are assigned to B_j, then to minimize the overhead associated with B_j, these hosts should begin their executions of B_j at times $t, \quad t+T_j/n_j, \quad t+2T_j/n_j, \quad t+3T_j/n_j, \ldots, \quad t+((n_j-1)T_j)/n_j.$

Proof. We can write the overhead of B_j, τ_j, as follows:

$$\tau_j = \frac{1}{T_j} \sum_{i=1}^{n_j} \mu M \frac{p_j}{T_j} t_i \frac{t_i}{2} = \mu M \frac{p_j}{2T_j^2} \sum_{i=1}^{n_j} t_i^2 \tag{4.3}$$

where t_i is the time between the start of the $(i-1)st$ and that of the ith host, for $i \neq 1$, and t_1 is the time between the start of the $N_j th$ host and the first host.

This is an optimality problem in which we need to minimize τ_j subject to the constraint

$$\sum_{i=1}^{n_j} t_i = T_j$$

$$0 \leq t_i \leq T_j, \quad \text{for } all\ i.$$

This is a particular case of the problem considered in Appendix C. The solution (C2) from this appendix is

$$for\ all\ i \quad \frac{\partial t_i^2}{\partial t_i} = const \ \ independent\ of\ i\,.$$

implying that t_i is constant for all i, or that

$$t_1 = t_2 = \cdots = t_{n_j} = \frac{T_j}{n_j} \tag{4.4}$$

which was the result to be proved.

Next we want to determine how the hosts are distributed among blocks. We assume that hosts that are responsible for the same block are scheduled according to Theorem 1. Thus from (4.3) and (4.4) the overhead involved in waiting for B_j per unit of time is

$$\tau_j = \mu M \frac{p_j}{2T_j^2} \frac{T_j^2}{n_j} = \frac{\mu M p_j}{2n_j} \tag{4.5}$$

To find the optimal distribution we need to minimize $\sum_{j=1}^{K} \tau_j$ subject to constraint

$$\sum_{j=1}^{K} n_j = N_O$$

$$for\ all\ j \quad n_j > 0$$

This is once again the problem solved in the Appendix C.

$$\frac{\partial \tau_j}{\partial n_j} = \frac{\partial \frac{\mu M p_j}{2n_j}}{\partial n_j} = -\frac{\mu M p_j}{2n_j^2} = constant$$

and it is independent of j. Therefore we can define α so that:

$$\frac{\partial \tau_j}{\partial n_j} = -\frac{\mu M p_j}{2n_j^2} = -\frac{1}{2\alpha} \tag{4.6}$$

There are $K+1$ equations with $K+1$ unknowns:

$$\sum_{j=1}^{K} n_j = N_O \tag{4.7}$$

$$n_1^2 - \alpha\mu M p_1 = 0$$

$$n_2^2 - \alpha\mu M p_2 = 0$$

$$\cdots$$

$$n_K^2 - \alpha\mu M p_K = 0$$

from which we obtain

$$\text{for } all\ j \quad n_j = \sqrt{\alpha\mu M p_j} \tag{4.8}$$

This means that the number of hosts controlling block B_j is directly proportional to the square root of the probability that block B_j is being executed. Substituting $\sqrt{\alpha\mu M p_j}$ for n_j in Equation 4.7 we have

$$\sum_{i=1}^{K} \sqrt{\alpha\mu M p_i} = N_O$$

so that

$$\alpha = \left(\frac{N_O}{\sum_{i=1}^{K} \sqrt{\mu M p_i}} \right)^2$$

Substituting α into (4.8):

$$n_j = \frac{N_O \sqrt{p_j}}{\sum_{i=1}^{K} \sqrt{p_i}}$$

Then using (4.5) we have:

$$\tau_j = \frac{\mu M p_j}{2n_j} = \frac{\mu M p_j \sum_{i=1}^{K} \sqrt{p_i}}{2N_O \sqrt{p_j}} = \frac{\mu M \sqrt{p_j} \sum_{i=1}^{K} \sqrt{p_i}}{2N_O}$$

so that

$$\tau = \frac{\sum_{j=1}^{K} \tau_j}{M} = \frac{\mu \sum_{i=1}^{K} \sqrt{p_j} \sum_{j=1}^{K} \sqrt{p_i}}{2N_O} \tag{4.9}$$

From Appendix D we see that K is a good approximation for $\sum_{(i=1 j=1)}^{K} \sqrt{p_i}\sqrt{p_j}$, giving more pessimistic results that are nonetheless satisfactory. Thus, substituting in (4.9) above,

$$\tau = \frac{K\mu}{2N_O} = \frac{K(1-\tau)}{2N_O}$$

implying

$$2\tau N_O = K - K\tau$$

or that

$$\tau = \frac{K}{2N_O + K}$$

and that

$$\mu = \frac{2N_O}{2N_O + K}$$

Case 2 $K \gg N_O$. If K_i is the number of the blocks that are executed by the host H_i, then $\sum_{i=1}^{N_O} K_i = K$. Let Θ_i be time required by host H_i to execute all of its blocks, i.e., if H_i is assigned blocks $B_{i1}, B_{i2}, \cdots B_{iK_i}$, then

$$\Theta_i = \sum_{j=1}^{K_i} T_{i_j}$$

and

$$\sum_{i=1}^{N_O} \Theta_i = \sum_{j=1}^{K} T_j = T \tag{4.10}$$

where T_{i_j} is the execution time of block B_{i_j} . (4.10) is correct because each block is assigned to exactly one host. Let $\tilde{\tau}_i$ be the total waiting time of all PEs waiting to execute any of the blocks assigned to host H_i per unit of time. Then

$$\tilde{\tau}_i = \sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}} \mu M \frac{\Theta_i}{2} = \mu M \frac{\Theta_i}{2} \sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}}$$

Our goal is to divide all K blocks into N_O nonintersecting subsets so that the total overhead τ is minimal. Stated formally, we seek a distribution of the blocks to minimize

$$\tau = \frac{\sum_{i=1}^{N_O} \tilde{\tau}_i}{M} = \frac{\mu}{2} \sum_{i=1}^{N_O} (\Theta_i \sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}}) \tag{4.11}$$

subject to constraint (4.10).

Assume that we have the optimal distribution. Assume also that two different hosts require times Θ_1 and Θ_2 to complete one cycle of execution of their respective subsets of blocks, and that $\Theta_1 \neq \Theta_2$. Suppose that we could exchange n_1 identical (same probabilities p_1 and same execution times T_1) blocks of the first host with n_2 identical blocks of the second host such that the times of one cycle of the hosts do not change.

$$T_1 n_1 = T_2 n_2 = t \tag{4.12}$$

Using Equation 4.12, the change of the overhead $\Delta\tau$ is:

$$\Delta\tau = \frac{\mu}{2}(-\Theta_1(n_1\frac{p_1}{T_1} - n_2\frac{p_2}{T_2}) + \Theta_2(n_1\frac{p_1}{T_1} - n_2\frac{p_2}{T_2}))$$

$$= \frac{\mu}{2}t(\Theta_2 - \Theta_1)(\frac{p_1}{T_1^2} - \frac{p_2}{T_2^2})$$

From the assumption of optimality of the initial distribution it follows that $\Delta\tau$ must be positive. Thus, both expressions in the parentheses must have the same sign. Thus, if $\Theta_2 > \Theta_1$ then p_1/T_1^2 must be greater than p_2/T_2^2. This means that if we rearrange all the hosts in order of increasing Θ then from host to host p/T^2 gradually decreases. It follows that for each host H_i each of the parameters $p_{i_j}/{T_{i_j}}^2$, $i = 1 \cdots K_i$, is close in value: we let α_i approximate the average value of the $p_{i_j}/{T_{i_j}}^2$ values for host H_i. Then $\tilde{\tau}_i$ can be rewritten as

$$\tilde{\tau}_i = \mu M\frac{\Theta_i}{2}\sum_{j=1}^{K_i}\frac{p_{i_j}}{{T_{i_j}}^2}T_{i_j} \approx \mu M\frac{\Theta_i}{2}\alpha_i\sum_{j=1}^{K_i}T_{i_j} = \mu M\alpha_i\frac{\Theta_i^2}{2}. \qquad (4.13)$$

Because we expect the difference in values to be very small we henceforth treat (4.13) as a strict equality. Combining (4.13) and (4.11), and adding the constraint (4.10) we have a new formalization of our optimal problem: We seek to minimize

$$\tau = \frac{\mu}{2}\sum_{i=1}^{N_O}\alpha_i\Theta_i^2$$

subject to the constraint

$$\sum_{i=1}^{N_O}\Theta_i = T$$

From Appendix C we have

$$\frac{d(\alpha_i \Theta_i^2)}{d\Theta_i} = const$$

or

$$\alpha_i \Theta_i = const \tag{4.14}$$

Thus, for each host the product of the length of the cycle Θ_i and α_i is the same. This makes it possible to create an optimal design of the system by specifying an optimal distribution of the blocks among the hosts.

From (4.14) we can assume that for all i the expression $\Theta_i \alpha_i$ may be represented by β. Thus

$$\beta = \Theta_i \alpha_i = \frac{\sum_{i=1}^{N_O} \alpha_i \Theta_i}{N_O}$$

From (4.13) it is not hard to see that

$$\sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}} = \alpha_i \Theta_i$$

Thus,

$$\beta = \frac{\sum_{i=1}^{N_O} \sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}}}{N_O} = \frac{\sum_{j=1}^{K} \frac{p_j}{T_j}}{N_O} \tag{4.15}$$

$$\tau = \frac{\mu}{2} \sum_{i=1}^{N_O} \Theta_i^2 \alpha_i = \frac{\mu\alpha}{2} \sum i=1^{N_O} \Theta_i = \frac{\mu\alpha}{2} T \tag{4.16}$$

Substituting (4.15) into (4.16) we can get:

$$\tau = \frac{\mu T}{2}\frac{\sum_{j=1}^{K}\frac{p_j}{T_j}}{N_O} = \frac{\mu TK}{2N_O}\frac{\sum_{j=1}^{K}\frac{p_j}{T_j}}{K} = \mu TK2N_O E[\frac{p_j}{T_j}] \approx$$

$$\frac{\mu K}{2N_O}T\frac{E[p_j]}{E[T_j]} = \frac{\mu K}{2N_O}T\frac{\sum_{1}^{K}p_j}{\sum_{1}^{K}T_j} = \frac{\mu K}{2N_O}$$

where $E[x]$ denotes the expected value of x. Since $\tau + \mu = 1$

$$\mu = \frac{2N_O}{K+2N_O}$$

Case 3 . In cases 1 and 2 we considered two possibilities: $K \gg N_O$ and $N_O \gg K$. It is more realistic to assume that a program may have both large blocks and small blocks, and that the methods used in cases 1 and 2 may both need to be applied for a "real" program. In this case, some large blocks are decomposed and executed by several hosts, some small blocks are grouped and executed by the same host, and blocks of medium size are assigned to dedicated single hosts. We use the following notation to clarify these ideas.

N_s, N_m,and N_b are the total numbers of hosts that execute small blocks, medium-size blocks, and large blocks respectively;

K_s, K_m, and K_b are the numbers of small, medium, and large blocks, respectively, in the program;

p_s, p_m, p_b are the total probabilities of all small, medium, and large blocks, respectively, being executed;

τ_s, τ_m, τ_b are the total overhead per unit of time spent waiting for small, medium, and large blocks, respectively.

With these definitions, we can imagine the hosts of the machine as being partitioned into the following three disjoint sets:

$N_b < K_b$ | $N_m = K_m$ | $K_s < N_s$

The rightmost part contains hosts that execute "big" blocks, each of these blocks is executed by several hosts. The left part represents the hosts that execute "small" blocks; several of these blocks are executed by one host. The middle part of the figure consists of all the hosts each of which executes exactly one block of some average size. The middle part of the figure does not really have well-defined boundaries; we could redraw the figure instead as follows,

---

$N_b \leq K_b$ | $K_s \leq N_s$

---

eliminating the middle part and concentrating instead on finding the optimal boundary between the right and the left parts of the figure. This boundary can be placed in exactly $N_m + 1$ many positions, corresponding to those hosts that execute only one block each, since we can put any of these hosts in the left or right cell of the partition. From the previous analysis it follows directly that the overhead in each case can be expressed by the same formula:

$$\tau_s = \frac{\mu p_s K_s}{2N_s} \tag{4.17}$$

and

$$\tau_b = \frac{\mu p_b K_b}{2N_b} \tag{4.18}$$

The hosts adjacent to the boundary can be moved to the right or left cell without changing the overhead. (That is the condition of optimality).

Since the formulae for the right cell and the left cell are the same we can represent them by a single formula. Let $\bar{\tau}$ be the overhead of the right or the left cell, and let $\bar{p}$, $\bar{K}$, $\bar{N}$ represent the respective probabilities, numbers of blocks, and numbers of hosts for small or big blocks. Then we can write

$$\bar{\tau} = \frac{\mu \bar{p} \bar{K}}{2\bar{N}}$$

Now assume that we remove one of the hosts from either cell and assign one block of probability b to that host. Without loss of generality, assume that we removed the host from the left cell. Then there are $\bar{N} - 1$ hosts remaining in this cell, and $\bar{K} - 1$ blocks. Let $\hat{\tau}$ be the overhead of this cell together with the isolated host. Then

$$\hat{\tau} = \frac{\mu(\bar{p} - b)(\bar{K} - 1)}{2\bar{N} - 2} + \frac{\mu b}{2}$$

$$= \mu \bar{p} \frac{\bar{K} - 1}{2\bar{N} - 2} + \mu \frac{b}{2} - \mu b \frac{\bar{K} - 1}{2\bar{N} - 2}$$

$\bar{\tau} - \hat{\tau} = 0$ is a condition of optimality. Thus,

$$\frac{\bar{\tau}-\hat{\tau}}{\mu} = \frac{\bar{p}\bar{K}\bar{N} - \bar{p}\bar{K} - \bar{p}\bar{K}\bar{N} + \bar{p}\bar{N}}{2\bar{N}(\bar{N}-1)} - \frac{b\bar{N} - b - b\bar{K} + b}{2(\bar{N}-1)}$$

$$= \frac{\bar{p}(\bar{N}-\bar{K})}{2\bar{N}(\bar{N}-1)} - \frac{b(\bar{N}-\bar{K})}{2(\bar{N}-1)}$$

$$= \frac{(\bar{N}-\bar{K})}{2(\bar{N}-1)} \left(\frac{\bar{p}}{\bar{N}} - b\right)$$

Solving, we get

$$\frac{(\bar{\tau}-\hat{\tau})2(\bar{N}-1)}{\mu(\bar{K}-\bar{N})} = \frac{\bar{p}}{\bar{N}} - b \tag{4.19}$$

and since $\bar{\tau} - \hat{\tau} = 0$, the left side of (4.19) is zero, so that

$$b = \frac{\bar{p}}{\bar{N}}$$

implying that

$$b = \frac{p_s}{N_s} = \frac{p_b}{N_b} = \frac{p_s + p_b}{N_s + N_b} = \frac{1}{N_O}$$

Thus, the number of hosts that must be assigned to each part is directly proportional to total probability of blocks of that part. Using this, we can rewrite (4.17) and (4.18) as follows:

$$\tau_s = \frac{\mu K_s p_s}{2N_s}$$

$$\tau_b = \frac{\mu K_b p_b}{2N_b}$$

Solving for τ,

$$\tau = \tau_s + \tau_b = \frac{\mu}{2}\left(\frac{K_b p_b}{2N_b} + \frac{K_s p_s}{2N_s}\right)$$

$$= \frac{\mu}{2}\frac{1}{N_O}(K_b + K_s) = \mu\frac{K}{2N_O}$$

$$\tau = \mu\frac{K}{2N_O}$$

or, since $\tau + \mu = 1$,

$$\mu = \frac{2N_O}{(2N_O+K)}$$

which is the same as (4.2). Because synchronous and asynchronous modes yield the same utilization factor, the remaining analysis is based upon the asynchronous mode.

4.3.4. Optimal Parameters of the Architecture

If N, the total number of special and ordinary hosts in the machine is fixed, what are the numbers N_S of special hosts and M of PEs that optimize the performance of the system? We answer this question below.

4.3.4.1 The Optimal Number of Special Hosts.

First consider how many special hosts are optimal. On the one hand, the bottleneck in mutant execution arises when a PE needs to execute a mutated section, but it must wait for the special host assigned to it to be ready since that host might be in the midst of controlling a different PE. This suggests that N_S be as large as possible to avoid these waits. But as N_S increases, the number of ordinary hosts N_O decreases, limiting the number of

different blocks that can be executed in parallel. Thus, N_S should not be too large.

To solve this problem, we treat special hosts as ordinary hosts having attached queues in case they are busy when a PE requests service. Let B_M be the block containing the mutated section, let t_M be the time needed to execute a mutated section, and m, the total number of PEs waiting for the host to execute it. Then the real utilization factor is:

$$\mu^* = \frac{M-m}{M}\mu \tag{4.20}$$

where μ can be taken from the previous calculations (4.14) by substituting N_O by $N - N_S$

$$\mu = \frac{2(N - N_S)}{2(N - N_S)+K}$$

since N_S is the number of hosts that are assigned to do the mutated section.

Consider the equation

$$\frac{p_M}{T_M}(M-m)\mu = \frac{N_S}{t_M}$$

The left part is the number of mutants that arrived at this mutated section per unit of time. The right part is the number of mutants that complete the mutated section per unit of time. So in a state of equilibrium they have to be equal. From the last equation we have:

$$(M-m) = \frac{N_S T_M}{\mu t_M p_M} \tag{4.21}$$

Substituting $M-m$ by (4.21) in (4.20) we have:

$$\mu^* = \frac{N_S T_M}{t_M p_M \mu M}\mu = \frac{N_S T_M}{t_M p_M M}$$

Showing that the utilization factor increases monotonically with N_S. Thus the optimal value of N_S must be constrained. More precisely, Equation 4.21 has to be rewritten as:

$$M - m = \min \begin{cases} \dfrac{N_S T_M}{t_M p_M \mu} \\ M \end{cases} \quad or$$

Thus the optimal value of N_S can be obtained from

$$\frac{N_S T_M}{t_M p_M \mu} = M$$

From the above we received the following optimal value of N_S:

$$N_S = \frac{\mu M t_M p_M}{T_M} \tag{4.22}$$

which corresponds with a queue length of mutants waiting for the execution of the mutated section and

$$\mu^* = \frac{\mu M t_M p_M T_M}{T_M t_M p_M M} = \mu = \frac{2(N - N_S)}{2(N - N_S) + K} \tag{4.23}$$

The above analysis shows that the optimal queue length is zero, implying that in order to maximize speed-up, no PE should ever have to wait for a special host. Thus, each time we add more PEs to the system, because we are essentially adding the potential to execute more mutants, to maintain optimal performance, we need to increase the number of special hosts to control them. Thus, even assuming that PEs have no cost, given that $N = N_O + N_S$ is fixed, there is an optimal number of PEs for peak

performance of the system.

The expression (4.23) with N_S taken from (4.22) has a weakness; it contains μ as a parameter. So we consider instead (4.22) and (4.23) together as a system. From (4.23) we have:

$$\frac{K}{2(N - N_S)+K} = 1 - \mu$$

implying

$$K = (1 - \mu)(2\,(N - N_S) + K)$$

or that

$$2N_S(1 - \mu) = 2N - 2\mu N - \mu K$$

from which we developed

$$N_S = \frac{2N - 2\mu N - \mu K}{2(1-\mu)} = N - \frac{\mu K}{2-2\mu} \tag{4.24}$$

From (4.22) we have

$$N_S = \frac{\mu M t_M p_M}{T_M} = \beta\mu \tag{4.25}$$

where β is defined as

$$\beta = \frac{M t_M p_M}{T_M} \tag{4.26}$$

So from (4.24) and (4.25) we have

$$N - \frac{\mu K}{2-2\mu} = \beta\mu$$

or

$$2N(1-\mu)-\mu K = 2\beta(\mu-\mu^2) \tag{4.27}$$

From the square equation (4.27) we can develop μ as a function of n, K, and β using the standard methods, but the result looks quite complicated and hard to interpret or analysis. Instead we consider the following equation:

$$2N(1-\mu) - \mu K = 2\beta\mu \tag{4.28}$$

Since $\mu < 1$, the value of μ that we obtain from this equation will be smaller than the actual value that we can develop from (4.27). Since we are not trying to obtain exact results, the more pessimistic value from (4.28) will satisfy us. Thus,

$$\mu = \frac{2N}{2\beta+K+2N} \tag{4.29}$$

and the expression for μ^* is the same:

$$\mu^* = \frac{2N}{2\beta+K+2N}$$

4.3.4.2. Optimum Correlation between the Number of Hosts and the Number of PEs

Let us denote the speed-up of the system by SU. The speed-up is defined by formula

$$SU = M\mu \tag{4.30}$$

In this section we seek a value of M that maximizes SU. This value is the optimal number of PEs, neglecting their cost. From (4.29) and (4.30)

$$SU = \frac{2MN}{2\beta+K+2N} \tag{4.31}$$

From (4.26) $\beta = Mt_M \frac{p_M}{T_M}$.

Consider the first derivative of SU with respect to M

$$\frac{dSU}{dM} = \frac{2N}{2\beta+K+2N} - \frac{4MN}{(2\beta+K+2N)^2}\frac{d\beta}{dM} \tag{4.32}$$

Now suppose first that

$$\beta = \gamma M \tag{4.33}$$

for some constant γ. In this case β is linear function of M. This corresponds the case when t_M does not depend on M. In other words optimum value of M is small enough that all PEs can execute the mutants with the same mutated statement. For this case from (4.32) and (4.33) we have:

$$\frac{dSU}{dM} = \frac{2N}{2\beta+K+2N} - \frac{4MN\gamma}{(2\beta+K+2N)^2}$$

Applying (4.33) once more we have:

$$\frac{dSU}{dM} = \frac{2N}{2\beta+K+2N}\left(1 - \frac{2\beta}{2\beta+K+2N}\right) > 0$$

This value is always positive since

$$\frac{2N}{2\beta+K+2N} > 0$$

and

$$1 - \frac{2\beta}{2\beta+K+2N} > 0$$

because

$$\frac{2\beta}{2\beta+K+2N} < 1$$

Therefore, the case we just considered is unrealistic; M grows beyond the values for which t_M is independent of M.

The second case to consider is when

$$t_M = \gamma M$$

This corresponds to the situation in which execution time is a linear function of the number of mutants. This is reasonable assumption since for the twice as many PEs we would need twice as many mutated sections. From (4.34) and (4.26),

$$\beta = \frac{\gamma M^2 pK}{T_K} \tag{4.35}$$

Letting $\gamma' = \gamma pK / T_K$, we can write $\beta = \gamma' M^2$. Equation 4.31 becomes

$$SU = \frac{2MN}{2\gamma' M^2 + K + 2N}$$

In this expression SU increases with increasing M for small values of M, and for larger M , M^2 in the denominator dominates and SU decreases. The maximum point can be obtained from the equation

$$\frac{dSU}{dM} = 0 \tag{4.36}$$

Rewriting (4.35) using (4.36) and (4.32) we have:

$$\frac{dSU}{dM} = \frac{2N}{2\beta + K + 2N} - \frac{4MN}{(2\beta + K + 2N)^2} 2\gamma' M$$

$$= \frac{2N}{2\beta + K + 2N} \left(1 - \frac{4\gamma' M^2}{(2\beta + K + 2N)}\right)$$

Setting this derivative to zero, and substituting β for $\gamma' M^2$, we get

$$1 - \frac{4\beta}{2\beta+K+2N} = 0$$

or that

$$4\beta = 2\beta+K+2N$$

from which we get

$$\beta = \frac{K+2N}{2} \tag{4.37}$$

Using (4.37) and (4.29), we substitute into (4.25) for the values of N_S and μ, and obtain

$$N_S = \frac{K+2N}{2}\frac{2N}{2\beta+K+2N} \tag{4.38}$$

$$= \frac{(K+2N)N}{K+2N+K+2N}$$

$$= \frac{(K+2N)N}{2(K+2N)}$$

$$= \frac{N}{2}$$

Therefore, the optimal number of PEs should be such that the number of hosts that control the sequential (mutated) section is equal to the number of hosts that control the non-mutated blocks, i.e.,

$$N_S = N_O = \frac{N}{2} \tag{4.39}$$

The utilization factor for the part of the machine involved in executing non-mutated code is

$$\mu = 2N_O \; / \; (2N_O + K)$$

and the utilization factor for the part executing mutated sections is 100%. Therefore, the utilization factor for the whole system is:

$$\mu > 2N_O \; / \; (2N_O + K)$$

Thus, if K is approximately equal to N_O, the utilization factor of the entire system is very high, and will increase rapidly as N_O increases relative to K.

4.3.4.3. Speedup of the System as a Function of a Program Size

Let us define a metric that measures the size of a program as a geometrical average of the number of the statements $l(P)$ in the program and the number of variables $v(P)$.

$$SIZE(P) = \sqrt{l(P)\, v(P)}$$

Our machine has N_O regular hosts and N_S special hosts. Equation 4.39 shows that for optimal performance they should be equal. Below we suppose that our system has enough PEs for the optimal performance described in 4.3.4.2.

Let us assume that the units of time we use for measurement are such that exactly one instruction is executed in each unit of time. This is a simplification of actual architectures, where execution times vary from instruction to instruction, but for the purpose of this analysis it suffices. Then using Equation 4.21 and the fact that $N_S = N_O$ and $m = 0$ for optimal performance, we get

$$M = N_O \frac{T_M}{t_M p_M \mu}$$

t_M is both the execution time of and the number of the statements in the mutated section.

Setting $\mu = \frac{2N_O}{2N_O+K}$ we get

$$M = \frac{N_O(2N_O+K)T_M}{t_M p_M 2N_O} = \frac{(2N_O+K)T_M}{2t_M p_M}$$

whence

$$t_M = \frac{(2N_O+K)T_M}{2Mp_M} \tag{4.40}$$

On the other hand, if each statement makes as average d mutants then

$$M = t_M d$$

or

$$t_M = \frac{M}{d} \tag{4.41}$$

Combining (4.40) and (4.41) we get

$$\frac{M}{d} = \frac{(2N_O+K)T_M}{2Mp_M}$$

or

$$2M^2 p_M = (2N_O+K)lT_M$$

implying that

$$M = \sqrt{\frac{(2N_O+K)lT_M}{2p_M}}$$

On average, the best approximation for $\frac{T_M}{p_M}$ is the number of statements $l(P)$ in the program. The number of mutants from the statement d can be expressed as

$l \approx d_0 v(P)$ for some constant d_0, and so M can be approximated as follows:

$$M \approx \sqrt{\frac{(2N_O+K)d_0 v(P) l(P)}{2}}$$

and the speed-up by

$$SU = M\mu \approx \frac{2N_O}{2N+K}\sqrt{\frac{(2N_O+K)d_0 v(P) l(P)}{2}}$$

$$= \frac{\sqrt{2d_0}\ N_O}{\sqrt{2N_O+K}}\sqrt{v(P) l(P)} = \frac{d_1 N_O}{\sqrt{2N_O+K}} SIZE(P)$$

Thus, the total speed-up is proportional to the size of the program. If N_O and K are comparable in values it is proportional to the square root of N_O also. Therefore, the total speed-up of the system for realistic size software may be very significant.

4.4. Software Simulation of HiTest

Here we discuss the implementation of a software simulation of the HiTest architecture. The objective of the simulation is to test the theoretical results of the performance analysis from Section 4.3.

The core of the HiTest simulator is a set of three modules. The most complex one is the hardware simulation module; it operates by simulating the time consumption of every hardware unit of the system and the connections between them. The second module simulates the booting; it makes the system ready to start parallel execution. The third module simulates the execution path of the program or mutant; it is called whenever there is a need to determine which basic block is to be executed next. All three modules are written in the C programming language. The output of the simulation is a table that

compares the calculated theoretical utilization factor (formula 4.29) and the simulated one. The source code of the simulator is shown in Appendix F.

The function *main*, Figure F2, is a driver that calls HiTest and varies the parameters of HiTest architecture, the program to be tested, and the length of the experiment. When run, *main* requires the command line to specify the number of basic blocks (K) of the program to be tested, the number of hosts (N) and PEs (M) of the architecture, and the time limit (t) we assign for the given experiment. If the simulator is used improperly *main* shows on the screen how to use it properly.

The function *HiTest*, Figure F3, represents hardware simulation. In other words, it simulates ordinary hosts, PEs, and the switches between them in terms of timing. More precisely it simulates time intervals needed by every hardware element to perform its functions and the waiting time when the element waits until certain conditions halt, allowing it to proceed.

The function *load* prepares the charts for the ordinary hosts that determine which basic blocks and in which order each host executes. The chart of each host is represented as a circular linked list of all basic blocks that are executed by this host in the order they appear on the linked list.

The function *program* is called by *HiTest* every time a PE finishes the execution of the basic block. The *program* returns to *HiTest* the number of the basic block to execute next.

There were four different *load* and *program* pairs produced corresponding to four different types of experiments conducted. Each of them is further discussed.

We make a few assumptions about the organization of the system that we simulate.

1. For realistic programs the number of basic blocks K is supposed to be quite large. That is why we did a simulation only for the case when $K \geq N$; that is, every basic block is executed by exactly one host, as shown in 4.3.2 and 4.3.3. In other words, every host can execute several basic blocks, but a basic block is executed by only one host.

2. Unlike the theoretical analysis, we do not neglect the switching time, i.e., the time between the moment when both -- the host and the PE -- are ready-to-start and the moment when the transmission of instruction begins. This switching time is chosen to be one unit of time in our simulator.

3. As soon as a PE terminates the execution of one data case or mutant it starts the next one: basic block #1 always follows the last basic block. In other words we assume that the first and the last basic block include some additional work that is not a part of the program to be tested. For the first block this additional work is loading the initial content of the PE's local memory making it possible for the program (mutant) to run. For the last block this work includes comparing the outputs of the original program and the mutant, making the decision whether the mutant is killed or not. It should be noted that this additional work would be performed by any mutation system and, therefore, must not be considered as overhead.

We conducted four types of experiments.

4.4.1. First Set of Experiments.

In the first experiment we used a simple but unrealistic model of computation: after each basic block, except for the last one, the next block is chosen randomly among all basic blocks. Finishing the last basic block is considered as halting. The flow-graph of the program corresponding to this model is a complete graph with equal probabilities of taking all edges except the node corresponding to the last basic block, which does not have any outcoming edges.

The first experiment was a simulation of the theoretical "Simple Case" described in 4.2.1. That is $K = N$, the number of the basic blocks in the program equals the number of the hosts, and all basic blocks have the same execution time. We conducted this type of experiment to make sure that the simulator works correctly. The exact value of the utilization factor can be easily predicted for any chosen length of the execution of the basic blocks. We chose this time interval to be five units of time for our experiment.

Since the number of hosts and basic blocks is the same, *load*, Figure F4, just assigns each basic block to its host. The next basic block for each PE is chosen by the following algorithm:

```
IF current basic block is not the last one in the pro-
gram THEN
     the next block is taken randomly with equal pro-
     babilities;
ELSE (the program is terminated)
          basic block #1 is taken.
```

The execution path of the program obviously does not affect the utilization factor in this experiment, but to conduct the simulation we ignore this fact.

The utilization factor of this system is supposed to be not 1, as it was predicted theoretically in 4.2.1, but 5/(5+1) = .83333, where the numerator is the 5 units of time needed for the execution of one basic block and the denominator (5+1) includes 1 unit of switching time, as explained above.

The results of this experiment agree with the theory. They are presented in Table 4.1. The experiments showed that the utilization factor does not depend on either the number of hosts N (or the number of basic blocks K, which is the same as N) or the number of the PEs M. It only depends on the time chosen for the experiment. If the time (t) is chosen as a multiple of 6 the theoretically calculated utilization factor and the simulated one are exactly .83333.

Time assigned	Utilization factor simulated
5	1.00000
6	0.83333
100	0.84000
1000	0.83400
10000	0.83340
30000	0.83333

Table 4.1. Simple Case Simulation.

4.4.2. Second Set of Experiments.

In the second type of experiments we used the same function *program*, Figure F6, that was used in the first experiment. Therefore, the algorithm to choose the next basic block to be executed by a PE is the same also.

The difference between the first and the second experiment is in function *load*. The main task of the *load* function, Figure F5, was to assign the blocks to the hosts in the most optimal way. For the rest of the experiments we used the same *load* function. It assumes $K \geq N$ and the lengths of the basic blocks are chosen randomly from 10 to 60 units of time. Our theoretical calculations prove that one kind of optimal distribution of the basic blocks among the hosts is the distribution when each host spends approximately the same time to execute all of its basic blocks. (See Section 4.3.3.)

To avoid overly complicated calculations we used the following simple algorithm to achieve the above-mentioned optimal distribution:

```
REPEAT
        Schedule the unscheduled basic block of maximum
        execution time to the host whose sum of execution
        times of the basic blocks assigned to it is least
UNTIL all basic blocks are scheduled.
```

This experiment showed much better consistency with the theory, Section 4.3.3, than was expected. The experiments showed that the results do not depend on the number M of PEs. They are affected if the chosen time t is too small, but default time is used so that the results become independent of t. The difference between the theoretically

expected utilization factor and the simulated one very often appears only in the third significant figure. The simulated utilization factor is often higher than the theoretically predicted one (the theoretical estimation is supposed to be pessimistic, viz. formula 4.29), in spite of the fact that in the theoretical calculations we neglected the switching time.

The results are shown in Table 4.2. of Appendix F, where K represents the number of basic blocks for each part of the experiment. The first column represents the number of hosts among which the basic blocks were distributed. The second column shows the theoretical utilization factor, calculated using Formula 4.29. The third column shows the utilization factor computed by the simulator.

4.4.3. The Third Set of Experiments.

The third type of experiment involved the use of flow-graphs of real programs as an input for our simulator. All the flow-graphs we used during our experiment are shown in Appendix F, Figures F9 - F22. They are presented as adjacency lists where the value "0" indicates the end of the list.

In this type of experiment we first chose randomly the probabilities of taking branches of the flow-graph. When a PE finishes the execution of a basic block, the next basic block is chosen randomly again but taking into account the initial probabilities of taking branches. It should be noted that it is a quite realistic, commonly used model of program execution. It is used, for example, in [65]. The function *program* for this experiment is shown in Appendix F, Figure F7. The comparison of the theoretical and the simulated utilization factors is presented in Table 4.3. of Appendix F. Because of the great volume we placed all of the results of the simulation, Tables 4.2., 4.3., 4.4., and

4.5., in Appendix F.

On the top of each table of Table 4.3 we show the name of the program and the number of basic blocks used for the experiment. Each table has three columns. The first column shows the number of hosts among which the basic blocks are distributed. The second column shows the theoretical, calculated utilization factor. The formula for this calculation is $\mu = 2N / (2N + K)$, Formula 4.29. The third column shows the utilization factor received by our simulator.

4.4.4. The Fourth Set of Experiments.

The fourth type of experiment also involved the use of real program's flow-graphs. These flow-graphs were the same as the ones for the third experiment. But this time the probabilities of taking branches were part of the input taken from the flow-graph file. The function *program* is shown in Appendix F, Figure F8.

In this experiment we considered the effect of mutation on the utilization factor of the system. Since our goal is only measuring the timing of the execution, the values of the variables of the mutants are not of any importance for our work. The only thing that really matters is the behavior of the mutant that controls the flow of instructions. In other words how the mutant affects the probabilities of taking branches of the flow-graph compared to the probabilities of taking branches of the original program. Because we do not know how these probabilities differ, after how many blocks they can change, or how many forks can be affected, we performed the fourth type of experiment. Within this fourth type of experiment we produced two kinds of changes imitating mutation.

4.4.4.1. Change of the Probabilities in One Pair of Branches of the Flow-graph.

We prepared the input flow-graph file by assigning the probabilities of taking each branch equally for all branches except for one pair. This one pair of branches had changing probabilities: 0 / 1, 0.25 / 0.75, 0.5 / 0.5, 0.75 / 0.25, and 1 / 0. We repeated this experiment with each fork of the flow-graph to imitate the work of mutation. The results are presented in Table 4.4. of Appendix F.

Each table represents the results of the experiment after changing the probabilities all five times in one fork of the flow-graph. The number of the basic block in which the probabilities vary is shown on top of each table. The probabilities of taking branches was allowed to be changed only in that node.

In the first column of Table 4.4, N is the number of ordinary hosts. The second column, "calc", represents the theoretical utilization factor calculated by using the formula 4.29. The third column, "avg", shows the simulated utilization factor averaged over all five experiments. (It is changing the probabilities for one pair of branches). The fourth column of the table, "diff", is calculated using the following formula: `(avg - calc) / calc * 100%`. The fifth column, "min", and the last column, "max", are calculated using the formula: `(extrm - avg) / avg * 100%`, where "extrm" is respectively either minimum or maximum value of the utilization factor out of all five experiments for each given fork.

4.4.4.2. Change of the Probabilities in Two Pairs of Branches of the Flow-graph.

In the last experiment we assigned the changing probabilities to two pairs of branches of the flow-graph instead of one, and repeated the experiment with changes in

all possible pairs of forks. The results are presented in Table 4.5.

We do not know exactly how mutation would change the branch probabilities of the original program. By doing these experiments we are making an attempt to find out how changing these probabilities would affect the utilization factor. In experiment 4.4.4.1 we changed the probabilities in every fork of branches from 0 to 100%, in experiment 4.4.4.2 we changed the probabilities in each possible pair of forks. Both experiments showed very little dependence of the utilization factor on the changes, so we decided not to expand these experiments by taking three, four, or more groups of forks (these experiments would require significant run time). In Table 4.7 we present the analysis of the results shown in Table 4.4 and 4.5 of the Appendix F. The abbreviation "avg" stands for average difference, and the abbreviation "extrm" stands for both maximum and minimum differences. We calculated for how many cases the average and extreme differences between the calculated utilization factor and the simulated one are 5% or more, 10% or more, and 15% or more out of all the experiments. Tables 4.6 and 4.7 show that changing the probabilities in one fork of the flow-graph will affect the average difference of 10% or more between the calculated and simulated utilization factors only in 4% of the 318 experiments. And after changing the probabilities in two forks of the flow-graph the average difference of 10% or more is in only 5% of all the 420 experiments.

Table 4.6. Analysis of the results for the experiment 4.4.4.1.

	5% or more	10% or more	15% or more
avg	33%	4%	0%
extrm	13%	7%	3%

Table 4.7. Analysis of the results of the experiment 4.4.4.2.

	5% or more	10% or more	15% or more
avg	35%	5%	0.5%
extrm	40%	25%	12%

The similarity of the theoretical and the simulated utilization factors allows us to conclude that the theoretical utilization factor $\mu = 2N / (2N + K)$ is a good estimate for the HiTest mutation system.

4.5. Implementation Issues

Our purpose here has been to present the concepts that enable the architecture to be used for mutation analysis and software testing, not to describe all of the implementation details. Some issues deserve acknowledgment, however.

There needs to be some overall system control; we assume the existence of a central processor which would be used to compile and analyze the program, allocate basic blocks to hosts, generate mutated code, generate maps for the special hosts, and keep track of the outputs of the different mutants. These tasks are similar to those accomplished by Mothra [12], and are all feasible on our system.

This central processor is supposed to have writing and reading rights over all memory of the machine including its own local memory, the local memory of ordinary

and special hosts, and the local memory of the PEs. Prior to running of the mutants the central processor writes into the local memory of ordinary hosts the executable code of the basic blocks the given host is supposed to transmit. It also writes into the local memory of each special host an array of mutation sections corresponding to each PE that the given special host controls. It writes in the local memories of the PEs the initial data or, in other words, the state from which the corresponding mutant is supposed to start the execution, namely the state at the point where the mutant was weakly killed. It puts the number of the basic block to be executed in register B of every PE and turns all PEs' ready-to-start bits on. Then it enables ordinary hosts for execution and they execute the first instructions that are to put the number of the basic block they are about to execute in their register A and turn the ready-to-start bits on. From this point on, the main work of the central processor is just to check if the mutant is strongly killed or not when the corresponding PE terminates execution and to record this information in its own memory. After this, it puts the initial data of the next mutant into the local memory of the PE and changes the corresponding entry of an array of the mutated sections of the special host that controls this PE. Then it enables the PE for execution by turning its ready-to-start bit on.

Ordinary and special hosts are not supposed to do any writing into memory. They just read appropriate instructions from their local memory and transmit them to PEs. The only writing the ordinary hosts do is entering the basic block number in their register A and turning their ready-to-start bit on and off. The PE's local memory contains data only and it operates on this data instructions that the host currently controlling the PE transmits.

There also needs to be some input/output system for PEs. We assume that every input/output device has a server that can work as a special host. Every input/output instruction is considered as a separate basic block. When a PE needs input or output it sends a signal to the server of the corresponding input/output device. When the server is ready, it starts controlling the PE the same way the special host does, executing input/output instruction and then transmitting to the PE an instruction to enter the next basic block to be executed in its register and to turn its ready-to-start bit on, returning the PE to normal parallel execution mode. It should be noted that when doing mutation analysis the PE is not expected to execute real input/output instructions. An input corresponding to the data case and the output of the original program corresponding to the same data case should be stored in the PE's local memory. Then instead of performing actual input, the host transmits to the PE instructions just to copy the values from its local memory. And instead of performing actual output it just transmits instructions to check if the output is the same as that of the original program. If the output differs, the PE terminates the execution and sends a signal to the central processor. The central processor then marks the mutant as dead and follows the procedure discussed above.

5. Conclusions

5.1. Summary

We have presented new serial algorithms for mutation analysis that runs faster than existing ones under fairly general conditions. The speed-up possible with our algorithms depends upon the nature of both the program and the test set. It can be quite large, even proportional to the size of the program. In any case, our strong mutation algorithm runs no worse than the existing naive strong mutation algorithm. Our weak mutation algorithm performs at least as fast as the fastest known weak mutation algorithm, and under very realistic assumptions, it can be much faster.

We recognize that the data structures required by these algorithms can consume a large amount of space. For this reason we have proposed two different and complementary methods of reducing the space requirements of the algorithms without a significant decrease in their speed. In one method, the amount of space consumed by each program trace is reduced by changing the granularity of the representation, i.e., by representing a trace as a sequence of the program's states at entry to basic blocks rather than program statements. In the other method, the trace is made smaller by reducing the amount of space needed in each saved state. We believe that these methods can be combined to make this approach feasible.

If mutant creation is incremental, in that mutants of a program are generated in batches rather than all at once, then the algorithms we described here perform equally well; the total running time is the same assuming all else is equal, and the above performance analyses will be the same in each case. Inspection of Figure 3.5 shows that there

are two possible situations that can arise in incremental analysis: either a mutant M_j^k is added to the system before the driver has completed checking all mutants of statement I_j currently in the system, or it is added afterwards. In the first case, nothing needs to be done any differently than described above, and it is clear that the performance is identical to an all-at-once system. In the second case, the driver needs to be rerun after it terminates, since there are mutants that were never analyzed. The reader can verify though, that no work is repeated when the driver is rerun, because only unanalyzed mutants are analyzed, and all of the necessary data structures were created the first time around. Thus, the performance under incremental creation is the same as that under an all-at-once system.

Sometimes during test data evaluation, new test data is added to the test set to improve its quality. Our mutation analysis algorithms are designed to work equally well in this situation also. In this case, it is sufficient to rerun the driver routine with the new test data. Of course, we first need to create the MDS for each test, but otherwise there is no additional work. The point is that whether the test data arrives incrementally or is presented to the system all at once, the amount of work remains the same.

We have also presented the design of a dynamically reconfigurable, multiple SIMD architecture that is well suited for reducing the overall cost of mutation analysis as well as more general testing of large scale software. Our analysis shows that, if the system parameters are tuned properly for the kind of software to be expected, then optimal speed-ups can be obtained, and the utilization of the hardware and consequently the speed-up is much greater than is possible with ordinary SIMD machines. We conducted software simulations of the architecture and verified that our predicted performance

measures are accurate.

We believe that using the proposed algorithms together with a system such as HiTest would be a valuable asset for meeting high reliability requirements for large software systems.

5.2. Future Work

Our algorithms may be parallelized fairly easily. We suspect that parallel versions of these algorithms will obtain greater speed-up than current parallel mutation algorithms. This is one area of future research that we intend to explore. Another important direction is to implement a mutation analysis system based upon these algorithms. We are currently considering the issues pertaining to such an implementation. The important area of future work is to conduct experiments to get empirical evidence about the running time of these algorithms.

Other areas of research we plan include elaborating the details of HiTest (detailed design of all elements such as hosts, PEs, and the switch between them); writing the algorithms and software for the central processor that will be used to compile and analyze the program, allocate basic blocks to hosts, generate mutated code, generate maps for the special hosts, and keep track of the output of PEs (these tasks are similar to those accomplished by Mothra [4], and are all feasible on our system); and, finally, building of the actual hardware prototype.

In summary, we believe that our work can provide a foundation for more efficient mutation analysis systems, and hence, a basis for improving the quality of software in general.

Appendix A

Theorem 2. *For any real* P_{goal}, $0 < P_{goal} \leq 1$, *and any test set* T,

$$\sum_{k=1}^{|T|-1} kP_{goal}(1-P_{goal})^{k-1} + |T|(1-P_{goal})^{|T|-1} = \frac{1-(1-P_{goal})^{|T|}}{P_{goal}} \tag{A1}$$

Proof: The key to the proof is the following lemma:

Lemma 1. *For any number* $x < 1$ *and integer* $N > 0$.

$$\sum_{j=0}^{N-1}(j+1)x^j = \frac{1}{1-x}\left(\frac{1-x^N}{1-x} - Nx^N\right)$$

Proof: We prove this by induction on N. For $N = 1$, the left hand side is 1 and the right hand side is

$$\frac{1}{1-x}\left(\frac{1-x}{1-x} - x\right) = \frac{1}{1-x}(1-x) = 1$$

so it holds for $N = 1$. We assume it is true for $N = K - 1$ and show it is true for $N = K$.

$$\sum_{j=0}^{K}(j+1)x^j = \sum_{j=0}^{K-1}(j+1)x^j + (K+1)x^K \tag{A2}$$

Using the inductive hypothesis,

$$\sum_{j=0}^{K}(j+1)x^j = \frac{1}{1-x}\left(\frac{1-x^K}{1-x} - Kx^K\right) + (K+1)x^K$$

$$= \frac{1}{1-x}\left(\frac{1-x^K}{1-x} - Kx^K + (1-x)(K+1)x^K\right)$$

$$= \frac{1}{1-x}\left(\frac{1-x^K}{1-x} - Kx^K + Kx^K + x^K - Kx^{K+1} - x^{K+1}\right)$$

$$= \frac{1}{1-x}\left(\frac{1-x^K}{1-x} + x^K - (K+1)x^{K+1}\right)$$

$$= \frac{1}{1-x}\left(\frac{1-x^K + (1-x)x^K}{1-x} - (K+1)x^{K+1}\right)$$

$$= \frac{1}{1-x}\left(\frac{1-x^{K+1}}{1-x} - (K+1)x^{K+1}\right)$$

which proves that the hypothesis holds for $N=K$. By the axiom of induction, the lemma is proved.

The next step is to observe that if we replace $1 - P_{goal}$ by x in Equation A1 we obtain the following:

$$(1-x) + 2(1-x)x + 3(1-x)x^2 + \cdots + (|T|-1)(1-x)x^{|T|-2} + |T|x^{|T|-1}$$
$$= (1-x)(1 + 2x + 3x^2 + \cdots + (|T|-1)x^{|T|-2}) + |T|x^{|T|-1}$$

and to observe that we can apply the lemma to the expression in parentheses, since $1 - P_{goal} > 0$, yielding

$$(1-x)\frac{1}{1-x}\left(\frac{1-x^{|T|-1}}{1-x} - (|T|-1)x^{(|T|-1)}\right) + |T|(x)^{|T|-1}$$

$$= \frac{1-x^{|T|-1}}{1-x} - x^{(|T|-1)}$$

$$= \frac{1-x^{|T|-1} - (1-x)x^{(|T|-1)}}{1-x}$$

$$= \frac{1-x^{|T|}}{1-x}$$

Substituting x by $1 - P_{goal}$ we get

$$\frac{1-(1-P_{goal})^{|T|}}{P_{goal}} \tag{A3}$$

proving the theorem.

Appendix B

To determine whether or not a given test weakly kills a mutant, we traverse the linked list attached to the mutated statement of the MDS, checking whether the state of mutant and program differ after each execution of the statement. The worst case occurs when the entire list is traversed, corresponding to the case in which the very last execution of the mutated statement is the one that kills the mutant. The best case is when the very first execution of the mutated statement kills the mutant. It is this case whose time complexity we analyze here. We show that

$$\frac{r(P,T)}{l(P)}(1-P_{weak})(1-\varepsilon_{weak})$$

is an approximate greatest lower bound on the complexity of using our weak mutation algorithm to determine whether a mutant is weakly killed by the test set. We say it is approximate because we use a simplifying assumption to derive this formula, namely that zero steps are needed to reach the first node of the linked list, rather than a single time unit. The consequence is that the formula that we derive is actually not an attainable greatest lower bound but a close approximation to a greatest lower bound.

Because we assume that zero time is used by the algorithm to check the test that actually weakly kills the mutant, the cost associated with discovering that the kth test tried is the one that weakly kills the mutant is the cost of checking the first $k-1$ tests only; the kth test is considered ''free''. The probability that exactly k tests are needed to discover that the mutant is weakly killed is $P_{weak}(1-P_{weak})^{k-1}$. Thus the average cost of this best case scenario is

$$\frac{r(P,T)}{l(P)}\left(\sum_{k=2}^{|T|}(k-1)P_{weak}(1-P_{weak})^{k-1} + |T|(1-P_{weak})^{|T|}\right)$$

$$=\frac{r(P,T)}{l(P)}\left(\sum_{k=1}^{|T|-1}kP_{weak}(1-P_{weak})^{k} + |T|(1-P_{weak})^{|T|}\right) \qquad \text{(B1)}$$

The factor $r(P,T) / l(P)$ is the cost of traversing a single linked list, on average. The first $|T|-1$ terms of the sum are the contributions of the cases in which the mutant is weakly killed by the second, third, fourth, up to the $|T|th$ tests respectively. (If the mutant is weakly killed by the first test case tried, the contribution is zero.) The term $|T|(1-P_{weak})^{|T|}$ is the contribution of the case in which none of the tests in the test set weakly kill the mutant. In this case all $|T|$ linked lists must be traversed. This occurs with probability $(1-P_{weak})^{|T|}$ and since all $|T|$ linked lists have to be traversed completely, its cost is $|T|$ traversals.

The reader can observe that expression B1 may be rewritten as

$$\frac{r(P,T)}{l(P)}(1-P_{weak})\left(\sum_{k=1}^{|T|-1}kP_{weak}(1-P_{weak})^{k-1} + |T|(1-P_{weak})^{|T|-1}\right)$$

which, by Theorem 2, is

$$\frac{r(P,T)}{l(P)}(1-P_{weak})\frac{1-(1-P_{weak})^{|T|}}{P_{weak}}$$

Using the definition of ε_{weak} this becomes

$$\frac{r(P,T)}{l(P)}(1-P_{weak})(1-\varepsilon_{weak})$$

which is precisely what we wanted to prove.

Appendix C

A particular nonlinear programming problem arises often in the text that we state it in generality here and describe its solution. The problem is to minimize the function

$$F(x_1, x_2, \cdots x_n) = \sum_{i=1}^{n} f(x_i) \tag{C1}$$

subject to the constraint

$$K - \sum_{i=1}^{n} x_i = 0$$

Where $f(x)$ is concave and $x_i \geq 0$ There are two obvious ways to solve this problem:

- Using Lagrange's function

$$R = \sum_{i=1}^{n} f(x_i) - \lambda(K - \sum_{i=1}^{n} x_i)$$

and setting $\frac{\partial R}{\partial x_i} = 0$ from which we obtain

$$\frac{df(x_i)}{dx_i} - \lambda = 0$$

implying that

$$\frac{df(x_i)}{dx_i} = \lambda = const \quad independent \quad of \quad i. \tag{C2}$$

or

- Using substitution. In this case we let

$$x_n = g(x_1, x_2, \cdots x_{n-1}) = K - \sum_{i=1}^{n-1} x(x_i) \tag{C3}$$

The condition $\frac{dF}{dx_i} = 0$ can be rewritten as

$$\frac{dF}{dx_i} = \frac{\partial F}{\partial x_i} + \frac{\partial F}{\partial x_n}\frac{\partial g}{\partial x_i} = 0$$

From (C1)

$$\frac{\partial F}{\partial x_i} = \frac{df(x_i)}{dx_i}$$

and from (C3)

$$\frac{\partial g}{\partial x_i} = -1$$

for $i = 1,2, \cdots n-1$

so that

$$\frac{df(x_i)}{dx_i} = \frac{df(x_n)}{dx_n} = const \quad independent \quad of \quad i.$$

This is concave minimum problem with convex constrains. If it has a stability point inside the constrained space, then this point has to be a global minimum.

Appendix D

Some Comments on the Matter of Approximation

In this appendix we show that the expression

$$\sum_{i=1}^{K}(\sqrt{p_i}\sum_{j=1}^{K}\sqrt{p_j}) = (\sum_{i=1}^{K}p_i)^2$$

may be reasonably approximated by K. The first step in deriving this approximation is to observe that the following equalities are true:

$$\begin{aligned}\sum_{i=1}^{K}(\sqrt{p_i}\sum_{j=1}^{K}\sqrt{p_j}) &= \sum_{i=1}^{K}(\sqrt{p_i}(K\sqrt{p_i} + \sum_{j=1}^{K}(\sqrt{p_j} - \sqrt{p_i}))) \\ &= K\sum_{i=1}^{K}p_i + \sum_{i=1}^{K}\sqrt{p_i}\sum_{j=1}^{K}(\sqrt{p_j} - \sqrt{p_i}) \\ &= K + \sum_{i=1}^{K}\sqrt{p_i}\sum_{j=1}^{K}(\sqrt{p_j} - \sqrt{p_i}) \\ &= K + \Delta\end{aligned}$$

where Δ is defined as $\Delta = \sum_{i=1}^{K}\sqrt{p_i}\sum_{j=1}^{K}(\sqrt{p_j} - \sqrt{p_i})$

Using the identity

$$\sqrt{p_j}(\sqrt{p_i} - \sqrt{p_j}) + \sqrt{p_i}(\sqrt{p_j} - \sqrt{p_i}) = -(\sqrt{p_i} - \sqrt{p_j})^2$$

we have

$$\begin{aligned}\Delta &= -\sum_{i=1}^{K}\sum_{j=1}^{i}(\sqrt{p_i} - \sqrt{p_j})^2 \\ &= -\frac{1}{2}\sum_{i=1}^{K}\sum_{j=i+1}^{K}(\sqrt{p_i} - \sqrt{p_j})^2\end{aligned}$$

from which we observe that $\Delta \leq 0$. This shows that K is an upper, or more pessimistic,

bound for the overhead (which is how we used this approximation).

Let E be the expected value of $\sqrt{p_i}$, i.e.,

$$E = \frac{1}{K}\sum_{i=1}^{K}\sqrt{p_i}$$

Let $\Delta_i = \sqrt{p_i} - E$. Then, the depression of $\sqrt{p_i}$ is

$$\sigma = \frac{1}{K}\sum_{i=1}^{K}\Delta_i^2$$

Then

$$\begin{aligned}
\Delta &= -\frac{1}{2}\sum_{i=1}^{K}\sum_{j=1}^{K}(\sqrt{p_i} - \sqrt{p_j})^2 \\
&= -\frac{1}{2}\sum_{i=1}^{K}\sum_{j=1}^{K}(\Delta_i - \Delta_j)^2 \\
&= -\frac{1}{2}(\sum_{i=1}^{K}\sum_{j=1}^{K}(\Delta_i^2 - 2\Delta_i\Delta_j + \Delta_j^2) \\
&= -\frac{1}{2}(K\sum_{i=1}^{K}\Delta_i^2 + K\sum_{j=1}^{K}\Delta_j^2) + \sum_{i=1}^{K}\Delta_i\sum_{j=1}^{K}\Delta_j
\end{aligned}$$

But because $\sum_{i=1}^{K}\Delta_j = 0$,

$$\Delta = -K\sum_{i=1}^{K}\Delta_i^2 = -K^2\sigma$$

Appendix E

Calculations of the Utilization Factor for Near-Optimal Hosts-Block Mappings

There are two cases to consider:

Case 1. $K \ll N_O$ If n_i is a constant independent of i, then each block is assigned the same number of hosts, independent of the probability of its execution.

$$n_i = \frac{N}{K} \tag{D1}$$

Substituting (D1) in (4.11) we have

$$\tau = \frac{\mu}{2} \sum_{i=1}^{K} \frac{p_i}{n_i} = \frac{\mu}{2} \frac{K}{N} \sum p_i = \frac{\mu}{2} \frac{K}{N}$$

implying that

$$\mu = \frac{2N}{2N + K}$$

Suppose instead that n_i is proportional to p_i. Let us define $P = \frac{1}{N}$. Then

$$n_i = \frac{p_i}{P}$$

$$P = \frac{p_i}{n_i}$$

and

$$\tau = \frac{\mu}{2} \sum_{i=1}^{K} \frac{p_i}{n_i} = \frac{\mu}{2} \sum_{1}^{K} p = \frac{\mu}{2} \frac{K}{N}$$

or

$$\mu = \frac{2N}{2N + K}$$

Thus, whether we distribute the hosts without any dependence on the blocks' probabilities or we distribute them in proportion to the distribution of these probabilities we receive the same result. This means that the optimal distribution lies somewhere in between. Actually, in (4.8) we showed that n_i is proportional to $\sqrt{p_i}$ which corresponds with what we obtained here.

Case 2. $K \gg N_O$.

Consider the following distribution: each host gets the blocks whose execution times are the same. That is $T_{i_1} = T_{i_2} = T_{i_j} = T_i$. The number of these blocks is such that the total probability of their execution is the same for all the hosts. Thus, $\sum_{j=1}^{K} p_{i_j} \approx const = \frac{1}{N}$. From (4.11) we have

$$\tau = \frac{\mu}{2} \sum_{i=1}^{N} (\Theta_i \sum_{j=1}^{K_i} \frac{p_{i_j}}{T_{i_j}})$$

$$= \frac{\mu}{2} \sum_{i=1}^{N} \frac{\Theta_i}{T_i} \sum_{j=1}^{K_i} p_{i_j}$$

$$= \frac{\mu}{2} \frac{1}{N} \sum_{i=1}^{N} K_i$$

$$= \frac{\mu}{2} \frac{K}{N}$$

implying that

$$\mu = \frac{2N}{2N + K}$$

The last case to consider is that, without any dependence on the probabilities of the blocks' executions, the blocks are assigned to hosts in such a way that the sum of the execution times of all of the blocks assigned to the host is constant. That is $\Theta_1 = \Theta_2 = \cdots = \Theta$. Then using (4.11) we have

$$\begin{aligned}
\tau &= \frac{\mu}{2}\sum_{i=1}^{N}\Theta\sum_{j=1}^{K_i}\frac{p_{i_j}}{T_{i_j}} \\
&= \frac{\mu}{2}\Theta\sum_{\xi=1}^{K}\frac{p_\xi}{T_\xi} \\
&= \frac{\mu}{2}\frac{T}{N}KE\left(\frac{P_\xi}{T_\xi}\right) \\
&\approx \frac{\mu}{2}\frac{T}{N}\frac{E(P_\xi)}{E(T_\xi)} \\
&= \frac{\mu}{2}\frac{K}{N}
\end{aligned}$$

implying that

$$\mu = \frac{2N}{2N+K}$$

Thus, again we see that each different assignment of blocks to hosts results in the same utilization factor.

Appendix F

Source Code of the Simulator.

```
#define NULL    0

#define MAX_N   1000
#define MAX_M   1000000
#define MAX_K   10000

int N, M, K;                            /*  numbers of hosts, PEs and blocks */

int btime[MAX_K];                       /*  execution time of each block     */

struct list {                           /*  table of sets of PEs             */
               int num;                 /*  waiting to execute block i,      */
               struct list *next;       /*  for i = 0 to K - 1               */
            } *(waitlist[MAX_K]);

struct h_struc {                        /*  schedule of blocks assigned to   */
               int time;                /*  host j, for j = 0 to N - 1       */
               struct list *block;      /*  each host's blocks               */
              } host[MAX_N];            /*  are executed cyclically          */

struct pestruc {
               int time;                /*  remaining time to complete block */
               int block;               /*  NO of block currently executing  */
              } PE[MAX_M];              /*  for each PE                      */

long treal, tuseful, twait;

long tlimit;                            /* the lenght of the symulation      */

unsigned short xsubi[3];                /*  space for random number generator */
```

Figure F1. Common Definitions.

```
#include "def.h"
#include <stdio.h>

static char *program; /*pointer points to a string which is a filename*/
static char usage[]=
          "Usage: %s -K numblocks -N numhosts [-M numPEs] [-t timelimit]\n";
/*message prints out the format of command line */
/*The main procedure reads in the values of number of blocks, number of hosts*/
/*, number of PEs and the time limit which are given by the user.  It also   */
/*validates the values.  If any of the values is invalid, it will branch to  */
/*another procedure to print out an error message and the program will be    */
/*terminated.  Otherwise, it will simulate the parallel machine to test out  */
/*the performance of this architecture which works on mutated programs.      */

main(argc, argv) /* just the driver that insure correct calling of HiTest*/

int argc;
char *argv[];
{
    char c;
    extern char *optarg;

    program = argv[0];   /*gets filename*/

    K = 0;              /*initial the value of # of blocks*/
    N = 0;              /*initial the value of # of hosts*/
    tlimit = 77777;     /*initial the time limit if the user did not specify in*/
                        /*the input.                                           */
    M = 100;            /*initial the value of # of PEs if the user did not    */
                        /*specify in the input.                                */

    while ((c = getopt(argc, argv, "K:N:M:t:b")) != EOF) /*reads in the input*/
      switch(c) {
        case 'K':  /* specify the number of blocks in the program */
                   K = atoi(optarg);
                   if(K <= 0) /*make sure the value of # of blocks is positive*/
                       badval(c, K);  /*If not, print out an error message
                                                                  and halt*/
                   break;
        case 'N':  /* specify the number of hosts */
                   N = atoi(optarg);
                   if(N <= 0) /*make sure the value of # of hosts is positive*/
                       badval(c, N);  /*If not, print out an error message
                                                                  and halt*/
                   break;
        case 'M':  /* specify the number of PEs */
                   M = atoi(optarg);
                   if(M <= 0) /*make sure the value of # of PEs is positive*/
                       badval(c, M); /*If not, print out an error message
                                                                  and halt*/
                   break;
        case 't':  /* specify the time limit of the simulation */
                   tlimit = atoi(optarg);
                   if(tlimit <= 0) /*validates the value of time limit*/
                       badval(c, tlimit); /*If not, print out an error message
                                                                  and halt*/
                   break;
        default : dousage();   /*echo the input*/
```

Figure F2. Function Main.

```
        } /* switch */
      if ((K == 0) || (N == 0)) dousage();
      if (K < N) badcase(K, N); /*ensure number of blocks is not smaller than */
                                /*number of hosts.  Otherwise call an error   */
                                /*handler to print out a message.             */

      HiTest();       /*simulates the parallel system */
  } /* main */


/*procedure to show the user the format of command line */
dousage()
{
    fprintf(stderr, usage, program);  /*prints out the format of command line*/
    exit(1);                          /*and terminates the program              *
} /* dosage */

/*Argument: n - the value which is given by the user which corresponds to the   */
/*              variable c.                                                     */
/*          c - variable name which could be K, N, M or tlimit which depends    */
/*              on which variable has a bad value assigned to it.               */
/*procedure prints out an error message which tells the bad value, c corresponds*/
/*to the variable n which c is given by the user and termintes the program      */
badval(c, n)
{
    fprintf(stderr, "%s: bad value - %c = %d\n", program, c, n);
    exit(2);
} /* badval */

/*Arguement: k - the value of number of blocks which is inputed by the user.  */
/*           n - the value of number of hosts which is inputed by the user.   */
/*procedure prints out an error message and terminates the program if the value */
/*of k is smaller than n which means # of blocks is smaller than # of hosts     */
badcase(k, n)
{
    fprintf(stderr, "%s: bad values - K < N (%d < %d)\n", program, k, n);
    exit(3);
} /* badcase */
```

Figure F2. Continued.

```
#include "def.h"

int load(), program();

/*Procedure simulates the operations of hosts and PEs. It will keep on
  simulating the executions of blocks while the elapse time is within the time
  limit set by the user.  If any PE reaches the end of the program before
  tlimit, it will start a new case and go back to the first block and start
  executing the program again.  When the elapse time is equal to the tlimit,
  the simulation stops and call a procedure to give out the results.          */

HiTest()      /* the simulator */
{
    load();  /* initial all essential variables in order for HiTest to run*/
    treal = tuseful = twait = 0;  /*initial all time variables*/
    while(treal < tlimit) /*keep simulating while elapse time is within tlimit*/
    {
    int i;     /*index variable*/

        treal++;    /*increment the elapse time*/
        for(i=0; i<N; i++)          /* for each host */
            transmit(&(host[i]));  /*simulate the operation of hosts*/
        for(i=0; i<M; i++)          /* for each PE */
            compute(&(PE[i]), i);  /*simulate the operation of PEs*/
    }
    bookkeep();      /*print out the result of the performance*/
} /* HiTest */

/*Argument: h - is a pointer points to a structure which contains two fields. */
/*               The first field contains the remaining execution time of a    */
/*               particular block i which is currently executed by the host.   */
/*               The second field contains the pointer which points to a cyclic*/
/*               list of basic blocks assigned to the host.                    */
/*Procedure simulates the operation of a host.  First, it checks to see whether
  the block which is currently executing is finished by its PEs.  If it is, it
  will schedule an another block to be executed next which is the next block on
  the host's cyclic list.  Finally, it frees up all the PEs which have just
  finished block i.                */

transmit (h)
struct h_struc *h;
{
    if (h->time-- == 0) /* a block was just finished */
    {
    struct list *l;
        h->block = h->block->next; /*schedule a block j off the list to be
                                     transmited next*/
        h->time = btime[h->block->num]; /*gets the excution time of block */
        l = waitlist[h->block->num];
        while(l != NULL)        /*free up all PEs on the waitlist which are */
        {                       /*waiting for block j.                      */
        struct list *f;
            f = l;
            PE[l->num].time = h->time;
            l = l->next;
            free(f);
        }
        waitlist[h->block->num] = NULL;
    }
} /* transmit */
```

Figure F3. Function HiTest.

```
/*Argument: p - a pointer points to a structure which contains the remaining */
/*              time to complete a block i and the id. of block i is current-*/
/*              ly executed by a PE.                                          */
/*          e - the id number of a PE                                         */
/*Procedure simulates the operation of a PE.  If the PE is currently execut- */
/*ing a block, the busy time will be incremented.  If the PE has just         */
/*finished executing a block, it will schedule the next block to be executed */
/*by calling the function Program.  And the PE will be put on the waitlist    */
/*which associates to the new block.  If the PE is still on the waitlist and */
/*hasn't started to execute the new block, the idle time will be incremented */

compute (p,e)
struct pestruc *p;
int e;
{
    if(p->time > 0) /*PE is busy*/
        tuseful++;
    if(p->time-- ==0) /* time to switch to another block */
    {
    struct list *l, *lalloc();
        p->block = program(p->block, e); /*schedule a new block */
        l = lalloc();
        l->num = e;
        l->next = waitlist[p->block];  /*put PE on the new block's waitlist*/
        waitlist[p->block] = l;
    }
    if(p->time < 0)  /* PE is idling */
        twait++;     /*increment the idle time*/
} /* compute */

/*Procedure prints out the theoretical and the calculated UF */

bookkeep()
{
    printf("%4d ", N);    /*number of hosts*/
    printf("%6.5f ", (float)(2 * N) / (float)(2 * N + K));/*theoretical result*/
    printf("%6.5f\n",
           (float)tuseful/(float)(tuseful + twait)); /*simulated result*/
} /* bookkeep */

/*Function allocates enough space for the structure which has two fields.  The */
/*first field contains the id no. of a PE and the second field is a pointer    */
/*points to its own structure.  And the function returns the pointer points to */
/*that structure.                                                              */
struct list *lalloc()
{
char *malloc();

    return((struct list *) malloc(sizeof(struct list)));
} /* lalloc */
```

Figure F3. Continued.

```
#include "def.h"


/*This is a simple case which number of blocks and number of hosts are equal */
/*and assuming every blocks take the same amount of execution time.          */
/*Procedure initializes all basic blocks take 5 units of elapse time to ex-  */
/*cute and the waitlists of every blocks are empty.  And for each host i, it */
/*is responsible for executing block i.  Finally, it sets all PEs start the  */
/*first block all at the same time.                                          */

load()
{
int i;

    for(i=0; i<K; i++)
    {
    struct list *l, *lalloc();

        btime[i] = 5;  /*all blocks take 5 units of elapse time to execute */
        waitlist[i] = NULL;  /*waitlist for each block is empty */
        host[i].time = 0;

        l = lalloc();
        l->num = i;      /*each host choose a block with the same id no. as */
        l->next = l;     /*itself.  Because it is in a cyclic list and only */
        host[i].block = l; /*one block in it,  therefore its field, next,  */
    }                      /*is pointed to itself.                          */
    for(i=0; i<M; i++)
    {
        PE[i].time = 5;    /*all PEs start the first block*/
        PE[i].block = 0;
    }
}

/*Arguments: blocknum - the id no. of a block                                */
/*           penum - the id no. of a PE                                      */
/*Returns : a new block id no.                                               */
/*Function takes the blocknum to check whether the program is just terminated*/
/*If it is, returns 0 and starts a new case which means the new block for the*/
/*PE to execute next is the first block.  Otherwise, it will choose a block  */
/*randomly for the PE to execute next.                                       */

int program(blocknum, penum)
int blocknum, penum;
{
long nrand48();
short int xsubi[3];
    if(blocknum == K-1)  /* program terminated */
        return(0);       /* start a new case   */
    else
        return(nrand48(xsubi) % K);  /*choose a new block randomly*/
}
```

Figure F4. Functions Load and Program for the First Kind of Experiments.

```
#include <stdio.h>
#include "def.h"

struct two_int_list {
                        int num;      /*either id no. of a host or a block */
                        int time;     /*exeution time*/
                        struct two_int_list *next;
                    } *hostlist, *blist;
struct two_int_list *sortblocklist(), *reorder(), *cut(), *tlalloc();

/*Procedure mainly gets information of a flowchart and distributes the blocks*/
/*to the hosts evenly by assigning a block with maximum execution time to a  */
/*host with minimum total execution time.  And it finds the host which is    */
/*responsible for the first block.                                           */

load()
{
int i;
    progload();       /* reads in information of a flowchart */
    blist = sortblocklist();  /* all BB listed by descending exec. time */
    hostlist = NULL;
    for(i=0; i<N; i++)           /* initializes all hosts with 0 total */
    {struct two_int_list *til;   /* execution time.                    */
        til = tlalloc();
        til->num = i;
        til->time = 0;
        til->next = hostlist;
        hostlist  = til;
    }
    preload();   /*defines the initial state of HiTest*/
    while(blist)
    {
        if(blist->num == 0)
            i = hostlist->num;
        schedule(blist, hostlist); /* schedules a block with maximum execution
                time to the host with minimum total execution time of blocks */
        blist = cut(blist); /* delete the BB from blist */
        hostlist = reorder(hostlist); /* keep it sorted */
    }
    setfirstblock(i);   /*set the system to transmit the first block first*/
}

/*Return : bl - is a pointer points to a list of records which contains id no.*/
/*             of a block and its execution time and the list is sorted by de-*/
/*scending time.                                                              */
/*Function generates execution time for each block randomly (from 10 to 60).  */
/*Then it stores the id no. of a block and its execution time in a block and  */
/*all these blocks are linked in a list and are sorted by descending execution*/
/*time.  Finally, the function returns a pointer points to this list.         */

struct two_int_list *sortblocklist()

{struct two_int_list *bl;
 int i;
    bl = NULL;            /*initial value which is empty*/
    for(i=0; i<K; i++)          /*for each block*/
    {struct two_int_list *til, *pre, *pos;
     long nrand48();
     short int xsubi[3];
        btime[i] = nrand48(xsubi) % 51 + 10;/*generate execution time randomly*/
        til = tlalloc();                    /*from 10 to 60                   */
        til->num = i;
        til->time = btime[i];
        pos = bl;
        pre = NULL;
```

Figure F5. Function Load 2 and Some Supporting Procedures.

```
        while(pos != NULL && til->time < pos->time)  /*sorting the list by de-*/
        {                                            /*scending execution time*/
            pre = pos;
            pos = pos->next;
        }
        til->next = pos;
        if(pre)
            pre->next = til;
        else
            bl = til;
    }
    return(bl);       /*return the pointer which points to a sorted list*/
}

/*Arguments: bl - pointer points to a sorted list which contains id no. of a */
/*                block and its execution time in a descending manner         */
/*           hl - pointer points to a list which contains id no. of a host and*/
/*                0 total execution time                                      */
/*Procedure schedules the block with maximum execution time to the host with  */
/*minimum total execution time of blocks.  bl points to a sorted list with de-*/
/*creasing execution time of blocks and hl points to a sorted list with in-   */
/*creasing total execution time of hosts.  Therefore, the first block on the  */
/*bl list will be assigned to the first host on the hl list.  And the block   */
/*will be put in a cyclic list in which the host would be able to tell which  */
/*block it is currently executing.                                            */

schedule(bl, hl)

struct two_int_list *bl;
struct two_int_list *hl;
{
    hl->time = hl->time + bl->time;
    {struct list *l, *lalloc();
        l = lalloc();
        l->num = bl->num;
        if(host[hl->num].block)  /*The block is put in the host's cyclic list*/
        {
            l->next = host[hl->num].block->next;
            host[hl->num].block->next = l;
        }
        else
        {
            host[hl->num].block = l;/*If the cyclic list is empty, the pointer*/
            l->next = l;            /*field will be pointed to itself.        */
        }
    }
}

/*Arguments: i - is the id no. of a host which is reponsible in executing the */
/*               first block.                                                  */
/*Procedure gets the id no. of a host which is reponsible in executing the    */
/*first block and goes through its cyclic list to transmit the first block so */
/*the program can start immediately.                                          */

setfirstblock(i)
int i;
{
    while(host[i].block->next->num)
        host[i].block = host[i].block->next;
}
```

Figure F5. Continued.

```
/*Arguments hl - pointer points to a list which contains id no. of a host and */
/*,              its total execution time of blocks.                           */
/*Return hl - pointer points to a sorted list in ascending total execution    */
/*            time of blocks.                                                 */
/*Function takes an unsorted list hl and sorts it in ascending total execu-   */
/*tion time of blocks, then it returns the sorted list which is pointed by hl */

struct two_int_list *reorder(hl)

struct two_int_list *hl;
{struct two_int_list *pre, *pos;
    pos = hl->next;
    pre = NULL;
        while(pos != NULL && hl->time > pos->time) /*sorting*/
        {
            pre = pos;
            pos = pos->next;
        }
        if(pre)
        {
            pre->next = hl;
            pre = hl->next;
            hl->next = pos;
            return(pre);
        }
        else
            return(hl);
}

/*Arguments bl - pointer points to a sorted list which contains id no.        */
/* of blocks and there execution time.                                        */
/*Returns bl - same as above.                                                 */
/*Function takes a sorted list which is pointed by bl and deallocate the      */
/* first element on the list.  Then it returns the resulting list.           */

struct two_int_list *cut(bl)
struct two_int_list *bl;
{struct two_int_list *fl;
    fl = bl->next;
    free(bl);
    return(fl);
}

/*Return: a pointer points to a structure called two_int_list                 */
/*Function allocates enough memory for the above structure and returns the    */
/*pointer points to it.                                                       */

struct two_int_list *tlalloc() /* standard */
{
char *malloc();

    return((struct two_int_list *) malloc(sizeof(struct two_int_list)));
}
```

Figure F5. Continued.

```
#include "def.h"

/* in this experiment the program is represented as a set of BB such that
            first BB # 0 is executed
            after any BB except # K-1 any BB can be executed with the same prob.
            after BB # K-1 BB # 0 is executed (a new case starts)            */

/*Procedure initializes the state of HiTest which is empty by setting all     */
/* wait lists are empty, all hosts and PEs are idle and all PEs will start    */
/* execute the first block BB # 0 at the same time.                           */

preload()
{
int i;

    for(i=0; i<K; i++)
        waitlist[i] = NULL;  /*waitlists are empty*/
    for(i=0; i<N; i++)
    {
        host[i].time = 0;
        host[i].block = NULL;   /*hosts are idle*/
    }
    for(i=0; i<M; i++)
    {
        PE[i].time = btime[0];
        PE[i].block = 0;    /*PEs will start the first block*/
    }
}

/*Arguments: blocknum - is the id no. of a particular block                  */
/*           penum - is the id no. of a particular PE                        */
/*Return: an id no. of a block                                               */
/*Function randomly generates a block for the PE to execute next.  If the PE  */
/*has currently finished the last block (program terminated), it will return 0*/
/*to the caller therefore a new case will start. Otherwise, it generates a no.*/
/*randomly which will be the next block no. which the PE will execute next    */

int program(blocknum, penum) /* randomly generates the next BB to execute     */
int blocknum, penum;
{
short int xubi[3];
    if(blocknum == K-1)  /* program terminated */
        return(0);       /* start a new case */
    else
        return(nrand48(xsubi) % K);
}

progload(){} /* we don't need it for this experiment */
```

Figure F6. Function Program 2.

```
#include <stdio.h>
#include "def.h"

/* in this experiment we take as an input a flowgraph of a real program and
   generate randomly probabilities of taking branches */

struct probgraph {
                    int num;
                    double p;
                    struct probgraph *next;
                  } *flow[MAX_K];

/*flow[k] is the kth block which points to a list which contains information */
/*of a block id no. and a number which is between 0 and 1.  It means block k */
/*can branch to any block on the list with a probability equal to the number */
/*in the field p.  Each block needs a list structure as above except the last*/
/*block.                                                                     */

/*Procedure reads in the information of a flowgraph from a file and randomly  */
/*generates probabilites of each path. All these information will be stored in*/
/*a structure which contains its path id no. and its probability.             */

progload()
{
int i, num;

    for(i=0; i < K; i++)
    {double totprob = 1.0;
     struct probgraph *pbh;
        flow[i] = NULL;      /*initialization*/
        scanf("%d", &num);
        if(num != i+1)  /*the input is formated wrong*/
            er_in(num);
        else
        {
            scanf("%d", &num); /*reads in the rest of the blocks*/
            while(num)       /* 0 indicates the end of the list of the next BB */
            {double erand48();
             short int xsubi[3];
             struct probgraph *pralloc();
                pbh = pralloc();
                pbh->num = num - 1;
                pbh->p = totprob * erand48(xsubi);  /*generates probability*/
                pbh->next = flow[i];                /*randomly*/
                totprob = totprob - pbh->p;
                flow[i] = pbh;
                scanf("%d", &num);
            }
            pbh->p = pbh->p + totprob; /*the sum of all probabilities of paths*/
        }                              /*from a block has to be equal to 1    */
    }
    scanf("%d", &num);
    if(num)
        er_in(num);  /* 0 indicates the end of input */
}

/*Return: a pointer points to a structure called probgraph                    */
/*Function allocates enough memory space for the structure, probgraph and re- */
/*turns a pointer points to it.                                               */

struct probgraph *pralloc()   /* standard */
{char *malloc();
    return((struct probgraph *) malloc(sizeof(struct probgraph)));
}
```

Figure F7. Function Program 3.

```
/*Arguments: i - an integer and also is a invalid value for the input of the  */
/*             flowgraph.                                                     */
/*Procedure prints out an error message with i as a invalid value as input    */

er_in(i)
int i;
{   printf("Erroneus input. i = \n", i);
    exit(0);
}

/*Procedure initializes the empty state of HiTest by setting all waitlists to */
/*empty and all hosts and PEs are idle and all PEs will start the first block */
/*at the same time.                                                           */

preload()
{ int i;
    for(i=0; i<K; i++)          /*all waitlists are empty*/
        waitlist[i] = NULL;
    for(i=0; i<N; i++)
    {
        host[i].time = 0;
        host[i].block = NULL; /*all hosts are idle*/
    }
    for(i=0; i<M; i++)
    {
        PE[i].time = btime[0];    /*all PEs are ready to start the first block*/
        PE[i].block = 0;
    }
}

/*Arguments: blocknum - the id no. of a particular block.                     */
/*           penum - the id no. of a particular PE                            */
/*Return: a id no. of a block                                                 */
/*Function randomly chooses the new block for the PE to execute next accord- */
/*ing to the behavior of the flowgraph.  Firstly it generates a number be-    */
/*tween 0 and 1 and then searches down the list to find the block with the    */
/*probability which is greater than or equal to the random number.  If it     */
/*finds one and it will return the id no. of that block.                      */

int program(blocknum, penum) /* chooses the next BB randomly according to
                                the flowgraph with probabilities */
int blocknum, penum;
{
    if(blocknum == K-1)  /* program terminated */
        return(0);       /* start a new case   */
    else
    {
    double rp, erand48();
    short int xsubi[3];
    struct probgraph *pbh;
        rp = erand48(xsubi);     /*generate a random number*/
        pbh = flow[blocknum];
        if(pbh)
        while(pbh)       /*searching and comparing the probabilities to the  */
        {                /*random number.                                    */
            if(rp <= pbh->p)
                return(pbh->num);
            rp = rp - pbh->p;
            pbh = pbh->next;
        }
        else
            er_in();
        printf("Warning : strange probabillity\n");
        return(flow[blocknum]->num);
    }
}
```

Figure F7. Continued.

```
#include <stdio.h>
#include "def.h"

/* in this experiment we take as an input both flowgraph of a real program
   and probabilities of taking branches */

struct probgraph {                          /* flowgraph of the program repres.*/
                    int num;                /* as an array of adjacency lists */
                    float p;                /* with probabilities of taking  */
                    struct probgraph *next;/* a given path */
                  } *flow[MAX_K];

/*Procedure loads the flowgraph of a program together with the probabilities  */
/* of taking certain branches. All these information will be stored in a list */
/*structure which contains the id no. of a block and its probability          */

progload()
{
int i, num;
    scanf("%d", &num);
    while(num)
        scanf("%d", &num);
    for(i=0; i < K; i++)       /*reads in the information of the flowgraph*/
    {
     struct probgraph *pbh;
        flow[i] = NULL;       /*initialization*/
        scanf("%d", &num);
        if(num != i+1)
            er_in();
        else
        {
            scanf("%d", &num);     /*reads in the blocks that can branch to*/
            while(num)          /* num ==0 means the end of the list */
            {
             struct probgraph *pralloc();
                pbh = pralloc();
                pbh->num = num - 1;
                scanf("%f", &(pbh->p));      /*reads in the probability*/
                pbh->next = flow[i];
                flow[i] = pbh;
                scanf("%d", &num);
            }
        }
    }
    scanf("%d", &num);
    if(num)    /* num == 0 means the end of input */
        er_in();
}

/*Return: a pointer points to a structure probgraph                          */
/*Function allocates enough memory space for structure probgraph and returns */
/*the pointer points to it.                                                  */

struct probgraph *pralloc()  /* standard */
{char *malloc();
    return((struct probgraph *) malloc(sizeof(struct probgraph)));
}

/*Procedure prints out an error message and then terminates the program      */

er_in()
{
    printf("Erroneus input.\n");
    exit(0);
}
```

Figure F8. Function Program 4.

```
/*Procedure initializes the empty state of HiTest in order to get it started  */
/*to run.  All the waitlists are set to empty and the hosts are idle.  All PEs*/
/*are set to get ready to start the first block.                               */

preload()
{
int i;
    for(i=0; i<K; i++)
        waitlist[i] = NULL;  /*all waitlists are empty*/
    for(i=0; i<N; i++)
    {
        host[i].time = 0;
        host[i].block = NULL;        /*all hosts are idle*/
    }
    for(i=0; i<M; i++)
    {
        PE[i].time = btime[0];
        PE[i].block = 0;        /*all PEs are ready to execute the first block*/
    }
}

/*Arguments: blocknum - an id no. of a block                                   */
/*           penum - an id no. of a PE                                         */
/*Return : an id no. of a block                                                */
/*Function chooses a new block for the PE to execute next according to the     */
/*behavior of a given flowgraph and probabilities of taking branches.  If PE   */
/*has currently finished the last block, 0 will be returned and a new case     */
/*will start.  Otherwise, a random number between 0 and 1 will be generated    */
/*and it will compared to the probabilities of all paths.  If the random num-  */
/*is smaller than or equal to the probability of a path then the id no. of     */
/*the block which the path leads to will be returned.                          */

int program(blocknum, penum)
int blocknum, penum;
{
    if(blocknum == K-1)          /*start a new case*/
        return(0);
    else
    {
    double rp, erand48();
    short int xsubi[3];
    struct probgraph *pbh;
        rp = erand48(xsubi);      /*generate a random number*/
        pbh = flow[blocknum];
        if(pbh)
        while(pbh)
        {
            if(rp <= pbh->p)     /*compare the number to all the probabilities*/
                return(pbh->num);
            rp = rp - pbh->p;
            pbh = pbh->next;
        }
        else
            er_in();
        printf("Warning : strange probabillity\n");
        return(flow[blocknum]->num);
    }
}
```

Figure F8. Continued.

The Flow-graphs of Real Programs Used for the Experiment.

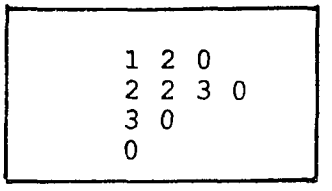

Figure F9. Program Loop.

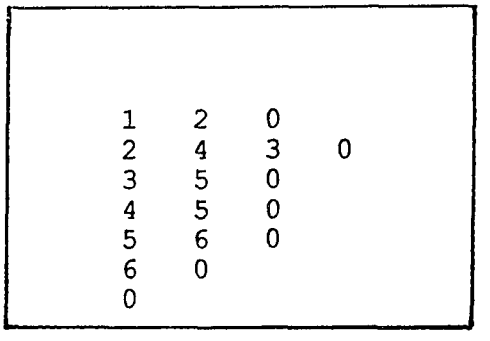

Figure F10. Program Push.

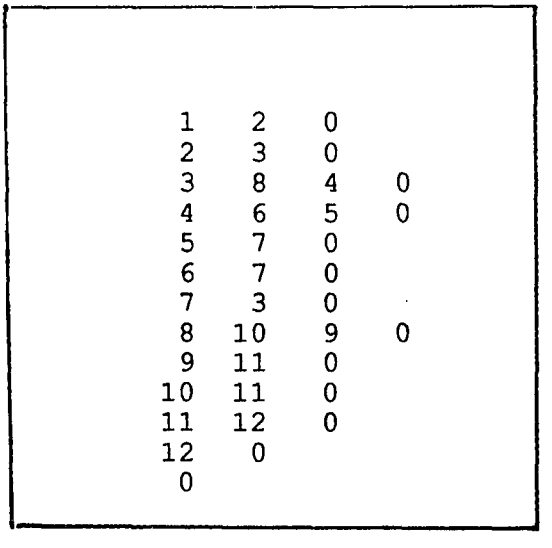

Figure F11. Program UpdateCG.

```
 1    2    0
 2    4    3    0
 3   17    0
 4    6    5    0
 5    7    0
 6    7    0
 7    9    8    0
 8   16    0
 9   11   10    0
10   15    0
11   12    0
12   14   13    0
13   12    0
14   15    0
15   16    0
16   17    0
17   18    0
18    0
 0
```

Figure F12. Program Openblock.

```
 1    2    0
 2    3    0
 3   14    4    0
 4   12    5    0
 5    6    0
 6   11    7    0
 7    9    8    0
 8   10    0
 9   10    0
10    6    0
11   13    0
12   13    0
13    3    0
14   16   15    0
15   17    0
16   17    0
17   18    0
18    0
 0
```

Figure F13. Program Tabulate.

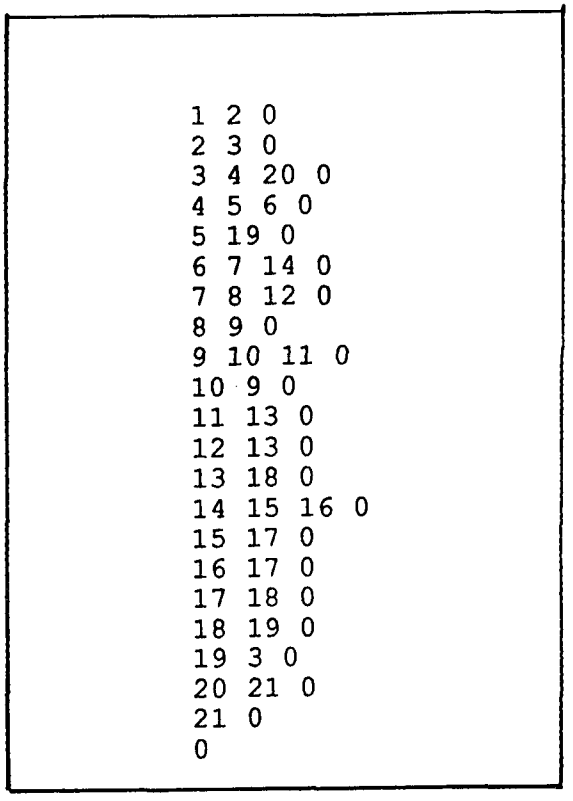

Figure F14. Program Expand.

```
 1    2    0
 2    3    0
 3    8    4    0
 4    5    0
 5    7    6    0
 6    5    0
 7    3    0
 8   34    9    0
 9   10    0
10   12   11    0
11   10    0
12   17   13    0
13   14    0
14   16   15    0
15   14    0
16   24    0
17   22   18    0
18   19    0
19   21   20    0
20   19    0
21   23    0
22   23    0
23   24    0
24   26   25    0
25   27    0
26   27    0
27   29   28    0
28   30    0
29   30    0
30   32   31    0
31   33    0
32   33    0
33   35    0
34   35    0
35   36    0
36    0
 0
```

Figure F15. Program Drawv.

```
 1   2   0
 2  35   3   0
 3  34  24  17  13   9   5   0
 4  36   0
 5   6   0
 6   8   7   0
 7   6   0
 8   4   0
 9  10   0
10  12  11   0
11  10   0
12   4   0
13  14   0
14  16  15   0
15  14   0
16   4   0
17  19  18   0
18  20   0
19  20   0
20  21   0
21  23  22   0
22  21   0
23   4   0
24  25   0
25  33  26   0
26  27   0
27  32  28   0
28  29   0
29  31  30   0
30  29   0
31  27   0
32  25   0
33   4   0
34   4   0
35  36   0
36  37   0
37   0
 0
```

Figure F16. Program Animate.

```
 1   2   0
 2   3   0
 3  41   4   0
 4   5   0
 5  10   6   0
 6   8   7   0
 7   9   0
 8   9   0
 9   5   0
10  11   0
11  40  12   0
12  13   0
13  24  14   0
14  22  15   0
15  20  16   0
16  18  17   0
17  19   0
18  19   0
19  21   0
20  21   0
21  23   0
22  23   0
23  13   0
24  25   0
25  27  26   0
26  25   0
27  35  28   0
28  29   0
29  34  30   0
30  32  31   0
31  33   0
32  33   0
33  29   0
34  36   0
35  36   0
36  38  37   0
37  39   0
38  39   0
39  11   0
40   3   0
41  42   0
42   0
 0
```

Figure F17. Program Patmatch.

```
 1   2   0
 2   3   0
 3  41   4   0
 4   5   0
 5  40   6   0
 6  38   7   0
 7   8   0
 8  10   9   0
 9   8   0
10  12  11   0
11  13   0
12  13   0
13  14   0
14  16  15   0
15  14   0
16  17   0
17  25  18   0
18  23  19   0
19  21  20   0
20  22   0
21  22   0
22  24   0
23  24   0
24  17   0
25  26   0
26  37  27   0
27  35  28   0
28  29   0
29  31  30   0
30  29   0
31  33  32   0
32  34   0
33  34   0
34  36   0
35  36   0
36  26   0
37  39   0
38  39   0
39   5   0
40   3   0
41  42   0
42   0
 0
```

Figure F18. Program Findpairs.

```
 1   2   0
 2  66   3   0
 3   4   0
 4   4   5   0
 5   7   6   0
 6  65   0
 7   9   8   0
 8  64   0
 9  11  10   0
10  63   0
11  13  12   0
12  62   0
13  14   0
14  58  15   0
15  16   0
16  27  17   0
17  25  18   0
18  20  19   0
19  24   0
20  22  21   0
21  23   0
22  23   0
23  24   0
24  26   0
25  26   0
26  28   0
27  28   0
28  16  29   0
29  31  30   0
30  32   0
31  32   0
32  39  35  34   0
33  14   0
34  33   0
35  37  36   0
36  38   0
37  38   0
38  33   0
39  41  40   0
40  57   0
41  55  42   0
42  44  43   0
43  48   0
44  46  45   0
45  47   0
46  47   0
47  48   0
48  50  49   0
49  51   0
50  51   0
51  53  52   0
52  54   0
53  54   0
54  56   0
55  56   0
56  57   0
57  33   0
58  60  59   0
59  61   0
60  61   0
61  62   0
62  63   0
63  64   0
64  65   0
65  67   0
66  67   0
67  68   0
68   0
 0
```

Figure F19. Program Parse2.

```
 1   2   0
 2   3   0
 3  42   4   0
 4  40   5   0
 5  30  29  25  11   7   0
 6  41   0
 7   9   8   0
 8  10  41   0
 9  10   0
10   6   0
11  13  12   0
12  14  41   0
13  14   0
14  23  15   0
15  22  21  20  19  18  17   0
16  24   0
17  16   0
18  16   0
19  16   0
20  16   0
21  16   0
22  16   0
23  24  41   0
24   6   0
25  27  26   0
26  28  41   0
27  28   0
28   6   0
29   6   0
30  32  31   0
31  33  41   0
32  33   0
33  38  34   0
34  37  36   0
35  39   0
36  35   0
37  35   0
38  39  41   0
39   6   0
40  41   0
41   3  72   0
42  44  43   0
43  72   0
44  49  45   0
45  47  46   0
46  48   0
47  48   0
48  71   0
49  51  50   0
50  52   0
51  52   0
52  54  53   0
53  70   0
54  62  55   0
55  57  56   0
56  61   0
57  59  58   0
58  60   0
59  60   0
60  61   0
61  69   0
62  64  63   0
63  68   0
64  66  65   0
65  67   0
66  67   0
67  68   0
68  69   0
69  70   0
70  71   0
71  72   0
72  73   0
73   0
 0
```

Figure F20. Program Switch.

```
 1   2   0
 2   4   3   0
 3   8   0
 4   6   5   0
 5   7   0
 6   7   0
 7   8   0
 8  47   9   0
 9  12  11   0
10  17  13   0
11  10   0
12  10   0
13  15  14   0
14  16   0
15  16   0
16  33   0
17  31  18   0
18  19   0
19  30  20   0
20  21   0
21  23  22   0
22  21   0
23  28  24   0
24  27  26   0
25  29   0
26  25   0
27  25   0
28  29   0
29  19   0
30  32   0
31  32   0
32  33   0
33  43  36  35   0
34  48   0
35  34   0
36  38  37   0
37  39   0
38  39   0
39  41  40   0
40  42   0
41  42   0
42  34   0
43  44   0
44  46  45   0
45  44   0
46  34   0
47  48   0
48  50  49   0
49  51   0
50  51   0
51  52   0
52  70  53   0
53  68  54   0
54  64  60  59  58  57  56   0
55  69   0
56  55   0
57  55   0
58  55   0
59  55   0
60  62  61   0
61  63   0
62  63   0
63  55   0
64  66  65   0
65  67   0
66  67   0
67  55   0
68  69   0
69  52   0
70  95  71   0
71  93  72   0
72  73   0
73  75  74   0
74  73   0
75  76   0
76  76  77   0
77  78   0
78  83  79   0
79  81  80   0
80  82   0
81  82   0
82  78   0
83  84   0
84  86  85   0
85  84   0
86  88  87   0
87  89   0
88  89   0
89  91  90   0
90  92   0
91  92   0
92  94   0
93  94   0
94 102   0
95  97  96   0
96 101   0
97  99  98   0
98 100   0
99 100   0
100 101   0
101 102   0
102 110 103   0
103 104   0
104 106 105   0
105 104   0
106 107   0
107 109 108   0
108 107   0
109 111   0
110 111   0
111 112   0
112   0
  0
```

Figure F21. Program Getprocdec.

```
  1    2    0
  2    3    0
  3    5    4    0
  4    3    0
  5    6    0
  6   81    7    0
  7   71   58   54    9    0
  8   76   75    0
  9   10    0
 10   18   11    0
 11   12    0
 12   14   13    0
 13   12    0
 14   15    0
 15   17   16    0
 16   15    0
 17   10    0
 18   20   19    0
 19   53    0
 20   22   21    0
 21   52    0
 22   24   23    0
 23   51    0
 24   26   25    0
 25   50    0
 26   49   45   44   43   42   41   40   39   38   37   36   35   34   33   32   31   30   29   28
 27   50    0
 28   27    0
 29   27    0
 30   27    0
 31   27    0
 32   27    0
 33   27    0
 34   27    0
 35   27    0
 36   27    0
 37   27    0
 38   27    0
 39   27    0
 40   27    0
 41   27    0
 42   27    0
 43   27    0
 44   27    0
 45   46    0
 46   48   47    0
 47   46    0
 48   27    0
 49   27    0
 50   51    0
 51   52    0
 52   53    0
 53    8    0
 54   56   55    0
 55   57    0
 56   57    0
 57    8    0
 58   63   59    0
 59   61   60    0
 60   62    0
 61   62    0
 62   70    0
 63   65   64    0
 64   69    0
 65   67   66    0
 66   68    0
```

Figure F22. Program Nexttok.

```
 67  68   0
 68  69   0
 69  70   0
 70   8   0
 71  73  72   0
 72  74   0
 73  74   0
 74   8   0
 75  80   0
 76  78  77   0
 77  79   0
 78  79   0
 79  80   0
 80   6   0
 81  82   0
 82  84  83   0
 83  82   0
 84  89  85   0
 85  87  86   0
 86  88   0
 87  88   0
 88  90   0
 89  90   0
 90 107  91   0
 91  96  92   0
 92  94  93   0
 93  95   0
 94  95   0
 95  97   0
 96  97   0
 97  99  98   0
 98 100   0
 99 100   0
100 105 101   0
101 102   0
102 104 103   0
103 102   0
104 106   0
105 106   0
106 114   0
107 108   0
108 110 109   0
109 108   0
110 112 111   0
111 113   0
112 113   0
113 114   0
114 116 115   0
115 117   0
116 117   0
117 118   0
118   0
  0
```

Figure F22. Continued.

The Tables of Results of the Software Simulation Experiments.

The Tables for the Second Type of Experiments.

Number of Hosts	Theoretical Utilization Factor	Simulated Utilization Factor
K = 12		
1	0.14286	0.15389
3	0.33333	0.35752
6	0.50000	0.53045
9	0.60000	0.62407
12	0.66667	0.67703
K = 20		
1	0.09091	0.09799
5	0.33333	0.34553
10	0.50000	0.51584
15	0.60000	0.61117
20	0.66667	0.67341
K = 40		
1	0.04762	0.04878
5	0.20000	0.20200
10	0.33333	0.33941
20	0.50000	0.51868
30	0.60000	0.60675
40	0.66667	0.67086

Table 4.2. Second type of experiments.

K = 60		
1	0.03226	0.03315
10	0.25000	0.25337
20	0.40000	0.40028
30	0.50000	0.50664
40	0.57143	0.57335
60	0.66667	0.67049
K = 100		
1	0.01961	0.02013
10	0.16667	0.16728
20	0.28571	0.28705
35	0.41176	0.41132
50	0.50000	0.50819
75	0.60000	0.60318
100	0.66667	0.67020
K = 150		
1	0.01316	0.01347
10	0.11765	0.11775
25	0.25000	0.25004
50	0.40000	0.39859
75	0.50000	0.51730
100	0.57143	0.57764
150	0.66667	0.66964

Table 4.2. Continued.

K = 250		
1	0.00794	0.00848
25	0.16667	0.16581
50	0.28571	0.28356
100	0.44444	0.43926
150	0.54545	0.54674
200	0.61538	0.62218
250	0.66667	0.66940
K = 400		
1	0.00498	0.00568
20	0.09091	0.09028
50	0.20000	0.19833
100	0.33333	0.33450
200	0.50000	0.50921
300	0.60000	0.60442
400	0.66667	0.67018
K = 700		
1	0.00285	0.00350
50	0.12500	0.12430
100	0.22222	0.22042
200	0.36364	0.36388
350	0.50000	0.51854
500	0.58824	0.59021
700	0.66667	0.66822

Table 4.2. Continued.

K = 1000		
1	0.00200	0.00273
50	0.09091	0.09084
100	0.16667	0.16586
200	0.28571	0.28351
350	0.41176	0.40662
500	0.50000	0.51967
700	0.58333	0.58360
1000	0.66667	0.66750

Table 4.2. Continued.

The Tables for the Third Type of Experiments.

Name of program: **loop**. Number of basic blocks: K = 3

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.40000	0.48255
2	0.57143	0.57217
3	0.66667	0.65375

Name of program: **push**. Number of basic blocks: K = 6

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.25000	0.40018
2	0.40000	0.39701
3	0.50000	0.51886
4	0.57143	0.61938
6	0.66667	0.67097

Table 4.3. The Third Type of Experiment.

Name of program: **updateCG**. Number of basic blocks: K = 12

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.14286	0.16747
2	0.25000	0.27963
4	0.40000	0.44469
6	0.50000	0.51691
9	0.60000	0.60443
12	0.66667	0.66993

Name of program: **openblock**. Number of basic blocks: K = 18

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.10000	0.10148
2	0.18182	0.19856
4	0.30769	0.32226
6	0.40000	0.41917
9	0.50000	0.56672
12	0.57143	0.59409
18	0.66667	0.66637

Table 4.3. Continued.

Name of program: **tabulate**. Number of basic blocks: K = 18

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.10000	0.14038
2	0.18182	0.22890
4	0.30769	0.36848
6	0.40000	0.47430
9	0.50000	0.60964
12	0.57143	0.66493
15	0.62500	0.69243
18	0.66667	0.71662

Name of program: **expand**. Number of basic blocks: K = 21

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.08696	0.07408
3	0.22222	0.20650
6	0.36364	0.31445
10	0.48780	0.44885
15	0.58824	0.54631
21	0.66667	0.65232

Table 4.3. Continued.

Name of program: **drawv**. Number of basic blocks: K = 36

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.05263	0.06400
3	0.14286	0.16650
6	0.25000	0.28792
12	0.40000	0.52447
18	0.50000	0.50097
26	0.59091	0.61846
36	0.66667	0.66379

Name of program: **animate**. Number of basic blocks: K = 37

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.05128	0.04895
6	0.24490	0.23889
15	0.44776	0.47889
25	0.57471	0.57854
37	0.66667	0.66077

Table 4.3. Continued.

Name of program: **patmatch**. Number of basic blocks: K = 42

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.04545	0.03432
3	0.12500	0.14750
6	0.22222	0.17307
12	0.36364	0.37129
18	0.46154	0.47292
21	0.50000	0.52566
36	0.63158	0.64514
42	0.66667	0.66687

Name of program: **findpairs**. Number of basic blocks: K = 42

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.04545	0.03458
3	0.12500	0.14513
6	0.22222	0.23721
15	0.41667	0.47798
21	0.50000	0.51413
36	0.63158	0.64124
42	0.66667	0.67245

Table 4.3. Continued.

Name of program: **parse2**. Number of basic blocks: K = 68

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.02857	0.01611
6	0.15000	0.09758
15	0.30612	0.22033
34	0.50000	0.42325
45	0.56962	0.51089
68	0.66667	0.72868

Name of program: **switch**. Number of basic blocks: K = 73

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.02667	0.03084
7	0.16092	0.14944
15	0.29126	0.26996
36	0.49655	0.48132
50	0.57803	0.57064
73	0.66667	0.67416

Table 4.3. Continued.

Name of program: **getprocdec**. Number of basic blocks: K = 112

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.01754	0.01457
6	0.09677	0.07292
12	0.17647	0.14600
25	0.30864	0.24027
40	0.41667	0.35027
56	0.50000	0.42990
70	0.55556	0.48645
90	0.61644	0.57986
112	0.66667	0.67438

Name of program: **nexttok**. Number of basic blocks: K = 118

Number of Hosts	Utilization factor/cal	Utilization factor/sim
1	0.01666	0.01861
6	0.09231	0.12278
12	0.16901	0.22530
24	0.28916	0.35502
40	0.40404	0.45388
59	0.50000	0.56564
80	0.57554	0.58299
100	0.62893	0.60847
118	0.66667	0.61835

Table 4.3. Continued.

Tables for the Fourth Type of Experiments.

Program name: **drawv**

Probability after block 3 is changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.44%	-7.85%	11.94%
6	0.250	0.260	4.19%	-0.46%	1.08%
12	0.400	0.440	10.01%	-12.94%	21.06%
18	0.500	0.506	1.13%	-2.67%	3.88%
25	0.581	0.569	-2.08%	-4.34%	6.52%
36	0.667	0.659	-1.17%	-0.19%	0.21%

Probability after block 5 is changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.30%	-9.84%	24.41%
6	0.250	0.253	1.23%	-11.16%	5.29%
12	0.400	0.434	8.52%	-4.38%	11.04%
18	0.500	0.516	3.17%	-3.97%	10.33%
25	0.581	0.580	-0.26%	-4.78%	11.57%
36	0.667	0.657	-1.45%	-1.36%	0.41%

Probability after block 8 is changing.

N	calc	avg	diff	min	max
1	0.053	0.053	1.61%	-4.18%	2.66%
6	0.250	0.261	4.27%	-1.76%	2.73%
12	0.400	0.425	6.30%	-2.64%	4.15%
18	0.500	0.504	0.89%	-1.55%	2.38%
25	0.581	0.564	-3.04%	-0.45%	0.39%
36	0.667	0.658	-1.28%	-0.63%	0.35%

Table 4.4. The Fourth Type of Experiments.

Probability after block 10 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.96%	-0.42%	1.32%
6	0.250	0.262	4.97%	-1.19%	4.05%
12	0.400	0.426	6.41%	-1.06%	2.45%
18	0.500	0.500	-0.08%	-2.44%	0.86%
25	0.581	0.570	-2.03%	-1.46%	4.39%
36	0.667	0.661	-0.87%	-0.30%	0.85%

Probability after block 12 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.19%	-2.98%	2.66%
6	0.250	0.259	3.76%	-2.38%	2.45%
12	0.400	0.422	5.56%	-0.21%	0.12%
18	0.500	0.503	0.51%	-0.16%	0.20%
25	0.581	0.563	-3.16%	-0.59%	0.49%
36	0.667	0.659	-1.19%	-0.18%	0.12%

Probability after block 14 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.38%	-1.91%	3.68%
6	0.250	0.276	10.42%	-6.87%	24.47%
12	0.400	0.428	6.96%	-1.37%	4.74%
18	0.500	0.506	1.14%	-0.72%	2.32%
25	0.581	0.573	-1.37%	-1.98%	7.10%
36	0.667	0.666	-0.12%	-1.12%	4.10%

Table 4.4. Continued.

Probability after block 17 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.31%	-0.85%	0.94%
6	0.250	0.260	3.86%	-0.20%	0.23%
12	0.400	0.422	5.58%	-0.54%	0.40%
18	0.500	0.502	0.50%	-0.43%	0.39%
25	0.581	0.563	-3.13%	-0.08%	0.18%
36	0.667	0.659	-1.17%	-0.05%	0.07%

Probability after block 19 is changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.34%	-3.42%	10.48%
6	0.250	0.273	9.05%	-4.88%	18.65%
12	0.400	0.435	8.77%	-3.09%	11.30%
18	0.500	0.549	9.83%	-9.10%	33.86%
25	0.581	0.575	-1.18%	-2.27%	7.99%
36	0.667	0.654	-1.97%	-3.25%	0.90%

Probability after block 24 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.52%	-0.09%	0.10%
6	0.250	0.259	3.78%	-0.06%	0.11%
12	0.400	0.422	5.58%	-0.72%	0.65%
18	0.500	0.503	0.53%	-0.17%	0.16%
25	0.581	0.563	-3.13%	-0.07%	0.03%
36	0.667	0.659	-1.12%	-0.08%	0.10%

Table 4.4. Continued.

Probability after block 27 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.63%	-1.23%	1.18%
6	0.250	0.260	3.82%	-0.55%	0.59%
12	0.400	0.422	5.50%	-0.96%	0.90%
18	0.500	0.502	0.49%	-0.61%	0.61%
25	0.581	0.563	-3.10%	-0.50%	0.53%
36	0.667	0.659	-1.20%	-0.08%	0.10%

Probability after block 30 is changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.29%	-2.61%	2.79%
6	0.250	0.260	3.89%	-0.79%	0.98%
12	0.400	0.422	5.57%	-0.55%	0.72%
18	0.500	0.503	0.51%	-0.56%	0.66%
25	0.581	0.564	-3.04%	-0.21%	0.44%
36	0.667	0.659	-1.18%	-0.04%	0.03%

Table 4.4. Continued.

Program name: **expand**

Probabilities after block 3 are changing

N	avg	diff	min	max
1	0.098556	13.34%	-36.39%	41.82%
3	0.214116	-3.65%	-14.11%	13.15%
5	0.311294	-3.50%	-18.09%	15.97%
10	0.497712	2.03%	-4.85%	3.97%
15	0.631786	7.40%	-2.36%	2.19%
21	0.655426	-1.69%	-1.79%	0.68%

Probabilities after block 4 are changing

N	avg	diff	min	max
1	0.094404	8.56%	-6.43%	4.92%
3	0.213178	-4.07%	-10.65%	9.03%
5	0.311844	-3.33%	-10.53%	8.58%
10	0.496244	1.73%	-7.20%	5.54%
15	0.632106	7.46%	-0.88%	1.20%
21	0.658994	-1.15%	-0.14%	0.25%

Probabilities after block 6 are changing

N	avg	diff	min	max
1	0.095430	9.74%	-1.16%	0.60%
3	0.215170	-3.17%	-1.02%	0.73%
5	0.314342	-2.55%	-1.95%	1.75%
10	0.500322	2.57%	-1.85%	2.12%
15	0.631298	7.32%	-0.52%	0.43%
21	0.658436	-1.23%	-0.28%	0.30%

Table 4.4. Continued.

Probabilities after block 7 are changing

N	avg	diff	min	max
1	0.095340	9.64%	-3.17%	2.92%
3	0.214794	-3.34%	-2.45%	1.69%
5	0.313818	-2.72%	-2.25%	1.81%
10	0.499636	2.43%	-2.25%	2.14%
15	0.631786	7.40%	-0.96%	0.95%
21	0.658464	-1.23%	-0.07%	0.04%

Probabilities after block 9 are changing

N	avg	diff	min	max
1	0.100838	15.96%	-7.20%	21.43%
3	0.237050	6.67%	-10.14%	36.54%
5	0.336266	4.24%	-7.39%	25.01%
10	0.513390	5.24%	-3.19%	10.23%
15	0.634564	7.88%	-0.67%	2.01%
21	0.656492	-1.53%	-1.22%	0.34%

Probabilities after block 14 are changing

N	avg	diff	min	max
1	0.095682	10.03%	-0.18%	0.19%
3	0.214790	-3.34%	-0.40%	0.45%
5	0.314016	-2.66%	-1.24%	0.75%
10	0.499962	2.49%	-0.25%	0.16%
15	0.631412	7.34%	-0.05%	0.09%
21	0.658578	-1.21%	-0.13%	0.12%

Table 4.4. Continued.

Program name: **findpairs**

Probability after block 3 is changing.

N	calc	avg	diff	min	max
1	0.045	0.047	4.35%	-9.65%	6.63%
5	0.192	0.201	4.73%	-2.72%	6.02%
10	0.323	0.335	3.99%	-6.28%	13.49%
20	0.488	0.485	-0.65%	-0.08%	0.20%
30	0.588	0.591	0.43%	-0.37%	0.62%
42	0.667	0.669	0.29%	-2.37%	1.13%

Probability after block 5 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	5.66%	-5.94%	4.58%
5	0.192	0.199	3.65%	-2.83%	4.07%
10	0.323	0.329	2.00%	-5.93%	8.71%
20	0.488	0.485	-0.68%	-0.19%	0.25%
30	0.588	0.590	0.37%	-0.19%	0.32%
42	0.667	0.670	0.47%	-2.40%	1.81%

Probability after block 6 is changing.

N	calc	avg	diff	min	max
1	0.045	0.047	4.11%	-11.43%	7.06%
5	0.192	0.197	2.61%	-3.49%	1.95%
10	0.323	0.323	0.06%	-3.33%	2.03%
20	0.488	0.480	-1.66%	-4.86%	2.76%
30	0.588	0.585	-0.48%	-4.12%	2.44%
42	0.667	0.671	0.63%	-0.54%	0.28%

Table 4.4. Continued.

Probability after block 8 is changing.

N	calc	avg	diff	min	max
1	0.045	0.047	3.79%	-8.34%	2.79%
5	0.192	0.197	2.41%	-2.77%	0.81%
10	0.323	0.326	1.15%	-0.57%	1.42%
20	0.488	0.484	-0.88%	-0.89%	0.32%
30	0.588	0.591	0.43%	-0.19%	0.53%
42	0.667	0.664	-0.47%	-4.88%	1.31%

Probability after block 10 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	6.31%	-0.14%	0.17%
5	0.192	0.199	3.23%	-1.12%	1.27%
10	0.323	0.325	0.78%	-0.20%	0.28%
20	0.488	0.484	-0.68%	-0.11%	0.10%
30	0.588	0.590	0.30%	-0.45%	0.42%
42	0.667	0.672	0.81%	-0.05%	0.03%

Probability after block 14 is changing.

N	calc	avg	diff	min	max
1	0.045	0.047	4.40%	-7.34%	2.65%
5	0.192	0.197	2.40%	-3.09%	1.16%
10	0.323	0.327	1.36%	-0.77%	2.30%
20	0.488	0.485	-0.51%	-0.29%	0.55%
30	0.588	0.585	-0.47%	-3.02%	0.96%
42	0.667	0.676	1.46%	-0.86%	2.73%

Table 4.4. Continued.

Probability after block 17 is changing.

N	calc	avg	diff	min	max
1	0.045	0.050	10.06%	-6.22%	14.29%
5	0.192	0.200	3.88%	-1.31%	3.14%
10	0.323	0.329	2.08%	-2.11%	5.19%
20	0.488	0.497	1.84%	-3.91%	10.13%
30	0.588	0.592	0.68%	-0.59%	1.58%
42	0.667	0.671	0.72%	-0.30%	0.12%

Probability after block 18 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	6.56%	-2.23%	2.50%
5	0.192	0.198	2.95%	-0.73%	0.77%
10	0.323	0.325	0.73%	-0.25%	0.31%
20	0.488	0.484	-0.72%	-0.08%	0.10%
30	0.588	0.590	0.31%	-0.10%	0.05%
42	0.667	0.672	0.79%	-0.08%	0.05%

Probability after block 19 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	6.15%	-0.22%	0.17%
5	0.192	0.198	3.22%	-0.23%	0.22%
10	0.323	0.325	0.78%	-0.23%	0.34%
20	0.488	0.485	-0.64%	-0.13%	0.18%
30	0.588	0.590	0.29%	-0.17%	0.12%
42	0.667	0.672	0.81%	-0.07%	0.03%

Table 4.4. Continued.

Probability after block 26 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	5.52%	-3.23%	1.48%
5	0.192	0.198	2.78%	-2.17%	0.89%
10	0.323	0.323	0.06%	-2.94%	1.29%
20	0.488	0.483	-1.03%	-1.54%	0.65%
30	0.588	0.592	0.67%	-0.70%	1.50%
42	0.667	0.677	1.48%	-1.15%	2.69%

Probability after block 27 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	6.14%	-0.69%	0.62%
5	0.192	0.198	3.11%	-1.19%	0.77%
10	0.323	0.325	0.79%	-0.58%	0.67%
20	0.488	0.484	-0.69%	-0.16%	0.26%
30	0.588	0.590	0.31%	-0.07%	0.08%
42	0.667	0.672	0.80%	-0.32%	0.36%

Probability after block 29 is changing.

N	calc	avg	diff	min	max
1	0.045	0.047	4.16%	-7.82%	2.81%
5	0.192	0.191	-0.89%	-16.02%	4.79%
10	0.323	0.312	-3.22%	-16.23%	4.60%
20	0.488	0.467	-4.22%	-14.61%	4.03%
30	0.588	0.571	-2.99%	-13.35%	3.67%
42	0.667	0.667	0.12%	-2.71%	0.72%

Table 4.4. Continued.

Probability after block 31 is changing.

N	calc	avg	diff	min	max
1	0.045	0.048	6.17%	-0.01%	0.01%
5	0.192	0.198	3.21%	-0.40%	0.27%
10	0.323	0.325	0.80%	-0.23%	0.24%
20	0.488	0.484	-0.76%	-0.15%	0.15%
30	0.588	0.590	0.34%	-0.08%	0.04%
42	0.667	0.672	0.85%	-0.05%	0.06%

Table 4.4. Continued.

Program name: **parse2**

Probability after block 2 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	8.16%	-8.93%	6.31%
10	0.227	0.230	1.02%	-7.74%	13.05%
25	0.424	0.407	-4.00%	-1.53%	2.17%
40	0.541	0.558	3.19%	-1.48%	2.29%
55	0.618	0.660	6.73%	-1.57%	1.52%
68	0.667	0.700	4.98%	-5.81%	4.23%

Probability after block 4 is changing.

N	calc	avg	diff	min	max
1	0.029	0.029	1.93%	-31.94%	12.77%
10	0.227	0.215	-5.47%	-16.90%	6.57%
25	0.424	0.425	0.19%	-5.46%	17.57%
40	0.541	0.583	7.93%	-5.93%	18.74%
55	0.618	0.723	17.07%	-10.82%	34.81%
68	0.667	0.760	13.97%	-8.92%	28.38%

Probability after block 5 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	10.23%	-7.91%	8.47%
10	0.227	0.224	-1.40%	-0.43%	0.74%
25	0.424	0.407	-3.98%	-2.96%	3.41%
40	0.541	0.557	2.98%	-1.44%	1.73%
55	0.618	0.660	6.88%	-1.52%	1.75%
68	0.667	0.706	5.91%	-1.23%	1.35%

Table 4.4. Continued.

Probability after block 7 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.11%	-3.73%	4.09%
10	0.227	0.224	-1.45%	-0.67%	1.10%
25	0.424	0.406	-4.23%	-0.47%	0.53%
40	0.541	0.556	2.91%	-0.67%	0.81%
55	0.618	0.661	6.91%	-1.40%	1.55%
68	0.667	0.706	5.84%	-0.38%	0.61%

Probability after block 9 is changing.

N	calc	avg	diff	min	ma
1	0.029	0.031	8.83%	-1.84%	1.89%
10	0.227	0.224	-1.51%	-0.21%	0.26%
25	0.424	0.405	-4.32%	-0.66%	0.59%
40	0.541	0.556	2.82%	-0.31%	0.24%
55	0.618	0.660	6.77%	-0.19%	0.21%
68	0.667	0.706	5.83%	-0.50%	0.44%

Probability after block 11 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.91%	-0.96%	0.60%
10	0.227	0.224	-1.50%	-0.75%	0.79%
25	0.424	0.406	-4.19%	-0.46%	0.70%
40	0.541	0.556	2.88%	-0.23%	0.22%
55	0.618	0.660	6.83%	-0.56%	0.61%
68	0.667	0.706	5.87%	-0.39%	0.48%

Table 4.4. Continued.

Probability after block 14 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.31%	-0.58%	0.64%
10	0.227	0.227	0.03%	-2.23%	6.43%
25	0.424	0.411	-3.11%	-1.57%	4.76%
40	0.541	0.558	3.14%	-0.38%	1.13%
55	0.618	0.654	5.83%	-3.78%	1.47%
68	0.667	0.698	4.66%	-4.49%	1.67%

Probability after block 16 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.58%	-0.50%	0.43%
10	0.227	0.224	-1.44%	-0.29%	0.29%
25	0.424	0.406	-4.26%	-0.18%	0.16%
40	0.541	0.556	2.84%	-0.12%	0.15%
55	0.618	0.659	6.71%	-0.05%	0.05%
68	0.667	0.705	5.82%	-0.14%	0.16%

Probability after block 17 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.28%	-0.77%	0.67%
10	0.227	0.224	-1.42%	-0.30%	0.20%
25	0.424	0.406	-4.29%	-0.18%	0.15%
40	0.541	0.556	2.90%	-0.11%	0.11%
55	0.618	0.660	6.84%	-0.11%	0.19%
68	0.667	0.705	5.81%	-0.03%	0.05%

Table 4.4. Continued.

Probability after block 18 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.09%	-0.80%	0.84%
10	0.227	0.224	-1.35%	-0.37%	0.33%
25	0.424	0.406	-4.28%	-0.13%	0.08%
40	0.541	0.556	2.86%	-0.08%	0.08%
55	0.618	0.660	6.83%	-0.09%	0.10%
68	0.667	0.705	5.81%	-0.07%	0.04%

Probability after block 20 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	10.01%	0.00%	0.00%
10	0.227	0.224	-1.56%	-0.23%	0.32%
25	0.424	0.406	-4.24%	-0.16%	0.18%
40	0.541	0.556	2.88%	-0.14%	0.12%
55	0.618	0.660	6.84%	-0.10%	0.07%
68	0.667	0.705	5.81%	-0.09%	0.04%

Probability after block 28 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.45%	-0.96%	1.12%
10	0.227	0.227	0.01%	-1.73%	5.72%
25	0.424	0.410	-3.13%	-1.39%	4.57%
40	0.541	0.559	3.50%	-0.68%	2.27%
55	0.618	0.657	6.29%	-2.08%	0.68%
68	0.667	0.698	4.73%	-4.24%	1.20%

Table 4.4. Continued.

Probability after block 29 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.33%	-0.76%	0.62%
10	0.227	0.223	-1.72%	-0.35%	0.38%
25	0.424	0.406	-4.19%	-0.22%	0.14%
40	0.541	0.556	2.85%	-0.08%	0.16%
55	0.618	0.660	6.84%	-0.12%	0.16%
68	0.667	0.706	5.85%	-0.02%	0.03%

Probability after block 32 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	8.87%	-0.56%	1.05%
10	0.227	0.224	-1.41%	-0.31%	0.62%
25	0.424	0.406	-4.25%	-0.16%	0.24%
40	0.541	0.556	2.87%	-0.05%	0.11%
55	0.618	0.660	6.83%	-0.09%	0.06%
68	0.667	0.705	5.82%	-0.07%	0.04%

Probability after block 34 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.34%	-0.41%	0.61%
10	0.227	0.224	-1.52%	-0.21%	0.19%
25	0.424	0.406	-4.28%	-0.13%	0.12%
40	0.541	0.556	2.89%	-0.05%	0.02%
55	0.618	0.660	6.82%	-0.08%	0.07%
68	0.667	0.705	5.82%	-0.03%	0.04%

Table 4.4. Continued.

Probability after block 35 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	10.01%	0.00%	0.00%
10	0.227	0.224	-1.51%	-0.21%	0.32%
25	0.424	0.406	-4.24%	-0.17%	0.13%
40	0.541	0.556	2.85%	-0.12%	0.12%
55	0.618	0.660	6.84%	-0.10%	0.10%
68	0.667	0.705	5.82%	-0.09%	0.12%

Probability after block 39 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.82%	-0.27%	0.17%
10	0.227	0.224	-1.47%	-0.26%	0.28%
25	0.424	0.406	-4.18%	-0.23%	0.17%
40	0.541	0.556	2.82%	-0.09%	0.11%
55	0.618	0.660	6.83%	-0.10%	0.06%
68	0.667	0.706	5.83%	-0.07%	0.06%

Probability after block 41 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.69%	-0.92%	0.29%
10	0.227	0.224	-1.62%	-0.20%	0.21%
25	0.424	0.406	-4.25%	-0.15%	0.06%
40	0.541	0.556	2.86%	-0.09%	0.06%
55	0.618	0.660	6.84%	-0.10%	0.08%
68	0.667	0.705	5.82%	-0.13%	0.09%

Table 4.4. Continued.

Probability after block 42 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.37%	-0.89%	0.58%
10	0.227	0.224	-1.47%	-0.26%	0.20%
25	0.424	0.406	-4.21%	-0.20%	0.18%
40	0.541	0.556	2.87%	-0.03%	0.02%
55	0.618	0.660	6.84%	-0.10%	0.13%
68	0.667	0.705	5.81%	-0.13%	0.09%

Probability after block 44 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	10.01%	0.00%	0.00%
10	0.227	0.224	-1.65%	-0.07%	0.06%
25	0.424	0.406	-4.30%	-0.11%	0.07%
40	0.541	0.556	2.87%	-0.09%	0.09%
55	0.618	0.660	6.82%	-0.09%	0.15%
68	0.667	0.706	5.85%	-0.05%	0.05%

Probability after block 48 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.92%	-0.33%	0.08%
10	0.227	0.224	-1.57%	-0.16%	0.15%
25	0.424	0.406	-4.26%	-0.15%	0.12%
40	0.541	0.556	2.88%	-0.04%	0.03%
55	0.618	0.660	6.76%	-0.11%	0.15%
68	0.667	0.706	5.88%	-0.04%	0.10%

Table 4.4. Continued.

Probability after block 51 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.74%	-0.29%	0.25%
10	0.227	0.223	-1.72%	-0.11%	0.08%
25	0.424	0.406	-4.19%	-0.22%	0.12%
40	0.541	0.556	2.91%	-0.08%	0.07%
55	0.618	0.660	6.74%	-0.09%	0.07%
68	0.667	0.705	5.80%	-0.17%	0.07%

Probability after block 58 is changing.

N	calc	avg	diff	min	max
1	0.029	0.031	9.11%	-1.33%	0.82%
10	0.227	0.223	-1.74%	-0.34%	0.33%
25	0.424	0.406	-4.24%	-0.17%	0.15%
40	0.541	0.556	2.88%	-0.14%	0.19%
55	0.618	0.660	6.79%	-0.13%	0.15%
68	0.667	0.706	5.83%	-0.06%	0.10%

Table 4.4. Continued.

Tables for the Forth Type of Experiments.(2)

Program name: **drawv**

Probabilities after block 3 and 5 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.38%	-11.20%	29.14%
6	0.250	0.257	2.96%	-12.90%	20.96%
12	0.400	0.447	11.81%	-14.34%	35.27%
18	0.500	0.514	2.87%	-4.32%	10.69%
25	0.581	0.579	-0.37%	-5.98%	11.89%
36	0.667	0.658	-1.36%	-1.45%	0.64%

Probabilities after block 3 and 8 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.50%	-20.18%	12.96%
6	0.250	0.262	4.85%	-2.62%	7.34%
12	0.400	0.441	10.14%	-15.43%	20.92%
18	0.500	0.507	1.47%	-3.77%	3.53%
25	0.581	0.569	-2.19%	-6.95%	6.64%
36	0.667	0.658	-1.30%	-1.29%	0.75%

Probabilities after block 5 and 8 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.36%	-16.76%	26.18%
6	0.250	0.255	1.81%	-11.79%	11.67%
12	0.400	0.436	9.09%	-6.30%	10.78%
18	0.500	0.517	3.45%	-5.30%	10.17%
25	0.581	0.580	-0.29%	-5.80%	11.67%
36	0.667	0.657	-1.51%	-1.31%	0.87%

Table 4.5. Change of Probabilities in Two Pairs of Branches of the Flow-graph.

Probabilities after block 3 and 10 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.85%	-9.98%	11.50%
6	0.250	0.263	5.11%	-1.37%	4.09%
12	0.400	0.443	10.81%	-13.78%	20.19%
18	0.500	0.504	0.72%	-3.22%	4.31%
25	0.581	0.575	-1.14%	-6.05%	5.51%
36	0.667	0.660	-0.96%	-0.40%	0.94%

Probabilities after block 5 and 10 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.80%	-10.97%	24.57%
6	0.250	0.256	2.33%	-12.15%	6.92%
12	0.400	0.436	9.07%	-4.86%	10.55%
18	0.500	0.513	2.54%	-4.96%	11.01%
25	0.581	0.584	0.50%	-6.02%	10.72%
36	0.667	0.659	-1.21%	-1.60%	1.19%

Probabilities after block 8 and 10 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	1.95%	-4.51%	2.93%
6	0.250	0.263	5.20%	-2.69%	3.84%
12	0.400	0.428	7.01%	-3.44%	3.46%
18	0.500	0.502	0.42%	-3.03%	2.86%
25	0.581	0.569	-2.18%	-1.70%	4.59%
36	0.667	0.660	-1.06%	-0.85%	1.06%

Table 4.5. Continued.

Probabilities after block 3 and 12 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.29%	-12.13%	12.10%
6	0.250	0.261	4.21%	-3.89%	3.39%
12	0.400	0.440	9.94%	-13.03%	21.13%
18	0.500	0.506	1.17%	-2.71%	3.85%
25	0.581	0.569	-2.10%	-5.18%	6.54%
36	0.667	0.659	-1.19%	-0.21%	0.45%

Probabilities after block 5 and 12 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.60%	-13.27%	25.36%
6	0.250	0.253	1.29%	-11.35%	8.47%
12	0.400	0.434	8.57%	-4.44%	11.12%
18	0.500	0.516	3.19%	-4.16%	10.34%
25	0.581	0.580	-0.28%	-5.21%	11.58%
36	0.667	0.657	-1.44%	-1.37%	0.58%

Probabilities after block 8 and 12 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	1.71%	-4.28%	7.23%
6	0.250	0.261	4.31%	-5.24%	2.69%
12	0.400	0.425	6.32%	-2.68%	4.13%
18	0.500	0.504	0.90%	-1.76%	2.38%
25	0.581	0.564	-3.05%	-1.10%	0.56%
36	0.667	0.658	-1.29%	-0.61%	0.63%

Table 4.5. Continued.

Probabilities after block 10 and 12 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.81%	-3.57%	2.68%
6	0.250	0.263	5.02%	-3.71%	4.00%
12	0.400	0.426	6.42%	-1.06%	2.45%
18	0.500	0.500	-0.10%	-2.43%	0.96%
25	0.581	0.570	-2.03%	-1.98%	4.39%
36	0.667	0.661	-0.90%	-0.49%	0.87%

Probabilities after block 3 and 14 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.45%	-9.28%	10.86%
6	0.250	0.274	9.61%	-6.18%	27.59%
12	0.400	0.445	11.23%	-14.11%	19.73%
18	0.500	0.509	1.70%	-3.25%	3.30%
25	0.581	0.578	-0.60%	-6.26%	6.27%
36	0.667	0.664	-0.36%	-1.03%	4.44%

Probabilities after block 5 and 14 are changing.

N	calc	avg	diff	min	max
1	0.053	0.058	9.28%	-10.98%	24.11%
6	0.250	0.268	7.09%	-16.05%	30.03%
12	0.400	0.438	9.56%	-5.29%	10.08%
18	0.500	0.518	3.61%	-4.67%	9.88%
25	0.581	0.588	1.10%	-6.35%	10.06%
36	0.667	0.663	-0.56%	-2.24%	4.58%

Table 4.5. Continued.

Probabilities after block 8 and 14 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.41%	-4.93%	7.50%
6	0.250	0.274	9.77%	-7.69%	26.77%
12	0.400	0.430	7.42%	-4.01%	4.29%
18	0.500	0.507	1.40%	-2.34%	2.08%
25	0.581	0.572	-1.64%	-2.16%	7.52%
36	0.667	0.664	-0.47%	-1.44%	4.55%

Probabilities after block 10 and 14 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.47%	-2.77%	3.95%
6	0.250	0.276	10.38%	-6.84%	25.22%
12	0.400	0.430	7.44%	-2.00%	4.27%
18	0.500	0.502	0.40%	-2.92%	3.15%
25	0.581	0.578	-0.62%	-3.12%	6.32%
36	0.667	0.666	-0.08%	-1.37%	4.06%

Probabilities after block 12 and 14 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.08%	-3.82%	8.13%
6	0.250	0.273	9.18%	-7.23%	29.63%
12	0.400	0.427	6.68%	-1.27%	5.28%
18	0.500	0.505	1.04%	-0.79%	2.56%
25	0.581	0.571	-1.76%	-2.01%	7.83%
36	0.667	0.664	-0.37%	-1.02%	4.46%

Table 4.5. Continued.

Probabilities after block 3 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.40%	-8.98%	11.99%
6	0.250	0.260	4.20%	-0.72%	1.08%
12	0.400	0.440	9.93%	-13.40%	21.14%
18	0.500	0.506	1.13%	-3.40%	3.88%
25	0.581	0.569	-2.09%	-4.48%	6.53%
36	0.667	0.659	-1.17%	-0.18%	0.22%

Probabilities after block 5 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.45%	-10.75%	25.13%
6	0.250	0.253	1.29%	-11.22%	5.54%
12	0.400	0.434	8.55%	-4.41%	11.08%
18	0.500	0.516	3.14%	-4.56%	10.38%
25	0.581	0.580	-0.31%	-4.86%	11.62%
36	0.667	0.657	-1.42%	-1.39%	0.59%

Probabilities after block 8 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.053	1.48%	-4.06%	3.11%
6	0.250	0.261	4.33%	-2.18%	2.68%
12	0.400	0.425	6.29%	-3.48%	4.16%
18	0.500	0.504	0.87%	-2.45%	2.40%
25	0.581	0.564	-3.04%	-0.46%	0.39%
36	0.667	0.658	-1.28%	-0.63%	0.47%

Table 4.5. Continued.

Probabilities after block 10 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.57%	-1.86%	1.70%
6	0.250	0.263	5.02%	-1.42%	4.00%
12	0.400	0.426	6.40%	-1.32%	2.46%
18	0.500	0.499	-0.10%	-2.42%	1.38%
25	0.581	0.570	-2.01%	-1.53%	4.38%
36	0.667	0.661	-0.89%	-0.40%	0.87%

Probabilities after block 12 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.13%	-3.33%	2.71%
6	0.250	0.260	3.82%	-2.76%	2.40%
12	0.400	0.422	5.55%	-0.86%	0.87%
18	0.500	0.502	0.49%	-0.83%	1.10%
25	0.581	0.563	-3.14%	-0.68%	0.47%
36	0.667	0.659	-1.18%	-0.22%	0.11%

Probabilities after block 14 and 17 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.13%	-2.04%	4.43%
6	0.250	0.277	10.67%	-7.21%	25.29%
12	0.400	0.428	6.91%	-1.79%	4.78%
18	0.500	0.506	1.12%	-1.04%	2.39%
25	0.581	0.574	-1.35%	-2.05%	7.09%
36	0.667	0.666	-0.14%	-1.19%	4.13%

Probabilities after block 3 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.056	7.18%	-11.11%	12.27%
6	0.250	0.271	8.31%	-4.46%	21.30%
12	0.400	0.450	12.60%	-15.49%	18.27%
18	0.500	0.543	8.52%	-10.11%	37.93%
25	0.581	0.579	-0.41%	-6.47%	7.45%
36	0.667	0.654	-1.83%	-3.53%	0.89%

Table 4.5. Continued.

Probabilities after block 5 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.059	11.37%	-12.92%	23.38%
6	0.250	0.263	5.34%	-14.65%	23.50%
12	0.400	0.444	11.07%	-6.70%	9.00%
18	0.500	0.553	10.55%	-11.29%	32.99%
25	0.581	0.589	1.30%	-6.67%	9.86%
36	0.667	0.653	-2.10%	-3.24%	1.26%

Probabilities after block 8 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.11%	-6.48%	18.58%
6	0.250	0.271	8.27%	-5.83%	20.76%
12	0.400	0.435	8.87%	-5.37%	11.71%
18	0.500	0.541	8.18%	-9.16%	36.53%
25	0.581	0.573	-1.46%	-2.38%	9.04%
36	0.667	0.654	-1.94%	-3.47%	1.11%

Probabilities after block 10 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.33%	-4.90%	13.70%
6	0.250	0.272	9.00%	-4.84%	18.70%
12	0.400	0.435	8.87%	-3.29%	11.45%
18	0.500	0.536	7.29%	-9.14%	37.57%
25	0.581	0.579	-0.41%	-3.26%	7.48%
36	0.667	0.656	-1.55%	-3.69%	1.54%

Table 4.5. Continued.

Probabilities after block 12 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.08%	-6.28%	23.63%
6	0.250	0.270	7.93%	-6.49%	21.58%
12	0.400	0.432	8.05%	-2.90%	13.54%
18	0.500	0.539	7.74%	-7.98%	38.95%
25	0.581	0.573	-1.51%	-2.72%	9.32%
36	0.667	0.654	-1.83%	-3.74%	0.77%

Probabilities after block 14 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.056	6.30%	-5.53%	12.90%
6	0.250	0.287	14.76%	-10.48%	20.50%
12	0.400	0.438	9.55%	-3.94%	10.53%
18	0.500	0.546	9.18%	-8.66%	34.66%
25	0.581	0.583	0.31%	-3.87%	6.85%
36	0.667	0.661	-0.92%	-4.34%	4.96%

Probabilities after block 17 and 19 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.90%	-4.00%	21.11%
6	0.250	0.269	7.79%	-3.97%	22.74%
12	0.400	0.432	8.08%	-2.85%	13.39%
18	0.500	0.540	7.92%	-7.62%	39.15%
25	0.581	0.573	-1.50%	-2.18%	9.29%
36	0.667	0.655	-1.82%	-3.67%	0.74%

Table 4.5. Continued.

Probabilities after block 3 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.44%	-7.98%	11.94%
6	0.250	0.260	4.18%	-0.81%	1.09%
12	0.400	0.440	9.97%	-13.86%	21.10%
18	0.500	0.506	1.16%	-2.70%	3.85%
25	0.581	0.569	-2.08%	-4.48%	6.53%
36	0.667	0.659	-1.16%	-0.19%	0.26%

Probabilities after block 5 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.30%	-9.93%	24.43%
6	0.250	0.253	1.33%	-11.24%	5.95%
12	0.400	0.434	8.56%	-5.06%	11.09%
18	0.500	0.516	3.17%	-4.27%	10.34%
25	0.581	0.580	-0.28%	-4.84%	11.60%
36	0.667	0.657	-1.43%	-1.38%	0.45%

Probabilities after block 8 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.053	1.61%	-4.18%	2.79%
6	0.250	0.261	4.31%	-1.98%	2.69%
12	0.400	0.425	6.33%	-3.72%	4.12%
18	0.500	0.504	0.89%	-1.68%	2.38%
25	0.581	0.564	-3.07%	-0.43%	0.43%
36	0.667	0.658	-1.26%	-0.65%	0.46%

Table 4.5. Continued.

Probabilities after block 10 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.96%	-0.51%	1.32%
6	0.250	0.262	5.00%	-1.23%	4.02%
12	0.400	0.426	6.47%	-1.66%	2.40%
18	0.500	0.500	-0.07%	-2.46%	1.00%
25	0.581	0.570	-2.01%	-1.51%	4.38%
36	0.667	0.661	-0.89%	-0.43%	0.87%

Probabilities after block 12 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.19%	-3.05%	2.75%
6	0.250	0.260	3.86%	-2.61%	2.73%
12	0.400	0.423	5.63%	-0.90%	1.07%
18	0.500	0.503	0.53%	-0.31%	0.18%
25	0.581	0.563	-3.14%	-0.64%	0.65%
36	0.667	0.659	-1.17%	-0.20%	0.20%

Probabilities after block 14 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.38%	-1.98%	3.70%
6	0.250	0.276	10.58%	-7.01%	24.62%
12	0.400	0.428	6.94%	-2.02%	4.78%
18	0.500	0.506	1.12%	-0.90%	2.38%
25	0.581	0.574	-1.35%	-2.02%	7.09%
36	0.667	0.666	-0.12%	-1.23%	4.10%

Table 4.5. Continued.

Probabilities after block 17 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.31%	-0.94%	1.03%
6	0.250	0.260	3.84%	-0.62%	0.51%
12	0.400	0.422	5.58%	-1.24%	1.21%
18	0.500	0.503	0.53%	-0.51%	0.53%
25	0.581	0.563	-3.13%	-0.22%	0.18%
36	0.667	0.659	-1.17%	-0.13%	0.15%

Probabilities after block 19 and 24 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.33%	-3.53%	10.54%
6	0.250	0.272	8.99%	-5.03%	19.02%
12	0.400	0.435	8.67%	-3.79%	11.41%
18	0.500	0.549	9.74%	-9.16%	34.22%
25	0.581	0.575	-1.12%	-2.37%	8.29%
36	0.667	0.654	-1.96%	-3.32%	0.96%

Probabilities after block 3 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.42%	-9.63%	11.97%
6	0.250	0.261	4.21%	-0.95%	1.06%
12	0.400	0.440	9.95%	-13.85%	21.12%
18	0.500	0.506	1.15%	-3.61%	3.87%
25	0.581	0.569	-2.10%	-5.30%	6.55%
36	0.667	0.659	-1.18%	-0.18%	0.22%

Table 4.5. Continued.

Probabilities after block 5 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.63%	-11.53%	24.66%
6	0.250	0.253	1.33%	-11.27%	6.14%
12	0.400	0.434	8.57%	-5.03%	11.04%
18	0.500	0.516	3.15%	-4.73%	10.38%
25	0.581	0.580	-0.28%	-5.40%	11.60%
36	0.667	0.657	-1.44%	-1.37%	0.55%

Probabilities after block 8 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.053	1.61%	-4.18%	4.01%
6	0.250	0.261	4.31%	-2.54%	2.69%
12	0.400	0.425	6.30%	-4.03%	4.15%
18	0.500	0.504	0.88%	-2.67%	2.39%
25	0.581	0.564	-3.05%	-1.12%	0.62%
36	0.667	0.658	-1.27%	-0.63%	0.46%

Probabilities after block 10 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.75%	-1.77%	1.53%
6	0.250	0.263	5.00%	-1.69%	4.02%
12	0.400	0.426	6.38%	-1.78%	2.48%
18	0.500	0.500	-0.07%	-2.45%	1.62%
25	0.581	0.570	-2.02%	-2.04%	4.39%
36	0.667	0.661	-0.90%	-0.39%	0.87%

Table 4.5. Continued.

Probabilities after block 12 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.25%	-3.99%	3.70%
6	0.250	0.260	3.85%	-2.85%	2.83%
12	0.400	0.422	5.57%	-1.02%	0.84%
18	0.500	0.503	0.51%	-0.85%	0.86%
25	0.581	0.564	-3.08%	-1.17%	1.08%
36	0.667	0.659	-1.19%	-0.19%	0.19%

Probabilities after block 14 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.37%	-3.64%	4.00%
6	0.250	0.277	10.68%	-7.38%	25.54%
12	0.400	0.428	6.94%	-2.29%	4.76%
18	0.500	0.505	1.09%	-1.47%	2.41%
25	0.581	0.574	-1.34%	-2.45%	7.14%
36	0.667	0.666	-0.15%	-1.25%	4.13%

Probabilities after block 17 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.36%	-1.51%	1.44%
6	0.250	0.260	3.81%	-0.88%	0.70%
12	0.400	0.422	5.60%	-1.24%	1.17%
18	0.500	0.502	0.49%	-1.28%	1.22%
25	0.581	0.563	-3.09%	-0.55%	0.69%
36	0.667	0.659	-1.20%	-0.08%	0.10%

Table 4.5. Continued.

Probabilities after block 19 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.056	5.70%	-5.35%	13.68%
6	0.250	0.272	8.88%	-5.28%	18.83%
12	0.400	0.435	8.67%	-3.85%	11.50%
18	0.500	0.548	9.69%	-9.61%	34.07%
25	0.581	0.575	-1.12%	-2.69%	8.15%
36	0.667	0.653	-1.98%	-3.27%	0.92%

Probabilities after block 24 and 27 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.63%	-1.32%	1.27%
6	0.250	0.260	3.85%	-0.61%	0.69%
12	0.400	0.422	5.55%	-1.38%	1.09%
18	0.500	0.502	0.48%	-0.80%	0.76%
25	0.581	0.563	-3.09%	-0.50%	0.67%
36	0.667	0.659	-1.17%	-0.11%	0.14%

Probabilities after block 3 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	4.24%	-11.72%	12.15%
6	0.250	0.261	4.23%	-1.77%	1.32%
12	0.400	0.440	9.95%	-13.39%	21.12%
18	0.500	0.506	1.16%	-3.44%	3.85%
25	0.581	0.569	-2.08%	-4.99%	6.52%
36	0.667	0.659	-1.19%	-0.23%	0.27%

Table 4.5. Continued.

Probabilities after block 5 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.057	8.53%	-12.90%	24.69%
6	0.250	0.253	1.24%	-11.20%	6.58%
12	0.400	0.434	8.60%	-4.75%	11.03%
18	0.500	0.516	3.16%	-4.77%	10.36%
25	0.581	0.580	-0.27%	-5.16%	11.58%
36	0.667	0.657	-1.42%	-1.39%	0.51%

Probabilities after block 8 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.053	1.62%	-4.20%	7.40%
6	0.250	0.261	4.35%	-3.00%	2.65%
12	0.400	0.425	6.30%	-3.12%	4.15%
18	0.500	0.504	0.89%	-2.54%	2.38%
25	0.581	0.564	-3.04%	-0.90%	0.44%
36	0.667	0.658	-1.27%	-0.63%	0.59%

Probabilities after block 10 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.85%	-3.14%	2.57%
6	0.250	0.263	5.07%	-1.97%	3.95%
12	0.400	0.426	6.41%	-1.33%	2.45%
18	0.500	0.500	-0.07%	-2.45%	1.51%
25	0.581	0.570	-1.99%	-1.81%	4.35%
36	0.667	0.661	-0.90%	-0.45%	0.87%

Table 4.5. Continued.

Probabilities after block 12 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.31%	-5.08%	5.50%
6	0.250	0.260	3.91%	-3.18%	3.43%
12	0.400	0.422	5.61%	-0.62%	0.69%
18	0.500	0.502	0.49%	-0.78%	0.74%
25	0.581	0.563	-3.10%	-0.85%	1.01%
36	0.667	0.659	-1.20%	-0.20%	0.16%

Probabilities after block 14 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	3.16%	-3.91%	4.76%
6	0.250	0.277	10.66%	-7.52%	24.95%
12	0.400	0.428	6.93%	-1.81%	4.76%
18	0.500	0.506	1.12%	-1.32%	2.38%
25	0.581	0.574	-1.32%	-2.37%	7.08%
36	0.667	0.666	-0.13%	-1.21%	4.11%

Probabilities after block 17 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.054	2.26%	-3.19%	3.07%
6	0.250	0.260	3.88%	-1.37%	1.16%
12	0.400	0.422	5.60%	-0.71%	0.70%
18	0.500	0.503	0.52%	-1.04%	1.05%
25	0.581	0.563	-3.09%	-0.41%	0.49%
36	0.667	0.659	-1.18%	-0.13%	0.15%

Table 4.5. Continued.

Probabilities after block 19 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.34%	-3.42%	10.48%
6	0.250	0.273	9.05%	-4.88%	18.65%
12	0.400	0.435	8.77%	-3.09%	11.30%
18	0.500	0.549	9.83%	-9.10%	33.86%
25	0.581	0.575	-1.18%	-2.27%	7.99%
36	0.667	0.654	-1.97%	-3.25%	0.90%

Probabilities after block 24 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.34%	-3.42%	10.48%
6	0.250	0.273	9.05%	-4.88%	18.65%
12	0.400	0.435	8.77%	-3.09%	11.30%
18	0.500	0.549	9.83%	-9.10%	33.86%
25	0.581	0.575	-1.18%	-2.27%	7.99%
36	0.667	0.654	-1.97%	-3.25%	0.90%

Probabilities after block 27 and 30 are changing.

N	calc	avg	diff	min	max
1	0.053	0.055	5.34%	-3.42%	10.48%
6	0.250	0.273	9.05%	-4.88%	18.65%
12	0.400	0.435	8.77%	-3.09%	11.30%
18	0.500	0.549	9.83%	-9.10%	33.86%
25	0.581	0.575	-1.18%	-2.27%	7.99%
36	0.667	0.654	-1.97%	-3.25%	0.90%

Table 4.5. Continued.

Program name: **expand**

Probabilities after block 3 and 4 are changing.

N	calc	avg	diff	min	max
1	0.087	0.099	13.91%	-36.71%	66.90%
3	0.222	0.213	-4.30%	-13.52%	23.84%
5	0.323	0.309	-4.36%	-17.35%	25.07%
10	0.488	0.494	1.24%	-8.59%	10.89%
15	0.588	0.632	7.52%	-2.88%	2.08%
21	0.667	0.656	-1.66%	-1.82%	0.86%

Probabilities after block 3 and 6 are changing.

N	calc	avg	diff	min	max
1	0.087	0.098	13.25%	-36.35%	49.29%
3	0.222	0.214	-3.68%	-14.08%	14.23%
5	0.323	0.311	-3.45%	-18.13%	22.41%
10	0.488	0.498	2.15%	-4.96%	7.86%
15	0.588	0.632	7.39%	-3.27%	2.20%
21	0.667	0.655	-1.69%	-1.79%	1.13%

Probabilities after block 4 and 6 are changing.

N	calc	avg	diff	min	max
1	0.087	0.095	8.72%	-6.57%	6.23%
3	0.222	0.213	-4.06%	-10.66%	10.44%
5	0.323	0.312	-3.38%	-10.48%	13.11%
10	0.488	0.497	1.87%	-7.33%	10.00%
15	0.588	0.632	7.49%	-1.72%	1.17%
21	0.667	0.659	-1.13%	-0.60%	0.70%

Table 4.5. Continued.

Probabilities after block 3 and 7 are changing.

N	calc	avg	diff	min	max
1	0.087	0.098	13.19%	-36.31%	42.94%
3	0.222	0.214	-3.68%	-14.08%	15.89%
5	0.323	0.311	-3.68%	-17.94%	18.03%
10	0.488	0.497	1.97%	-4.80%	6.92%
15	0.588	0.632	7.40%	-4.10%	2.19%
21	0.667	0.655	-1.69%	-1.79%	0.75%

Probabilities after block 4 and 7 are changing.

N	calc	avg	diff	min	max
1	0.087	0.095	8.84%	-6.67%	9.79%
3	0.222	0.213	-4.10%	-10.62%	12.48%
5	0.323	0.311	-3.48%	-10.39%	11.52%
10	0.488	0.496	1.74%	-7.20%	8.86%
15	0.588	0.632	7.47%	-2.48%	1.19%
21	0.667	0.659	-1.14%	-0.16%	0.24%

Probabilities after block 6 and 7 are changing.

N	calc	avg	diff	min	max
1	0.087	0.095	9.52%	-6.07%	4.43%
3	0.222	0.215	-3.34%	-3.94%	5.18%
5	0.323	0.314	-2.75%	-6.32%	2.25%
10	0.488	0.500	2.52%	-6.66%	2.18%
15	0.588	0.631	7.33%	-1.47%	2.08%
21	0.667	0.659	-1.22%	-0.32%	0.29%

Table 4.5. Continued.

Probabilities after block 3 and 9 are changing.

N	calc	avg	diff	min	max
1	0.087	0.101	16.67%	-38.21%	39.41%
3	0.222	0.230	3.49%	-20.03%	45.57%
5	0.323	0.327	1.23%	-21.91%	32.07%
10	0.488	0.508	4.07%	-6.72%	12.06%
15	0.588	0.635	7.93%	-3.08%	1.99%
21	0.667	0.654	-1.93%	-1.55%	0.95%

Probabilities after block 4 and 9 are changing.

N	calc	avg	diff	min	max
1	0.087	0.099	14.11%	-10.98%	33.99%
3	0.222	0.230	3.39%	-17.09%	44.93%
5	0.323	0.328	1.68%	-14.93%	30.31%
10	0.488	0.506	3.76%	-9.01%	12.41%
15	0.588	0.635	7.95%	-1.59%	1.95%
21	0.667	0.657	-1.38%	-1.42%	0.48%

Probabilities after block 6 and 9 are changing.

N	calc	avg	diff	min	max
1	0.087	0.100	14.99%	-7.36%	32.90%
3	0.222	0.232	4.48%	-8.76%	44.16%
5	0.323	0.332	2.88%	-8.63%	29.01%
10	0.488	0.511	4.82%	-5.20%	11.18%
15	0.588	0.634	7.73%	-0.90%	2.22%
21	0.667	0.657	-1.46%	-1.36%	0.53%

Table 4.5. Continued.

Probabilities after block 7 and 9 are changing.

N	calc	avg	diff	min	max
1	0.087	0.100	14.70%	-7.44%	34.21%
3	0.222	0.232	4.41%	-9.70%	43.91%
5	0.323	0.331	2.54%	-7.27%	29.41%
10	0.488	0.510	4.52%	-4.22%	11.58%
15	0.588	0.634	7.75%	-1.28%	2.20%
21	0.667	0.657	-1.48%	-1.32%	0.33%

Probabilities after block 3 and 14 are changing.

N	calc	avg	diff	min	max
1	0.087	0.099	13.31%	-36.38%	42.23%
3	0.222	0.214	-3.73%	-14.03%	13.24%
5	0.323	0.311	-3.60%	-18.00%	17.94%
10	0.488	0.498	2.06%	-4.88%	4.08%
15	0.588	0.632	7.42%	-2.43%	2.18%
21	0.667	0.655	-1.68%	-1.80%	0.93%

Probabilities after block 4 and 14 are changing.

N	calc	avg	diff	min	max
1	0.087	0.094	8.56%	-6.43%	5.20%
3	0.222	0.213	-4.08%	-10.64%	9.70%
5	0.323	0.311	-3.48%	-10.39%	9.99%
10	0.488	0.496	1.74%	-7.21%	5.93%
15	0.588	0.632	7.48%	-1.05%	1.18%
21	0.667	0.659	-1.14%	-0.34%	0.24%

Table 4.5. Continued.

Probabilities after block 6 and 14 are changing.

N	calc	avg	diff	min	max
1	0.087	0.095	9.74%	-1.16%	0.86%
3	0.222	0.215	-3.28%	-1.00%	0.84%
5	0.323	0.314	-2.59%	-1.91%	3.82%
10	0.488	0.501	2.61%	-1.88%	2.42%
15	0.588	0.631	7.34%	-0.56%	0.41%
21	0.667	0.659	-1.22%	-0.30%	0.57%

Probabilities after block 7 and 14 are changing.

N	calc	avg	diff	min	max
1	0.087	0.095	9.64%	-3.36%	3.10%
3	0.222	0.215	-3.40%	-2.40%	2.59%
5	0.323	0.314	-2.77%	-2.92%	3.02%
10	0.488	0.500	2.43%	-2.25%	2.22%
15	0.588	0.631	7.35%	-1.07%	1.00%
21	0.667	0.658	-1.23%	-0.21%	0.22%

Probabilities after block 9 and 14 are changing.

N	calc	avg	diff	min	max
1	0.087	0.101	15.96%	-7.37%	21.49%
3	0.222	0.237	6.50%	-10.18%	37.24%
5	0.323	0.336	4.06%	-8.16%	26.03%
10	0.488	0.513	5.25%	-3.20%	10.37%
15	0.588	0.635	7.90%	-0.70%	2.00%
21	0.667	0.657	-1.52%	-1.25%	0.60%

Bibliography

[1] A.T. Acree. "On Mutation". *PhD thesis*. Department of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1980.

[2] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Mutation Analysis." *Technical Report GIT-ICS-79/08*. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, September 1979.

[3] H. Agrawal, R.A. DeMillo, B. Hathaway, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, and E.H. Spafford. "Design of Mutant Operators for the C Programming Language." *Technical Report SERC-TR-41-P*. Software Engineering Research Center, Purdue University, West Lafayette, IN, 1989.

[4] A. Aho, R. Sethi, and J. Ullman. "Compilers: Principles, Techniques, and Tools." Addison-Wesley, Reading, MA, 1986.

[5] D. Baldwin and F.G. Sayward. "Heuristics for Determining Equivalence of Program Mutations." *Technical Report 161*, Department of Computer Science, Yale University, New Haven, Conn, 1979.

[6] F. Briggs, K.S. Fu, K. Hwang, and J. Patel. "PM4- A Reconfigurable Multimicroprocessor System for Pattern Recognition and Image Processing." In *Proc. AFIPS 1979 National Computer Conference* , v.48. June 1979, pp.255-265.

[7] T.A. Budd. "Mutation Analysis of Program Test Data." *PhD Thesis,* Yale University, New Haven, Conn, 1980.

[8] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of

Programs." In *Proc. ACM Symposium on Principles of Programming Languages*, Jan. 1980, pp.220-233.

[9] T.A. Budd, R.A. DeMillo, F.G. Sayward, and R.J. Lipton. "The Design of a Prototype Mutation Analysis System for Program Testing." In *Proc. of the 1978 Conference on Information Science and Systems*, 1978, pp.623-627.

[10] T.A. Budd and R.J. Lipton. "Mutation Analysis of Decision Table Programs." In *Proc. of the 1978 Conference on Information Science and Systems*, The Johns Hopkins University, 1978, pp.346-349.

[11] B. Choi. "Software Testing Using High Performance Computers." *PhD Thesis*, Purdue University, West Lafayette, IN, 1990.

[12] B. Choi, R.A. DeMillo, E.W. Krauser, A.P. Mathur, R.J. Martin, A.J. Offutt, H. Pan, and E.H. Spafford. "The Mothra Toolset." In *Proc. of the Hawaii International Conference on System Sciences*, January 1989.

[13] B. Choi, A.P. Mathur, B. Paterson. "Architecture of PMothra: A Tool for Mutation-Based Testing on the Hypercube." *Technical Report SERC-TR-45-P*, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1989.

[14] B. Choi and A.P. Mathur. "Experience with PMothra: A Tool for Mutant Based Testing on a Hypercube." In *Distributed and Multiprocessor Systems Workshop*, October 1989, pp.237-254.

[15] B. Choi, A.P. Mathur, B. Paterson. "PMothra - Scheduling Mutants for Execution on a Hypercube." In *Proc. ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification*, December 1989, pp.58-65.

[16] W.M. Craft. "Detecting Equivalent Mutants Using Compiler Optimization Techniques." *Master's Thesis,* Department of Computer Science, Clemson University, Clemson, SC, 1987.

[17] R.A. DeMillo, G.S. Guindi, K.N. King, and W.M. McKracken. "An Overview of the Mothra Testing Environment." *Technical Report SERC-TR-3-P,* Software Engineering Research Center, Purdue University, West Lafayette, IN, 1987.

[18] R.A. DeMillo, G.S. Guindi, K.N. King, W.M. McKracken, and A.J. Offutt. "An Extended Overview of the Mothra Software Testing Environment." In *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, ACM, July 1988, pp.142-151.

[19] R.A. DeMillo, D.E. Hocking, and M.J. Merrit. "A Comparison of Some Reliable Test Data Generation Procedures." *Technical Report GIT-ICS-81/08*, Georgia Institute of Technology, Atlanta, GA, 1981.

[20] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. "Using the Hypercube for Reliable Testing of Large Software." *Technical Report SERC-TR-24-P,* Software Engineering Research Center, Purdue University, West Lafayette, IN, 1988.

[21] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. "Compiler-Integrated Program Mutation." In *Fifteenth Annual International Computer Software and Applications Conference,* September 1991, pp.351-356.

[22] R.A. DeMillo, E.W. Krauser, and A.P. Mathur. "Compiler Support for Program Testing on MIMD Architectures." In *Pacific Northwest Software Quality Conference*, 1991, pp.221-234.

[23] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer* 11(4), April 1978, pp.34-43.

[24] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Program Mutation: A New Approach to Program Testing." In *Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, Infotech International*, 1979, pp.107-126.

[25] R.A. DeMillo and A.P. Mathur. "On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Producing Software." In *Proc. of the Thirteenth Minnowbrook Workshop on Software Engineering*, July 1990.

[26] V.N. Fleyshgakker and S.N. Weiss. "Software Testing on a Highly Parallel Multiple SIMD Architecture." *Technical Report TR-CS-91-07*, Department of Computer Science, Hunter College, New York, NY, 1991.

[27] V.N. Fleyshgakker and S.N. Weiss. "HiTest: an Architecture for Highly Parallel Software Testing." In *Proc. of the International Conference on Computer Systems and Software Engineering*, May 1992, pp.347-352.

[28] P.G. Frankl and S.N. Weiss. "An Experimental Comparison of Effectiveness of the All-Uses and All-Edges Adequacy Criteria." In *Proc. of the 1991 ACM Symposium on Testing, Analysis, and Verification*, pp. 1-10. October 1991.

[29] M.R. Girgis and M.R. Woodward. "An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis." In *Proc. of the Eighth International Conference on Software Engineering, IEEE Computer Society*, August 1985, pp.313-319.

[30] M.R. Girgis and M.R. Woodward. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria." In *Proc. of the Workshop on Software Testing*, pp. 64-73. ACM, July 1986.

[31] J.S. Gourlay. "A Mathematical Framework for the Investigation of Testing". *IEEE Transactions on Software Engineering*, SE-9 (6), November 1983, pp.686-709.

[32] R.G. Hamlet. "Testing Programs with the Aid of a Compiler." *IEEE Transactions on Software Engineering*, SE-3(4), July 1977, pp.279-290.

[33] W.D. Hillis. "The Connection Machine." MIT Press, Cambridge, MA, 1985.

[34] J.R. Horgan and A.P. Mathur. "Weak Mutation is Probably Strong Mutation." *Technical Report SERC-TR-83-P*, Software Engineering Research Center, Purdue University,West Lafayette, IN, December 1990.

[35] W.E. Howden. "Weak Mutation Testing and Completeness of Test Set." *IEEE Transactions on Software Engineering*, SE-8(4), July 1982, pp.371-379.

[36] K.N. King and A.J. Offutt. "A FORTRAN Language System for Mutation Based Software Testing." *Software Practic and Experience*, 21(7), July 1991, pp.685-718.

[37] E.W. Krauser, A.P. Mathur, V.J. Rego. "High Performance Software Testing on SIMD Machines." *IEEE Transactions on Software Engineering*, SE-17 (5), May 1991, pp.403-423.

[38] E.W. Krauser, A.P. Mathur, V.J. Rego. "High Performance Software Testing on SIMD Machines." *Technical Report SERC-TR-17-P*, Software Engineering Research Center, Purdue University, Westr Lafaette, IN, 1988.

[39] E.W. Krauser, A.P. Mathur, V.J. Rego. "High Performance Testing on SIMD Machines." In *Proc. of the Second Workshop on Software Testing, Verification , and Analysis*, ACM, July 1988, pp.171-177.

[40] R.J. Lipton and F.G. Sayward. "The Status of Research on Program Mutation." In *Digest for the Workshop on Software Testing and Test Documentation*, December 1978, pp.355-373.

[41] A.P. Mathur and E.W. Krauser. "Modeling Mutation on a Vector Processor." *Technical Report GIT-SERC-87/07*, Software Engineering Research Center, Georgia Institute of Technology, Atlanta, GA, 1986.

[42] A.P. Mathur and E.W. Krauser. "Modeling Mutation on a Vector Processor." In *Proc. of the Tenth International Conference on Software Engineering*, April 1988.

[43] A.P. Mathur and E.W. Krauser. "Mutant Unification for Improved Vectorization." *Technical Report SERC-TR-14-88*, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 1988.

[44] B. Marick. "Two Experiments in Software Testing". *Technical Report UIUCDCS-R-90-1644*, Department of Computer Sciece, University of Illinois at Urbana-Champaign, November 1990.

[45] B. Marick. "The Weak Mutation Hypothesis." In *Proc. of Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 190-199. ACM, October 1991.

[46] B. Marick. "GCT User Documentation." Testing Foundations, 809 Balboa, Champaign, IL 61820, February 1993.

[47] L.J. Morell. "A Theory of Fault Based Testing." *IEEE Transactions on Software Engineering*, SE-14(6):868-874, June 1988.

[48] S. Ntafos. "On Required Element Testing." *IEEE Transactions on Software Engineering*, SE-10(6), November 1984, pp. 795-803.

[49] G.J. Nutt. "Microprocessor Implementation of a Parallel Processor." In *Proc. 4th Annual Symposium on Architecture*, March 1977, pp.147-152.

[50] A.J. Offutt. "The Coupling Effect: Fact or Fiction?" In *Proc. ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification*, December 1989, pp.131-140.

[51] A.J. Offutt and S.D. Lee. "How Strong is Weak Mutation?" In *Proc. of Symposium on Testing, Analysis, and Verification (TAV4)*, ACM, October 1991, pp.200-213.

[52] A.J. Offutt, G. Rothermel, and C. Zapf. "An Experimental Evaluation of Selective Mutation." In *Proc. of the 15th International Conference on Software Engineering*, May 1993, pp.100-107.

[53] M.J. Quinn. "Designing Efficient Algorithms for Parallel Computers." McGraw-Hill Book Company, New York, NY.

[54] S. Rapps and E.J. Weyuker. "Selecting Software Test Data Using Data Flow Information." *IEEE Transactions on Software Engineering*, SE-11 (4), April 1985, pp. 367-375.

[55] V.J. Rego and A.P. Mathur. "Exploiting Parallelism Across Program Execution: A Unification Technique and its Analysis." *Technical Report CSD-TR-751-1988*, Department of Computer Science, Purdue University, West Lafaette, IN, 1988.

[56] V.J. Rego and A.P. Mathur. "Exploiting Parallelism Across Program Execution: A Unification Technique and its Analysis." In *Proc. of the International Seminar on Performance of Distributed and Parallel Systems,* December 1988, pp.397-412.

[57] V.J. Rego and A.P. Mathur. "Concurrency Enhancement through Program Unification: A Performance Analysis." *Journal of Parallel and Distributed Computing*, 1990, 8:201-217.

[58] I.J. Riddell, M.A. Hennell, M.R. Woodward, and D. Hedley. "Practical Aspects of Program Mutation. *Technical Report*. Department of Statistics and Computational Mathematics, University of Liverpool, Liverpool, England, 1980.

[59] D.J. Richardson and M.C. Thompson. "The RELAY Model of Error Detection and its Applications." In *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, ACM, July 1988, pp.223-230.

[60] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley, Jr., and S.D. Smith. "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition." *IEEE Transactions on Computers*, C-30(12), Dec. 1981, pp.935-947.

[61] R.H. Untch, A.J. Offutt, and M.J. Harrold. "Mutation Analysis Using Mutant Schemata." In *Proc. of the International Symposium on Software Testing and Analysis*, Cambridge, MA, June 1993.

[62] S.N. Weiss. "Methods of Comparing Test Data Adequacy Criteria." In *Proc. COMPSAC 90*, pp. 1-6, October 1990.

[63] S.N. Weiss and V.N. Fleyshgakker. "Improved Serial Algorithms for Mutation

Analysis." In *Proc. of the International Symposium on Software Testing and Analysis*, Cambridge, MA, pp. 149-158. June 1993.

[64] S.N. Weiss and V.N. Fleyshgakker. "Improved Algorithms for Mutation Analysis and their Complexity." *Technical Report CS-TR-92-02*, Department of Computer Science, Hunter College, New York, NY, 1992.

[65] T.C. Wesselkamper. "Computer Program Schemata and the Processes They Generate." *IEEE Transactions on Software Engineering*, SE-8 (4), July 1982, pp. 412-419.

[66] E.J. Weyuker, S.N. Weiss, and R. Hamlet. "Comparison of Program Testing Strategies." In *Proc. Fourth Symposium on Software Testing, Analysis, and Verification*, pp. 1-10. ACM, October 1991.

[67] P.A. Wilsey, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," *Technical Report TR 133-11-91*, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.

[68] M.R. Woodward and K. Halewood. "From Weak to Strong, Dead or Alive: An Analysis of Some Mutation Testing Issues." In *Proc. of the Second Workshop on Software Testing, Verification, and Analysis*, ACM, July 1988, pp.152-158.

[69] D. Wu. "Syntax Directed and Semantic Aided Mutation." *Technical Report*, Department of Statistics and Computational Mathematics, University of Liverpool, Liverpool, England, March 1987.

[70] D. Wu, M.A. Hennell, D. Hedley, and I.J. Riddell. "A Parallel Method for Software

Quality Control via Program Mutation." in *Proc. of the Second Workshop of Software Testing, Verification, and Analysis*, ACM, July 1988, pp. 159-170.

[71] D. Wu, I.J. Riddell, M.A. Hennell, and D. Hedley. "The Minimum Set of Test Data on Syntax Directed Mutation of Boolean Expressions. *Technical Report*, Department of Statistics and Computational Mathematics, University of Liverpool, Liverpool, England, April 1987.