RL-TR-96-259
Final Technical Report
March 1997

A NEW PARADIGM FOR FAST, INTERACTIVE SIMULATION

University of Massachusetts at Amherst

Christos G. Cassandras and WEI-BO Gong

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-259 has been reviewed and is approved for publication.

APPROVED:

ALEX F. SISTI
Project Engineer

FOR THE COMMANDER:

JOSEPH CAMERA, Technical Director
Intelligence & Reconnaissance Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/IRAE, 32 Hangar, Rome, NY 13441-4114. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1997	Final Jun 94 - Sep 95

4. TITLE AND SUBTITLE

A NEW PARDIGM FOR FAST, INTERACTIVE SIMULATION

5. FUNDING NUMBERS

C - F30602-94-C-0109
PE - 62702F
PR - 4594
TA - 15
WU - L5

6. AUTHOR(S)

Christos G. Cassandras and WEI-BO Gong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Massachusetts at Amherst
Department of Electrical and Computer Engineering
Amherst MA 01003

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/IRAE
32 Hangar Road
Rome NY 13441-4114

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-259

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Alex F. Sisti/IRAE/(315) 330-4518

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of this effort has been to investigate and develop specific algorithms and demonstrations establishing the feasibility and value of novel techniques for (a) sensitivity estimation, (b) response surface generation, (c) parallel simulation, (d) ordinal optimization, and (e) interfaces in hierarchical simulation, to be used towards the creation of a fast, interactive simulation environment for C3I systems. Towards this objective, several specific tasks were performed intended to investigate new approaches and to demonstrate their capability to (a) provide substantial improvement in simulation time, (b) preserve statistics in hierarchical simulation models, and (c) facilitate design and decision making processes based on simulation.

14. SUBJECT TERMS

Concurrent Simulation, Real-Time Simulation, Hierarchical Simulation, Pertubation Analysis, Ordinal Optimization, Statistical Fidelity

15. NUMBER OF PAGES

138

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

TABLE OF CONTENTS

1. INTRODUCTION.

This report summarizes the work we have performed for the project entitled "A New Paradigm for Fast, Interactive Simulation". The objective of this effort has been to investigate and develop specific algorithms and demonstrations establishing the feasibility and value of novel techniques for (a) sensitivity estimation, (b) response surface generation, (c) parallel simulation, (d) ordinal optimization, and (e) interfaces in hierarchical simulation, to be used towards the creation of a fast, interactive simulation environment for C^3I systems. Towards this objective, several specific tasks were performed intended to investigate new approaches and to demonstrate their capability to (a) provide substantial improvement in simulation time, (b) preserve statistics in hierarchical simulation models, and (c) facilitate design and decision making processes based on simulation.

In what follows, so as to place our work in the proper context, we first identify some of the major challenges faced by simulation technology today and the approaches we are proposing to follow (Section 1.1). We then describe the organization of this report (Section 1.2).

1.1. Issues in Discrete Event Simulation.

Simulation is widely recognized as one of the most versatile and general-purpose tools available today for modeling complex processes and solving problems in design, performance evaluation, decision making, and planning. This includes C^3I environments, where most problems confronted by designers and decision makers are of such complexity that their analysis and solution far surpass the scope of available analytical and numerical methods; this leaves simulation as the only alternative of "universal" applicability.

The importance of discrete event simulation has given rise to a number of commercially available software packages (e.g., SIMAN, SLAM, SIMULA, SIMSCRIPT, MODSIM, EXTEND) whose applicability ranges from very generic to highly specialized. However, the use of typical simulation software is limited by factors such as the following: (a) One must have thorough knowledge of the specific tool at a detailed technical level before attempting to use it in a modeling effort. (b) One must be an experienced programmer, in addition to a decision maker. (c) In order to make decisions based on simulation, one usually needs to run a large number of simulations and then carefully manage all output data collected on a case-by-case basis. (d) The field of simulation was developed primarily as a special branch of statistics involving dynamical phenomena. Manual handling and analysis of input/output data is still the norm, while design of interfaces, componentware interoperability, intelligent and automated analysis of output have been neglected. For example, the practice of Object Oriented Programming (OOP), with few exceptions,

is still nascent in simulation languages despite the fact that the OOP idea actually originated in simulation. In addition, hardware advances, such as massively parallel computers and workstation networking, are only beginning to be noticed in simulation theory and practice. (e) The ultimate purpose of simulation is often system performance evaluation and optimization. However, simulation is notoriously computer time-consuming when it comes to parametric studies of system performance. Unless substantial speedup of the performance evaluation process can be achieved, systematic performance studies of most real-world problems are beyond reach, even with supercomputers.

With this brief discussion in mind, we identify below four issues that we believe constitute the major challenges faced by simulation technology today, and introduce some key ideas which have been the subject of further study in this project.

1. *"What if" capability.*

One of the main goals of simulation is to set up a model and then ask many (say N) "what if" questions. This normally requires a nominal simulation run and then N additional runs, one for each "what if". Given that in many cases even a single simulation run can be an extremely time-consuming process, this basic "what if" capability is severely limited and often infeasible. There is, therefore, a need for developing "intelligent" schemes through which "what if" information can be extracted from a small number of simulation runs. In many cases, it is indeed possible to answer multiple "what if" questions from data obtained from *the nominal simulation run alone*. The basis of this capability lies on the theory of Perturbation Analysis (PA) developed over the past several years, now well-documented in the technical literature, including several books [1]-[3]. Traditional PA theory has focused on "small" changes in continuous parameters of a model and is frequently aimed at estimating sensitivities (i.e., derivatives) of particular performance measures with respect to certain model parameters. This project has been aimed at building on principles of PA theory in order to develop methodologies and specific algorithms with the capability to obtain answers to multiple "what if" questions from a single simulation run. This has given rise to *concurrent simulation* techniques which will be further discussed in this report.

2. *Optimization.*

Often, the purpose of simulation is to compare many alternatives in order to identify the optimal one. In evaluating multiple designs or decisions so as to select the "best" one, traditional techniques rely on computing the *cardinal* values of a performance (or cost) objective function over all possible alternatives. This process is clearly infeasible when analytical expressions of the objective function are unavailable or too hard to estimate (usually requiring enormous numbers of long simulation runs). In contrast, *ordinal* optimization is driven by the *relative order* of estimates

of the objective function -- not their absolute values. The advantage here is that we can exploit inherent robustness properties of these order statistics with respect to substantial estimation noise. In simple terms: why waste time to get "good" performance estimates, when relatively "poor" but quickly obtained estimates can be provably adequate to order these performance estimates? [4] The result is a new design evaluation and optimization framework. Combined with concurrent simulation capabilities (described above), we believe this framework explicitly tackles the problem of complexity by rapidly narrowing down a potentially very large set of alternative designs which are candidates for the global optimum.

3. *Hierarchical decomposition.*

One way to reduce complexity is through hierarchical decomposition of a simulation model. The challenge here is to do it without sacrificing accuracy. By "accuracy" we mean that the statistical information generated at the low level, high resolution simulation model should be preserved accurately at the higher level models. Our emphasis is in the area of battle simulations such as those discussed in [26]-[29]. Concurrent and parallel simulation is one way to break down the complexity of simulating a large size battle simulation model. However we have found that it is quite difficult to completely parallelize a battle simulation model which usually has many parts that are intrinsically sequential. One such example is the sorting of the importance of the weapons. We have therefore concentrated on the preservation of the stochastic fidelity in hierarchical battle simulation models. We have worked with a concrete battle simulation model [29] and analyzed various approaches for the preservation of stochastic fidelity for this specific model. We believe that the main ideas and results presented in this report can be useful for other battle simulation models as well.

4. *Interactive use of simulation.*

Simulation remains a tool which, as a rule, one cannot use in interactive mode. A simulation run is typically set up and, while it is running (possibly for a long time), a user cannot conveniently observe its progress or alter the model. Clearly, this imposes serious limitations to the usefulness of simulation. For example, a simple input error may not be detected until after the completion of a simulation run, rendering the output useless and wasting a significant amount of computer time. To transform simulation into an interactive design and decision making tool requires a combination of new analysis techniques such as the ones discussed above with new software capabilities, including OOP and Graphical User Interfaces (GUI). Although the latter are not within the scope of this project, we have made a significant effort to develop software demonstrations of our new approaches in the context of OOP and GUI-based software.

1.2. Report Organization.

The contents of this report may be outlined as follows.

- **Section 2:** The basic structure of a discrete event simulator is presented to provide the framework for explaining the concurrent simulation algorithms we have developed and contrast them to "brute force" simulation. Two general approaches are described: the *Standard Clock* (SC) approach and *Augmented System Analysis* (ASA). The concept of "speedup" is also introduced in order to provide a clear quantitative measure of the improvement provided by these concurrent simulation techniques.

- **Section 3:** A testbed model is introduced in order to illustrate the use of the concurrent simulation techniques described in Section 2. A detailed comparative study between "brute force" simulation and the SC and ASA techniques is then provided, focusing on the amount of speedup generated by these techniques in answering multiple "what if" questions.

- **Section 4:** Optimization schemes making use of concurrent simulation are presented. Using the testbed model of Section 3, two such schemes, the *Stochastic Comparison* and *Stochastic Descent* algorithms, are compared.

- **Section 5:** A concrete hierarchical battle simulation model is introduced and the stochastic fidelity preservation issue is discussed in detail. We recommend a scenario grouping approach to tackle this issue and also present an innovative scheme to implement the recommended algorithms.

- **Section 6:** A brief introduction to the use of rational approximation techniques is presented and applied to a class of computationally complex performance analysis problems in stochastic discrete event systems such as computer systems and communication networks.

- **Section 7:** We present the main conclusions of our study, including lessons learned and recommendations. We also outline our ongoing work and some future research directions.

2. CONCURRENT SIMULATION.

As already mentioned in Section 1, a major objective of this project is motivated by the time consuming nature of system performance exploration through simulation: to obtain answers to N "what if" questions, $(N+1)$ simulations are needed. Therefore, our goal is the following: *From a **single** simulation, obtain answers to all N "what if" questions simultaneously.* The basic framework for dealing with this problem is presented in Section 2.1. In Section 2.2, we outline the functionality of a typical discrete event simulator, in order to subsequently contrast it to the techniques we develop. In Sections 2.3 we briefly review the "brute force" method for solving the problem above, and then introduce concurrent simulation techniques in Section 2.4. Finally, in Section 2.5 we describe means for quantitatively evaluating the efficiency of concurrent simulation methods compared to "brute force" simulation. In doing so, we introduce the "speedup factor" as a basic measure of efficiency and identify some fundamental bounds this measure satisfies.

2.1. The Basic Framework.

The *concurrent* simulation methodology we have developed is based on using all possible data involved in one simulation to drive N simulations concurrently (in parallel). This is illustrated in **Figure 1**, where it is assumed that a system model is given and a set of parameters (actions, designs) $\Theta = \{\theta_1,\ldots,\theta_m\}$ is specified. The problem then is to observe the system under θ_1 and from the observations made to estimate/predict/infer/learn the performance under all $\theta_2,\ldots,\theta_m$ on line and in parallel. This is also referred to as the *sample path constructability problem*, since it requires the construction of m-1 sample paths from data extracted from the nominal sample path observed under θ_1.

In discussing this approach, it is important to distinguish between the following different views:

1. BRUTE-FORCE REPETITIVE SIMULATION:
   simulate a system N times sequentially
2. PARALLEL SIMULATION OF **ONE** SYSTEM:
   distribute the simulation code over N processors or a network of N workstations
3. PARALLEL SIMULATION OF **MANY** SYSTEMS WITH THE SAME STRUCTURE, BUT DIFFERENT SETTINGS:
   share simulation data to perform N simulation experiments concurrently, either on a single processor or over N processors

To clearly differentiate between the second and third viewpoints above, we will reserve the term

"parallel (or distributed) simulation" for the case where N processors are used to simulate one system (2 above). We will use the term "concurrent simulation" for the case where N simulation experiments are performed simultaneously (3 above).

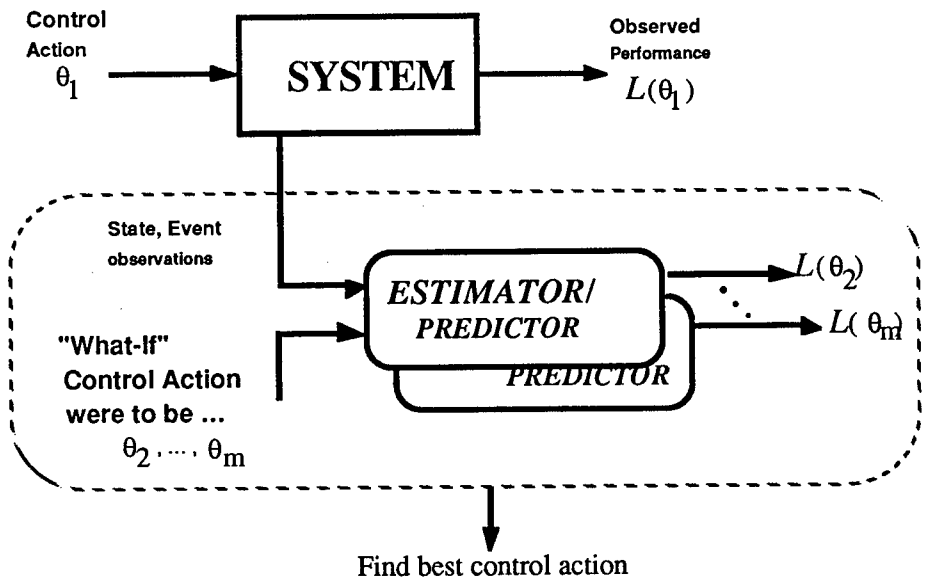


Figure 1: Concurrent simulation framework

We will also sometimes refer to concurrent simulation as "multithread simulation". This captures the idea that in this approach multiple concurrent threads unfold, all driven by common data. The main steps in this approach are as follows:

1. Set up a single simulation
2. Generate multiple simultaneous *simulation threads*
3. Each thread corresponds to a different system (same *structure*, but different *design*)

When implementing such an approach on *sequential* computers, processing is serial, but all simulation threads unfold in parallel. The main benefit here lies in sharing of data and of some data structure maintenance. On *parallel* computers, on the other hand, one processor is allocated to each simulation thread. In this case, the additional benefit is that all system state updates are also parallelized (e.g., see [14]-[15]). In this project, we have developed concurrent simulation techniques which may be implemented on either sequential or parallel processing computing environments. However, all numerical testing is limited to concurrent simulation on sequential computers.

2.2. Functionality of a Discrete Event Simulator.

We begin by outlining the main components of discrete event simulation in order to contrast various approaches for parallelizing the generation of simulation runs over different parameter settings.

1. ***Clock Structure*** (or ***Clock Mechanism***)

Every discrete event model to be simulated involves a set E of events (see also [5]-[7]). For example, in a communication system, "arrival of message at point A to be transmitted to point B", "message successfully received at point B", and "equipment at point A has failed" are such typical events. The evolution of the system's state is entirely driven by the occurrence of such events over time. We associate with every event $i \in E$ a clock sequence $\mathbf{v}_i = \{v_{i,1}, v_{i,2}, \ldots\}$, where, $v_{i,j}$ is the jth *lifetime* of event i. These lifetimes are the random variates generated through a computer's random number generation mechanism. For example, in a communication system, lifetimes for event "message successfully received at point B" correspond to random times required for the system to transmit a message from point A (where it originated) to point B. In actual simulation software, the sequence $\mathbf{v}_i$ is not predetermined; a new lifetime is generated as needed based on the concept of feasible events explained next.

2. ***Feasible Event Set*** for every state and ***Event Calendar***

Let x denote the state of the system being simulated. At any state x there is a set $\Gamma(x)$, a subset of E, which represents all those events feasible at that state. For example, if the state of a communication system is such that the transmitter at point A is currently idle, then event "message successfully received at point B" is obviously not feasible. In typical simulation software, one uses the set $\Gamma(x)$ to create a data structure known as the *Event Calendar*. Specifically, when an event i becomes feasible at some state x and is assigned some lifetime v_i, its scheduled time to occur is given by

$$t_i = t + v_i \qquad (1)$$

where t is the current simulation time. The Event Calendar is a list of all events in $\Gamma(x)$ accompanied by their corresponding scheduled times t_i ordered on a smallest scheduled event time first basis. In other words, the first item in this list is always the next event scheduled to occur.

3. ***Triggering Event***

Given the Event Calendar at state x, the *triggering event* e' is the event which occurs next at that state, i.e., the event with the smallest scheduled time value:

$$e' = arg \min_{i \in \Gamma(x)} \{t_i\} \quad (2)$$

When the triggering event is determined, the simulation clock is simply updated so that $t := t_i$, where $i = e'$.

4. ***State Transition Mechanism***

When a new event is selected from the Event Calendar, a state transition takes place. Mathematically, the state transition mechanism is simply expressed through a state transition function that maps the current state x and triggering event e' into a new state x':

$$x' = f(x, e') \quad (3)$$

In practice, the actual state transition mechanism corresponds to all operational rules characterizing the model being simulated. For example, in a communication system the next state may depend on complex protocols for selecting the next message type to be processed.

In summary, the main functions of a discrete event simulator (other than data collection and report generation which are largely application-dependent) are:

(F1) Maintain an Event Calendar which is updated through the triggering event mechanism

(F2) Update the simulation clock

(F3) Update the state after every event occurrence

(F4) Generate random variates corresponding to various event lifetimes.

In parallelizing simulation runs, we wish to perform a number of simulation experiments pertaining to a particular system with each experiment performed under different parameter settings. Sometimes the changes in these parameter settings are very simple, such as "reduce the average transmission time by 10%", but they can also be quite complex such as replacing a First-Come-First-Serve scheduling discipline by a new rule such as Shortest-Time-to-Deadline-First. In what follows, we describe three approaches we have studied in this project. The first is the obvious "brute force" approach whereby each simulation experiment is separately performed without any effort to share data among experiments. The remaining two are two specific concurrent simulation schemes.

2.3. Trivial Parallelization (Brute-Force) Method.

This is the obvious approach to carrying out a parametric design study: for each desired parameter setting (or alternative design), a separate simulation experiment is performed. In the case of a sequential computer, one has to serially perform the experiments. In the case of a Multiple-

Instruction-Multiple-Data (MIMD) parallel computer, one may be able to utilize each processor for each simulation run without any interaction or coordination among processors.

2.4. Concurrent Simulation Methods.

In order to explain the key ideas involved in the two methods described in Sections 2.4.1 and 2.4.2, we begin by returning to **Figure 1**, where the basic concurrent simulation framework is set. In seeking conditions under which the sample path constructability problem depicted in this figure can be solved, we find that there are two aspects of the problem that need to be addressed. First, the structure of the system itself may or may not be conducive to sample path constructability. Second, the stochastic characteristics of the lifetime sequences driving the model are critical in ensuring that the sample path we construct is indeed a valid one. A formal "constructability condition" that reflects these two aspects is described in [3]. We will omit here technical details (which can be found in [8]-[10]) and limit ourselves to an intuitive explanation of this condition.

Let Σ_1 denote the simulation model already available (i.e., the sample path generated under the parameter setting θ_1 in **Figure 1**) and $\Sigma_2,\ldots,\Sigma_N$ the additional simulation models to be run. Let $\{x_k^j\}$ be the state sequence corresponding to Σ_j. Now suppose that state x_k^1 is observed after the kth state transition in Σ_1. The corresponding feasible event set is $\Gamma(x_k^1)$. At the same time, suppose the state of some Σ_j, $j = 2,\ldots,N$, is x_k^j, with corresponding feasible event set $\Gamma(x_k^j)$. Now, if $\Gamma(x_k^j) \subseteq \Gamma(x_k^1)$, then all events required in Σ_j to determine the next state transition are "observable" in the observed set $\Gamma(x_k^1)$. This lends its name to the term "observability condition" for

$$\Gamma(x_k^j) \subseteq \Gamma(x_k^1) \qquad \text{for all } k = 1,2,\ldots \tag{4}$$

Unfortunately, this simple structural condition is not sufficient to guarantee that the sample path constructability problem is solved. In addition to it, one must ensure that the residual lifetimes of all events in $\Gamma(x_k^j)$ and $\Gamma(x_k^1)$ have the same probability distribution. Thus, in general, the constructability condition is not easy to satisfy for most systems of practical interest. The issue then becomes the development of schemes whereby *this condition can be enforced at some cost.* This leads to two different ways to solve the sample path constructability problem, which are described next.

2.4.1. Standard Clock (SC) Method.

This approach was first proposed in [11]. The main idea is to try to bypass the observability problem when (4) is not satisfied by creating a fictitious system, denoted by Σ_0, in which all

events are always feasible, i.e., by setting $\Gamma(x) = E$ for all possible states x. Then, by construction, (4) is always satisfied, since $\Gamma(x_k^j) \subseteq E$ by definition. Whenever an event is observed in the simulated system Σ_0 which is actually not feasible in the real system Σ_1, it is simply ignored. This leads to a stochastically correct sample path of Σ_1 as long as all event lifetimes are exponentially distributed. This obviously imposes a constraint on the types of systems for which this technique can be applied; in particular, it assumes that the system being simulated is modeled as a Markov chain. There are, however, a number of extensions and approximations one can use to overcome this problem (see [12]-[13]). We have obtained some very good results along these lines, but have not spent a significant amount of time in this particular direction.

Let us now briefly describe the SC method. Assuming that every event $i \in E$ has exponentially distributed lifetimes, it suffices to associate with it a single parameter λ_i, the rate characterizing this distribution. It is then well known (e.g., see [3],[16]) that when the system enters state x, the distribution of the ensuing interevent time is exponential with rate given by:

$$\Lambda(x) = \sum_{i \in \Gamma(x)} \lambda_i \tag{5}$$

Moreover, when the state is x, the distribution of the triggering event is given by:

$$p(i, x) = \frac{\lambda_i}{\Lambda(x)}, \qquad i \in \Gamma(x) \tag{6}$$

where $p(i, x)$ is the probability that the triggering event is i when the state is x. Another well known property of Markov chains is also exploited, known as uniformization: regardless of the actual event rate $\Lambda(x)$ for any state x in (5), we use a uniform rate given by

$$\Lambda = \sum_{i \in E} \lambda_i \tag{7}$$

and treat all events which are not feasible in x and contributing the event rate $\Lambda - \Lambda(x)$ as "fictitious events". If such an event is observed in the simulation when the state is x, it is simply ignored. Note that the mechanism through which the triggering event at any state is determined now becomes independent of the state. Specifically, in (6) we can now set $\Lambda(x) = \Lambda$ for all x. Therefore, the triggering event distribution at any state is simply $p_i = \lambda_i/\Lambda$.

The algorithm through which we may obtain N simulation runs in parallel using the SC method is presented below. The state of the jth simulation, $j = 1,\ldots,N$, is denoted by x^j and the corresponding total event rate (which may differ across experiments) by Λ_j.

Standard Clock Algorithm

Step 0. Initialize: x^j to a desired initial value for all $j = 1,\ldots,N$.

Repeat the following steps for all $j = 1,\ldots,N$:

Step 1. Generate an interevent time V with distribution $1 - e^{-t}$, $t > 0$.

Step 2. Generate an event type e with distribution $p_i = \lambda_i/\Lambda.$, $i \in E$.

Step 3. Check for event feasibility: if $e \in \Gamma(x^j)$, then go to *Step 4*, else skip it.

Step 4. Update state: $(x^j)' = f_j(x^j, e)$.

Step 5. Rescale interevent time: $V^j = V/\Lambda_j$ and return to *Step 1*.

The SC algorithm is also presented in **Figure 2**, in order to contrast it to the conventional simulation approach. Observe that the determination of event times is totally shared by all simulations; in a Single-Instruction-Multiple-Data (SIMD) parallel computer, for example, the generation of interevent times may therefore be performed at the front-end computer. In addition, *Steps 2,3,5* above can be executed by a single instruction. In many cases, the parameter Λ_j is unchanged across simulations, so it is not needed; in this case, the distribution used in *Step 1* is simply $1 - e^{-\Lambda t}$, $t > 0$.

We can now see how concurrency can be exploited in the SC Algorithm:

• *Sequential processing*: In this case, one has to serially perform the experiments, but only the state update function (F3) (i.e., *Step 4* above) needs to be repeated for each of the N simulation runs. All other functions are common to all. The cost for this sharing is contained in *Step 3* above, where one needs to check whether an event generated can indeed be used in the jth simulation (for each $j = 1,\ldots,N$) or should be ignored. A quantifiable measure of this cost is described in Section 2.5. Clearly, however, this cost is generally smaller than that of one complete simulation time T. Thus, as shown in some of our numerical results, the time required to perform N simulations through the SC approach is much smaller than NT.

• *Parallel processing*: Observe that the determination of event times is totally shared by all simulations; in a SIMD parallel computer, the generation of interevent times may therefore be performed at the front-end computer. In addition, *Steps 2-5* above can each be executed by a single program statement. Therefore, referring to the four simulation functions presented earlier, (F2) and (F4) are performed at the front end computer, while (F1) is eliminated. Instead, *Steps 2-3* are performed in parallel through common instructions. Finally, only (F3) remains unique to each of the N simulation runs considered (however, it can be carried out via a common instruction). It should be emphasized that this form of parallelization is drastically different from the approach of distributing simulation code for a single simulation over multiple processors (see also [17] and

selected papers in [18]).

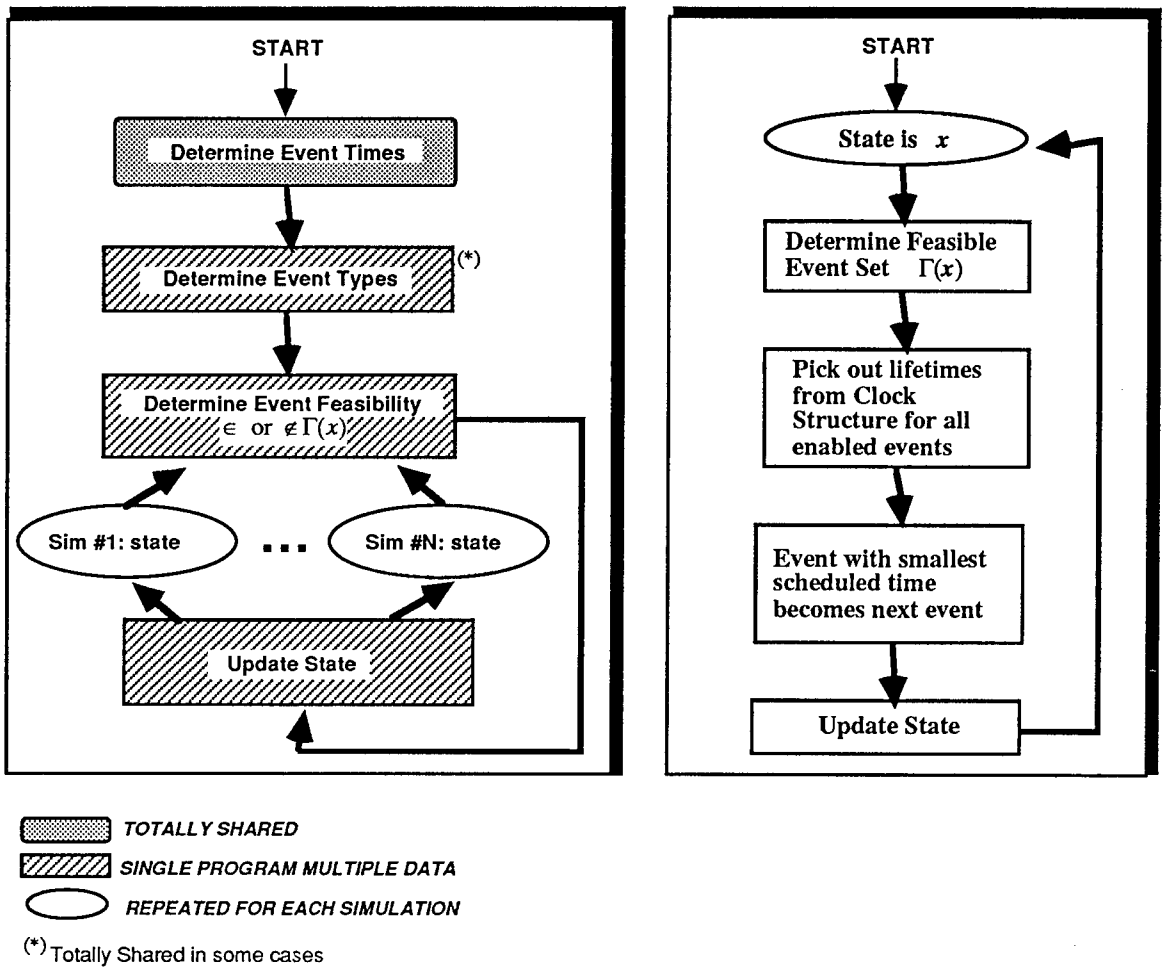


Figure 2. The SC Concurrent Simulation scheme compared to conventional simulation

2.4.2. Augmented System Analysis (ASA).

Unlike the SC approach, in Augmented System Analysis (ASA) the parallel sample path construction process is driven by a specific simulation run already available [3],[8]-[10]. This presents the practical advantage of not having to redesign a simulation software environment to accommodate the SC setup described in the last section. To describe the ASA approach, consider the joint state (x_k^1, x_k^j) following the kth transition at systems Σ_1 and Σ_j respectively. The evolution of this joint state (x_k^1, x_k^j), $k = 0,1,\ldots$, is viewed as the sample path of an "augmented system" which is driven by the exact same event sequence as the observed system Σ_1. As long as

the observed state and the state of some Σ_j, $j = 2,\ldots,N$, satisfy the "observability condition" $\Gamma(x_k^j) \subseteq \Gamma(x_k^1)$ in (4), then every observed event in Σ_1 is simply used to concurrently update the state of Σ_j. Intuitively, the observability condition indicates that the observed simulation experiment contains at every state all the event information necessary to proceed with state updates. If, however, $\Gamma(x_k^j) \supset \Gamma(x_k^1)$ for some state, then there is missing information and the simulation run corresponding to Σ_j must be "suspended" until some later time when the observability condition is again satisfied. The resulting procedure is known as *event matching*. It can be shown that this suspension preserves all statistical properties of the sample path being constructed in this fashion as long as all event lifetime distributions are exponentially distributed [3].

In order to describe the ASA event matching procedure more precisely, let us define a set $A = \{2,\ldots,N\}$, i.e., A contains the indices of the N-1 simulation experiments to be constructed in parallel. Let us also limit ourselves to parameters which do not affect event lifetime distributions; these are typically structural parameters such as the capacity of a queue or the population size of a closed queueing network. If following a state transition, $\Gamma(x_k^j) \supset \Gamma(x_k^1)$ for some $j \in A$, then j is removed from A until some future transition which causes $\Gamma(x_k^j) \subseteq \Gamma(x_k^1)$ to be satisfied; at this point, A is updated to include j once again. In other words, A represents the set of "active" simulation runs at any given time. In what follows we drop the subscript k and denote the current and next state of Σ_j by x^j and $(x^j)'$ respectively.

ASA - Event Matching Algorithm

Step 0. Initialize: $x^j = x^1$ for all $j = 1,\ldots,N$, and set $A = \{2,\ldots,N\}$.

Repeat the following steps for all $j = 2,\ldots,N$ with every event e observed in Σ_1:

Step 1. Update state: $(x^j)' = f_j(x^j, e)$ for all j such that $j \in A$

Step 2. Check for observability: if $\Gamma[(x^j)'] \supset \Gamma[(x^1)']$, then go to *Step 3*, else skip it and return to *Step 1*.

Step 3. Remove j from A and return to *Step 1*.

The ASA procedure is also shown in **Figure 3**, in order to contrast it to the conventional simulation approach. In this case, it is assumed that a simulation is already being performed, so that the objective is to generate a number of additional simulation runs in parallel. Similar to the SC method, ASA is based on the assumption that event lifetimes are exponentially distributed (actually, at most one non-exponential event lifetime distribution is allowed). In addition, there are cases where this assumption is not required; specifically, in the case where the system of interest is such that $\Gamma(x) = E$ for all states (see [19]).

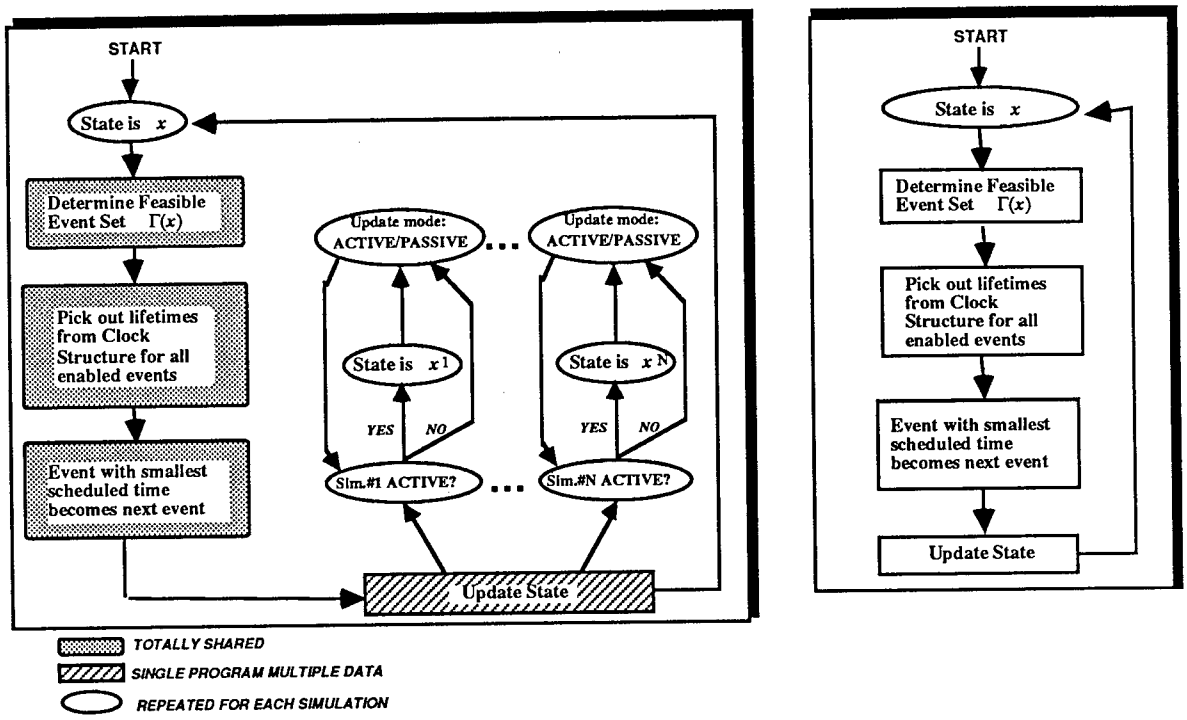


Figure 3. The ASA Concurrent Simulation scheme compared to conventional simulation

Let us now see how concurrency is exploited in the case of the ASA approach:

• *Sequential processing*: One has to serially perform all N experiments, but only the state update function (F3) needs to be repeated for each of the N simulation runs. All other functions are performed only once for the given nominal simulation. As in the SC method, however, this efficiency comes at some cost, which is contained in *Steps 2-4* of the Event Matching algorithm: one needs to check whether the feasible event set at the observed state $(x^1)'$ contains the feasible event set at state $(x^j)'$. If this is not true for some j, note that the jth simulation run is effectively suspended, since its state is forced to remain unchanged until the condition above is satisfied. This cost may be quite small, but there are also cases where it can be large. A quantifiable measure of this cost is provided in Section 2.5. As shown in some of our numerical results, the time required to perform N simulations through the ASA approach is much smaller than NT.

• *Parallel processing*: The implementation of the ASA approach on parallel computers is not entirely obvious. This is because here we assume that an existing simulation run is observed and the parallelism is exploited by extracting information from this run and using it to perform N additional

ones. However, in a parallel processing environment the nominal run must be executed on one of the processors. This leaves open the issue of how to use the front end computer in a SIMD or MIMD setting to synchronize the simulation runs executed in parallel.

There is one additional feature of the ASA approach that opens up some interesting possibilities. In a simulation environment, the choice of the nominal system is arbitrary. Depending on this choice, the computational cost incurred by *Steps 2-4* may be large or may be totally eliminated if the selected nominal system is such that the condition $\Gamma[(x^j)'] \supset \Gamma[(x^1)']$ is *never* satisfied. Therefore, determining appropriate choices that lead to elimination of *Steps 2-4* altogether is a critical issue further discussed in subsequent sections.

2.5. Quantifying Efficiency in Concurrent Simulation Methods.

During the course of this project, we implemented both the SC and ASA approaches described above and applied them to various problems in order to evaluate their efficiency relative to the "brute-force" method which we have taken to be the obvious baseline approach. As mentioned earlier, we have limited ourselves to sequential computers, since access to a parallel machine was not feasible in this project. However, it is easy to extrapolate to parallel computers, since one can only further benefit from a parallel processing environment. In this respect, the results included in this report may be viewed as lower bounds to the speedup that can be achieved.

To obtain a measure of "efficiency" for the concurrent simulation techniques under investigation, the following simple measure, referred to as the *speedup factor* for N parameter settings, is an obvious starting point::

$$\text{SPEEDUP FACTOR}(N) = \frac{\text{Brute force simulation time for } N \text{ runs}}{\text{Concurrent simulation time for } N \text{ runs}} \qquad (8)$$

By "brute force simulation time" here we mean the time to obtain N simulations on a sequential computer. That is, if the average simulation time for a single run is T, then the numerator in (8) is given by NT.

Although this appears to be a reasonable metric for a comparative study, it has several drawbacks. The most important drawback is that the speedup factor, as defined above, also depends on the specific number of events involved in the N simulation runs. In other words, the measure is dependent upon the "stopping condition" used in our simulation experiments. In particular, suppose each of the N simulation runs performed by "brute force" is defined to end after M events are generated and processed. In order for the speedup factor above to be well defined, we need to assume that the same is true for each of the runs generated through a concurrent simulation

technique. This is not generally true, however. In general, if a single run is ended after M events, the jth simulation run obtained through SC or ASA has processed a random number M_j of events which is generally not equal to M. Therefore, it makes sense to use the "simulation time per event" as a normalized measure, rather than the total simulation time.

To gain a better understanding of the speedup any of the proposed concurrent simulation techniques can achieve, let us return to the four simulation functions identified earlier (see Section 2.2). Consider a conventional discrete event simulation set to run until a total number of M events is generated. We can then define $C(M)$ to be the total time devoted to clock updates, $S(M)$ the total time devoted to state updates, $R(M)$ the total time devoted to generating random variates, and $E(M)$ the total time spent maintaining the Event Calendar (EC). Thus, when the Brute Force method is used for N simulations of M events each, the total number of events generated is NM, and the total time per event is given by $N\cdot[C(M) + S(M) + R(M) + E(M)]/NM$, or

$$T_{BF} = \frac{C(M)+S(M)+R(M)+E(M)}{M} \tag{9}$$

Let us now obtain the corresponding simulation time per event metrics, T_{SC} and T_{ASA}, for the SC and ASA techniques respectively. We will consider first the case of sequential processing and then parallel processing.

• *Sequential processing*: Starting with the SC approach, recall that all functions except state updates are shared by all concurrent simulations. Moreover, there is an inefficiency resulting from events which are not feasible, i.e., wasted random numbers. Let the time spent generating such infeasible events be denoted by $W_{SC}(M)$, where M is the number of events generated in the particular simulation. Let us adopt the convention that when simulations are concurrently performed, every individual simulation that reaches M events is stopped. Therefore, if N is the total number of simulations to be run, the total simulation time is given by $[C(M) + R(M) + E(M) + W_{SC}(M)] + N\cdot S(M)$, or, in terms of the total simulation time per event metric as in (9) above:

$$T_{SC} = \frac{[C(M)+R(M)+E(M)+W_{SC}(M)]+N\cdot S(M)}{NM} \tag{10}$$

A similar argument for the ASA approach gives

$$T_{ASA} = \frac{[C(M)+R(M)+E(M)+W_{ASA}(M)]+N\cdot S(M)}{NM} \tag{11}$$

where $W_{ASA}(M)$ is the corresponding potential inefficiency of ASA discussed in Section 2.4.2.

We can now obtain the speedup factors of SC and of ASA as a function of N. For simplicity, set $B(M) = C(M) + R(M) + E(M)$, and we get

$$SC\ \text{SPEEDUP}(N) = N \cdot \frac{B(M) + S(M)}{[B(M) + W_{SC}(M)] + N \cdot S(M)} \tag{12}$$

$$ASA\ \text{SPEEDUP}(N) = N \cdot \frac{B(M) + S(M)}{[B(M) + W_{ASA}(M)] + N \cdot S(M)} \tag{13}$$

Taking the limit as $N \to \infty$, it is interesting to observe that the speedup is *independent* of the approach used and is given by

$$\text{SPEEDUP}(\infty) = 1 + \frac{B(M)}{S(M)} \tag{14}$$

which provides a fundamental upper bound to the speedup we can attain for our concurrent simulation techniques. Observe that if $B(M) >> S(M)$, i.e., the time spent in state updates is small relative to the remaining simulation functions, then SPEEDUP(∞) can be very large. Conversely, if $B(M) << S(M)$, then the speedup achieved will be minimal. Interestingly, the quantities $B(M)$ and $S(M)$ can be estimated from a single simulation run, so one can a priori predict the speedup expected (or at least an upper bound) by a given technique when N is large.

• *Parallel processing*: In this case, the state update function corresponding to $S(M)$ can be parallelized (i.e., N processors can perform the updates in parallel). Accounting for some inefficiency due to communication delays across processors and synchronization, denoted by $P_{SC}(M)$ and $P_{ASA}(M)$ respectively, (12) and (13) become:

$$SC\ \text{SPEEDUP}(N) = N \cdot \frac{B(M) + S(M)}{[B(M) + W_{SC}(M) + S(M)] + N \cdot P_{SC}(M)} \tag{15}$$

$$ASA\ \text{SPEEDUP}(N) = N \cdot \frac{B(M) + S(M)}{[B(M) + W_{ASA}(M) + S(M)] + N \cdot P_{ASA}(M)} \tag{16}$$

In this case, letting $N \to \infty$, we may again obtain upper bounds for the speedup factor. This time, however, this bound depends on the specific technique used which may incur different types of communication and synchronization costs:

$$SC\ \text{SPEEDUP}(\infty) = \frac{B(M) + S(M)}{P_{SC}(M)}, \qquad ASA\ \text{SPEEDUP}(\infty) = \frac{B(M) + S(M)}{P_{ASA}(M)} \tag{17}$$

Note that if the communication and synchronization delay is $<< B(M)+S(M)$ (i.e., negligible

compared to the time required to simulate M events at any processor), then speedup can become extremely large. This motivates us to seek techniques designed to minimize these delays in order to maximize the speedup attained.

2.6. A Universal Concurrent Simulation Method.

The concurrent simulation methods presented in Section 2.4 are limited to models with Markovian event lifetime distributions. As already mentioned, several techniques mat be used to provide extensions and approximations. It is reasonable, however, to seek a concurrent simulation method "universally" applicable to arbitrary models.

During the course of this project, such a method was developed and a preliminary analysis of its properties carried out. In this section, we will limit ourselves to a description of this method, which is still the subject of ongoing research. The main idea of this new concurrent simulation approach lies in a fundamental change in processing the Event Calendar (see Section 2.2) of a discrete event simulator: Current practice is based on discarding all information related to an event as soon as this occurs; our approach is to save this information until *all* concurrent simulations have made effective use of it. This is tantamount to introducing some "memory" into concurrent simulation schemes described earlier.

A key element in the new approach is the concept of "time warping". This concept is encountered in some software environments where distributed processing takes place (e.g., [20]), resulting in occasional loss of synchronization across these processes. When this happens, one may "time warp" or "roll back" to a point where data consistency is guaranteed. As it applies to the ASA concurrent simulation approach discussed earlier, the idea of "time warping" is the following. We have seen that when an unobservable event is encountered, we need to suspend the sample path construction at some time instant t_1. While the nominal simulation evolves, several events may occur and their lifetimes recorded. When the unobservable event finally becomes feasible, at some time $t_2 > t_1$, it is possible to utilize much of this information, collected over $[t_1, t_2]$. In particular, we may be able to perform several state updates from time t_1 onwards, possibly constructing an entire piece of the desired sample path all the way up to the current time t_2. This is what we refer to as "time warping".

Time warping is equivalent to delaying the construction of a sample path until adequate information is collected from observed data, in order to correctly update the state, and possibly generate several state updates during the time warp. It should be clear that while this mechanism allows for generality in the system model, it involves additional storage requirements for various

observed event lifetimes, as well as some extra data processing. Therefore, a key issue that remains to be investigated in depth is the tradeoff between the speedup attained due to concurrency vs. the additional data storage requirements. We expect that specific implementations of our basic concept may be crucial.

To describe the time-warping mechanism, let x be the current state of the nominal system being simulated, t the current time, and e the current event. For a sample path denoted by $\xi(\theta_m)$, let x_m be the state, and t_m the time of the most recent event in the construction. In general, $t_m \leq t$, since violating observability may have forced us to suspend construction at a time instant t_m. In conventional simulation, note that an event is discarded from the Event Calendar as soon as it occurs and has been used to update time and the system state. Now, suppose that we are willing to maintain a *list of unprocessed events*, instead of immediately discarding them:

$$A_m = \{(e_1, t_1), (e_2, t_2), \ldots\} \tag{18}$$

where (e_k, t_k) are all events observed over the interval $(t_m, t\,]$, which have yet to be used in our construction. Next, if the constructed sample path is suspended at state x_m, let:

$$\Gamma(x_m) = \{\beta_1, \beta_2, \ldots, \beta_L\} \tag{19}$$

where $\Gamma(x_m)$ is the feasible event set at state x_m (see Section 2.2). Since the construction is suspended due to unobservable events, the only way to continue is if:

$$\beta_k = e_j \quad \text{for all } \beta_k \in \Gamma(x_m) \text{ and some } (e_j, t_j) \in A_m \tag{20}$$

Thus, the construction proceeds only when information about an event β_k can be found in the set A_m, for all $k = 1,\ldots,L$. Let I_m be the set of all indices j of elements in A_m that satisfy this condition. At this point, a time warp can take place. First, we can determine the next event for the constructed sample path, which is simply the triggering event $arg\ min_{j \in I_m} \{t_j\}$. We can then update the state and determine its feasible event set. It is possible that all events in this set still satisfy (20), in which case we update the state once again, and so on. The time warp comes to an end when either condition (20) is no longer satisfied or the construction has caught up with the observed sample path, i.e. $A_m = \varnothing$ or $t_m = t$. It should be clear that this approach is completely general.

Preliminary numerical results obtained in the latter stage of this project have yielded speedup factors in the neighborhood of 2 for a large class of models. We have found that the speedup is dependent on some basic structural features of the models being simulated. However, there remain several issues to investigate beyond this project, including (a) The tradeoff identified between speedup and storage requirements, (b) The development of an explicit prototype algorithm, and (c) Extensive testing of the approach through several types of models from C^3I application areas.

3. COMPARATIVE STUDIES ON CONCURRENT SIMULATION EFFICIENCY.

In this section, we introduce a testbed problem (Section 3.1) based on which we evaluate the concurrent simulation methods described in Section 2. In sections 3.2 and 3.3 we present several numerical results applicable to this problem intended to quantify the efficiency of such methods vis-a-vis "brute force" simulation and provide a comparison between different methods.

3.1. A Basic Resource Allocation Problem.

The problem considered here is motivated by resource allocation issues which are frequently encountered in C^3I systems. We begin by introducing the problem in a generic way and then provide different interpretations for specific applications. Consider *N users* and *M resources* to be allocated to these users. In our particular problem, the users may be thought of as processors in a computer system or switches in a communication network; the resources are memory or buffer slots to be allocated over the users. Adopting a standard queueing model, the system of interest is depicted in **Figure 4**.

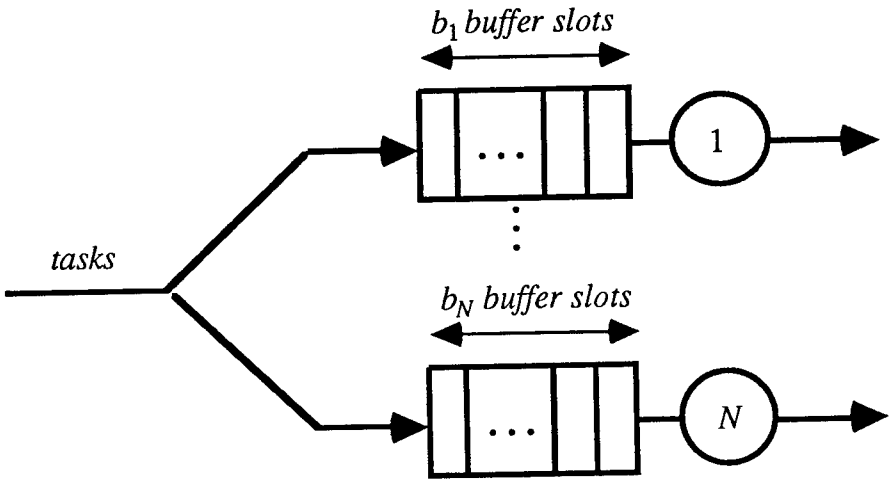


Figure 4. A Basic Resource Allocation Problem

In this model, we may think of tasks submitted to a system in order to be executed by one of the N processors in parallel. A task is sent to one of these processors in some arbitrary fashion (for simplicity, by randomly routing a task to one of the processors; however, the precise routing mechanism is not important as we will see). Tasks are queued at each processor, provided there is

buffer space, otherwise they are discarded and lost. An *allocation* is a vector vector $\mathbf{b} = [b_1,\ldots,b_N]$, where b_i is the number of buffer slots at processor i and $b_1+\ldots+b_N = K$. The cardinality of the set of all possible allocations, B, can be shown to be $(K+N-1)!/K!(N-1)!$ As an example, for $N = 6$ servers and $K = 20$ buffer slots, the size of the search space is $|B| = 53{,}130$ possible allocations. The problem of interest here is

$$\text{find } \mathbf{b}^* \in B \text{ to minimize } J(\mathbf{b}) = E[L(\mathbf{b})] \qquad \textbf{(P)}$$

where we are careful to distinguish between $L(\mathbf{b})$, the performance obtained over a *specific sample path* of the system (e.g., a single simulation run over some period of time), and $J(\mathbf{b})$, the *expectation over all possible sample paths*. It is clear that this problem is combinatorially hard. If an analytical expression for $J(\mathbf{b})$ is not available, one needs to simulate this system over all possible allocations in order to determine the optimal one $\mathbf{b}^*$. We will address problem **(P)** in more detail in Section 4.

In the next sections we concentrate first on any one of the branches in **Figure 4** and then on the entire system. In the former case, we are interested in a simulation-based parametric study over different buffer capacity values. In the latter, we are interested in a similar study over different allocation vectors.

3.2. Concurrent Simulation for a Single-Server Model.

Consider a single-server model representing any one of the branches in **Figure 4**. Thus, we obtain the simple system shown in **Figure 5**, where the number of buffer slots is denoted by k. Assume we can simulate the corresponding queueing model for an arbitrary arrival and service process under a particular value of k, say $k = b$. We are then interested in evaluating the system's performance under values of $k \neq b$. This would allow us to predict the behavior of the queueing model under different parameter settings. As we will see in Section 4, this information is essential in solving problem **(P)**.

In the case where the task arrival process and the service process are arbitrary, this model is the well-known G/G/1/k queueing system. If either one of these processes is characterized by an exponential (Markovian) distribution, then the 'G' is replaced by an 'M'. In view of the discussion of Section 2.2, a G/G/1/k queueing system is simulated as a process driven by an event set $E = \{a, d\}$, where a denotes a task arrival and d denotes a task departure. Letting X denote the number of tasks present in the system, note that the state transition mechanism is particularly simple: if a occurs, the X is increased by 1, and if d occurs X is decreased by 1 (provided $X > 0$). It is also

clear that the feasible event sets in this system are such that $\Gamma(X) = \{a, d\}$ for all $X > 0$, and $\Gamma(0) = \{a\}$, since no departure is feasible when the system is empty.

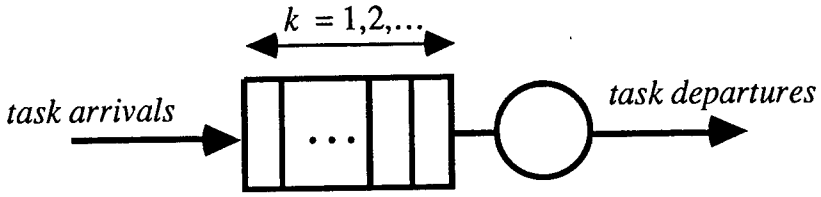


Figure 5. Single-Server Model

Our objective here is to apply the Standard Clock (SC) method of Section 2.4.1 and the ASA method of Section 2.4.2, and compare them to each other and to the brute-force repetitive simulation method (Section 2.3) over a number of values of k. The basis of this comparison is the speedup factors defined in (12) and (13). Obviously, we expect the values of these speedup factors to be greater than 1 if the concurrent simulation techniques are indeed efficient. There are, however, a number of additional issues that this comparative study has addressed:

1. Comparing the speedup factor of SC vs. ASA to determine which method is more efficient.
2. Studying the effect of model parameters on the speedup factor. For example, if the utilization of the queueing system increases, how does that affect speedup?
3. Recall that in the case of ASA, we have the additional degree of freedom in selecting the nominal system to simulate, from which others are concurrently simulated. Therefore, there is the additional issue of studying the effect of such a choice, and selecting the best possible such system, if possible.

In the subsections that follow we present the results of this comparative study. Since the SC method is limited to Markovian models, the model considered in this study is the M/M/1/k, where the range of k is {1,2,...,5000}.

3.2.1. CPU Times.

We begin with a set of simulation results intended to measure the CPU time observed for different methods. In all cases, the system is simulated for a total of 10,000 events (arrivals and departures). For consistency, all simulations were carried out on the same computer workstation.

We use the following notation in the figures referenced thereafter, where our results are summarized:

- BF denotes the Brute-Force method, i.e., if N different values of k are of interest, then N

separate simulation runs were performed.

- SC denotes the Standard Clock method.
- ASA-MIN denotes the ASA method (i.e., the Event Matching algorithm of Section 2.4.2) with the nominal system selected to be the one with the smallest value of k. For example, if the range of k is {1,...,5000}, the ASA-MIN algorithm is based on an M/M/1/1 system as the nominal one.
- ASA-MAX denotes the ASA method with the nominal system selected to be the one with the largest value of k. For example, if the range of k is {1,...,5000}, the ASA-MAX algorithm is based on an M/M/1/5000 system as the nominal one.

In order to study the effect of the model parameter values, we have defined ρ to denote the utilization in our queueing model, i.e., the ratio of the task arrival rate over the service rate. The values of ρ are then varied as indicated in what follows.

Figure 6 shows CPU times obtained as a function of the "degree of concurrency", i.e., the number of systems simulated concurrently, which varies from 1 to 5000. For each of the four methods considered, four different values of ρ were also considered: ρ = 0.2, 0.5, 0.8, 1.1. The efficiency of all three concurrent simulation methods applied here compared to the BF method is clearly illustrated. For example, the simulation of 5000 systems through BF requires about 1100 sec., compared to about 100 (an order of magnitude less) required by the ASA-MAX method. It is also clear that the ASA-MAX method is the most efficient one.

Another interesting observation is that the effect of ρ on the SC method is much more pronounced than it is on the ASA method. This is easier to see in **Figure 7**, which is simply a magnified version of **Figure 6** limited to a CPU time range that does not exceed 390 sec. In both figures "0.2 to 1.1 down" means that within the group of curves corresponding to the four values of ρ for a particular method the one on top is for ρ = 0.2 and the one at the bottom is for ρ = 1.1. Conversely, "0.2 to 1.1 up" means that within the group of curves corresponding to the four values of ρ for a particular method the one at the bottom is for ρ = 0.2 and the one at the top is for ρ = 1.1. The fact that the SC method is so sensitive to ρ is not surprising if one recalls the main source of inefficiency in the SC algorithm: One needs to check if an event is feasible and if it is not that event is simply discarded; in our case, d events generated by the simulator when the system is empty are the ones to be discarded. Clearly, when ρ is large, the queue is hardly ever empty and few d events need to be discarded; however, at low utilizations, such events are often infeasible, resulting in more wasted events and hence higher CPU times.

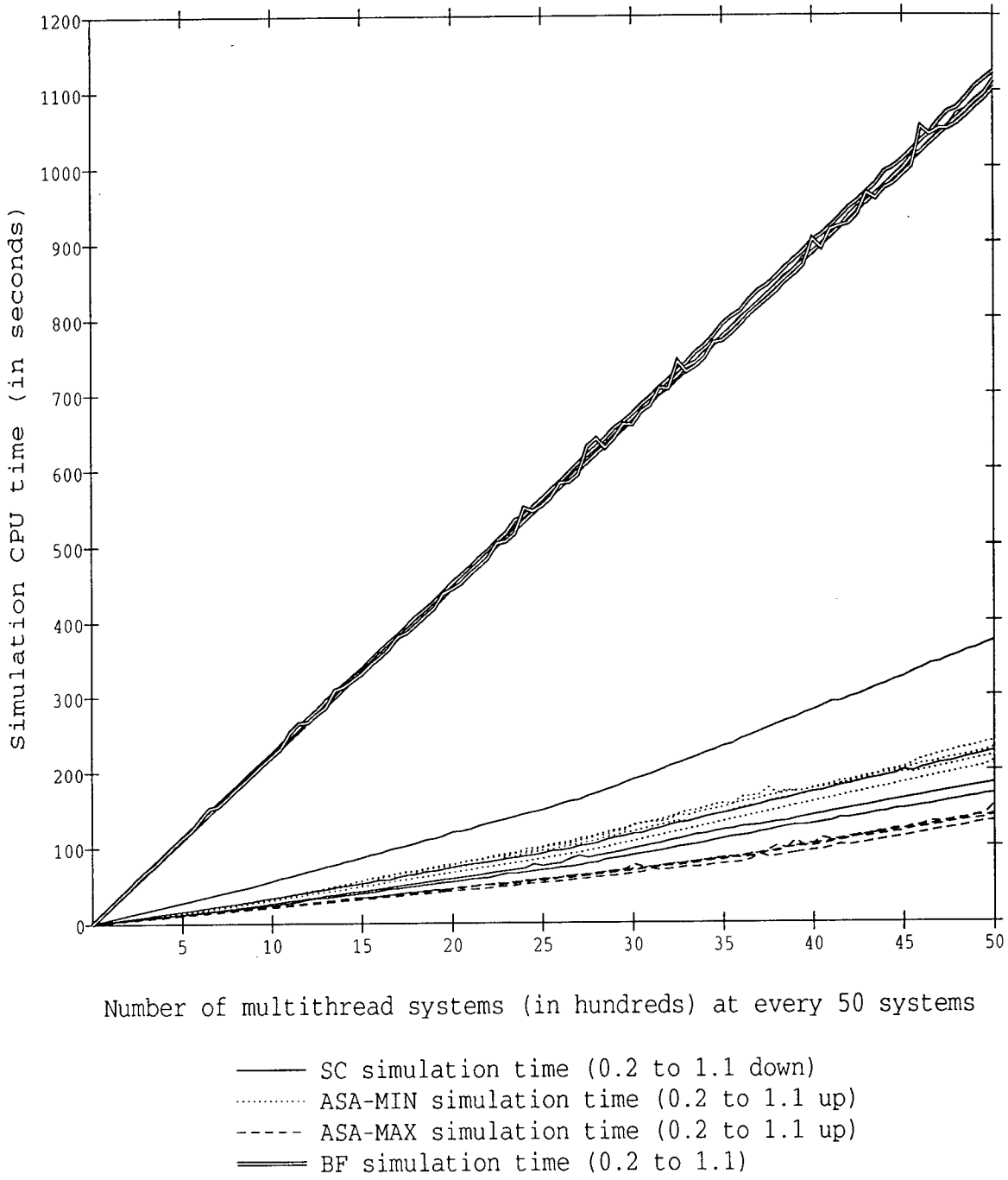


Figure 6: CPU Times for Single Server Model (10,000 events per run)

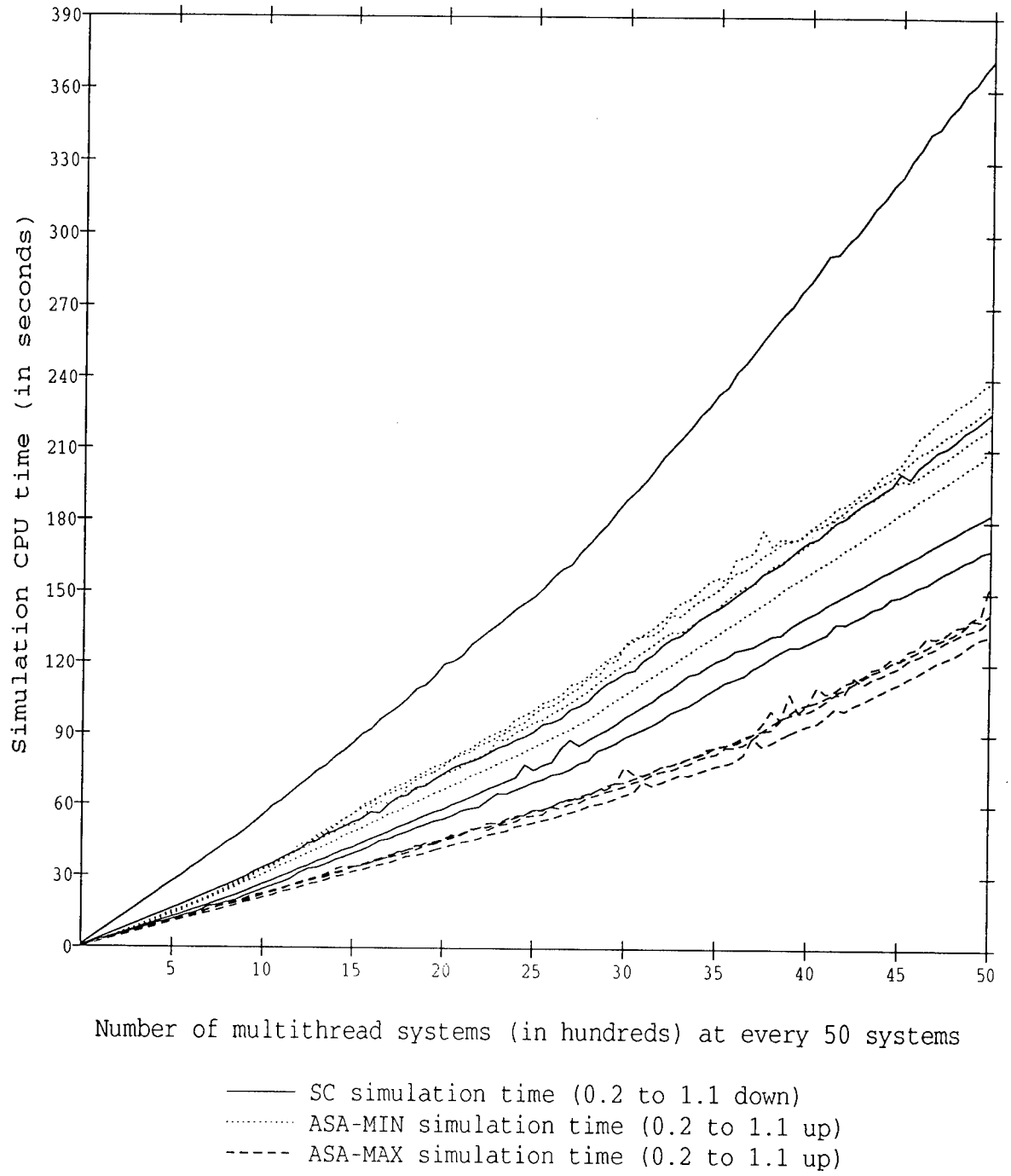


Figure 7: CPU Times for Single Server Model (magnification)

3.2.2. Speedup Factors.

Although the CPU time plotted in **Figures 6-7** provides an indication of the relative efficiency of the various methods considered, the speedup factor, as defined in Section 2.5, gives a much more direct quantifiable measure.

In **Figure 8**, the SC speedup factor is plotted as a function of the degree of concurrency (as in the previous section) for different values of the utilization parameter ρ. Note that this factor can be as high as 10, but subsequently goes down. It is conjectured that this is due to the fact that for a large number of concurrent simulations, memory limitations cause a large number of page faults, which in turn affects the CPU times involved. These memory limitations are entirely hardware-dependent and out of our control for the purpose of this study.

In **Figures 9-10**, similar results are shown for the ASA-MIN and ASA-MAX methods respectively. As already observed, the ASA-MAX approach is the most efficient one, as well as the most robust with respect to changes in ρ.

3.3. Concurrent Simulation for *N*-Parallel Server Model.

We now return to the N-parallel server model of **Figure 2**, with $N = 6$. We have performed a number of concurrent simulation experiments for this model in order to obtain results similar to those of Section 3.2. In the results reported next, the arrival rate considered was fixed at $l = 10$ tasks/sec. Each arriving task is routed to the ith branch with probability c_i, $i = 1,\ldots,6$. In particular: $c_1 = 0.1$, $c_2 = 0.2$, $c_3 = 0.1$, $c_4 = 0.2$, $c_1 = 0.2$, $c_1 = 0.2$. The service rates at the branches were set to $m_1 = 3$ tasks/sec, $m_2 = 2$ tasks/sec, $m_3 = 2$ tasks/sec, $m_4 = 3$ tasks/sec, $m_5 = 3$ tasks/sec, $m_6 = 2$ tasks/sec. As in the previous section:

- BF denotes the Brute-Force method, i.e., if M different values of k are of interest, then M separate simulation runs were performed.
- SC denotes the Standard Clock method.
- ASA denotes the ASA method (i.e., the Event Matching algorithm of Section 2.4.2) with the nominal system selected so that each queue has a capacity of 21 buffer slots (including the server itself). That is, each of the branches is an M/M/1/21 queueing model.

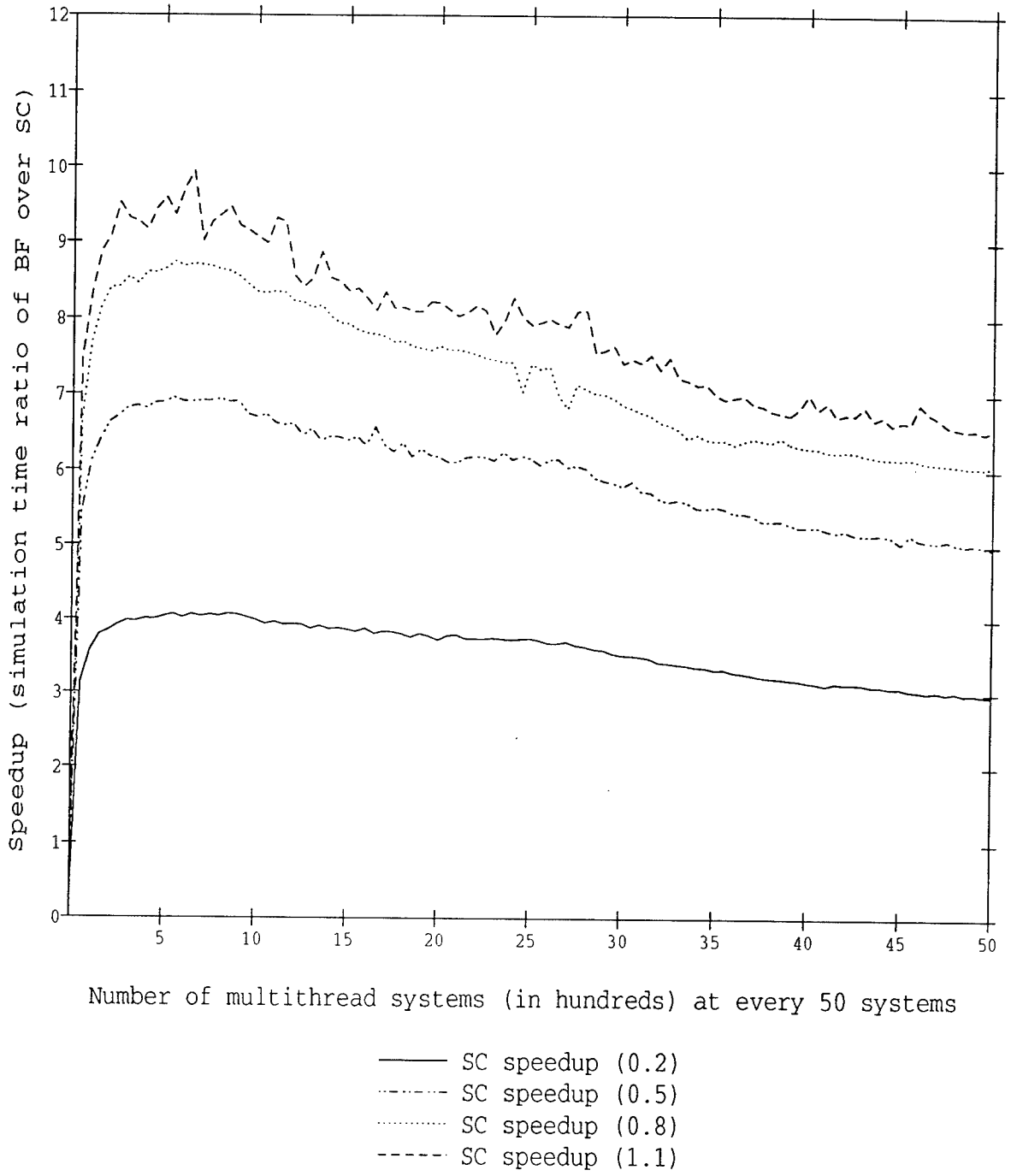


Figure 8: SC Concurrent Simulation Method Speedup Factors for Single Server Model (over different utilizations)

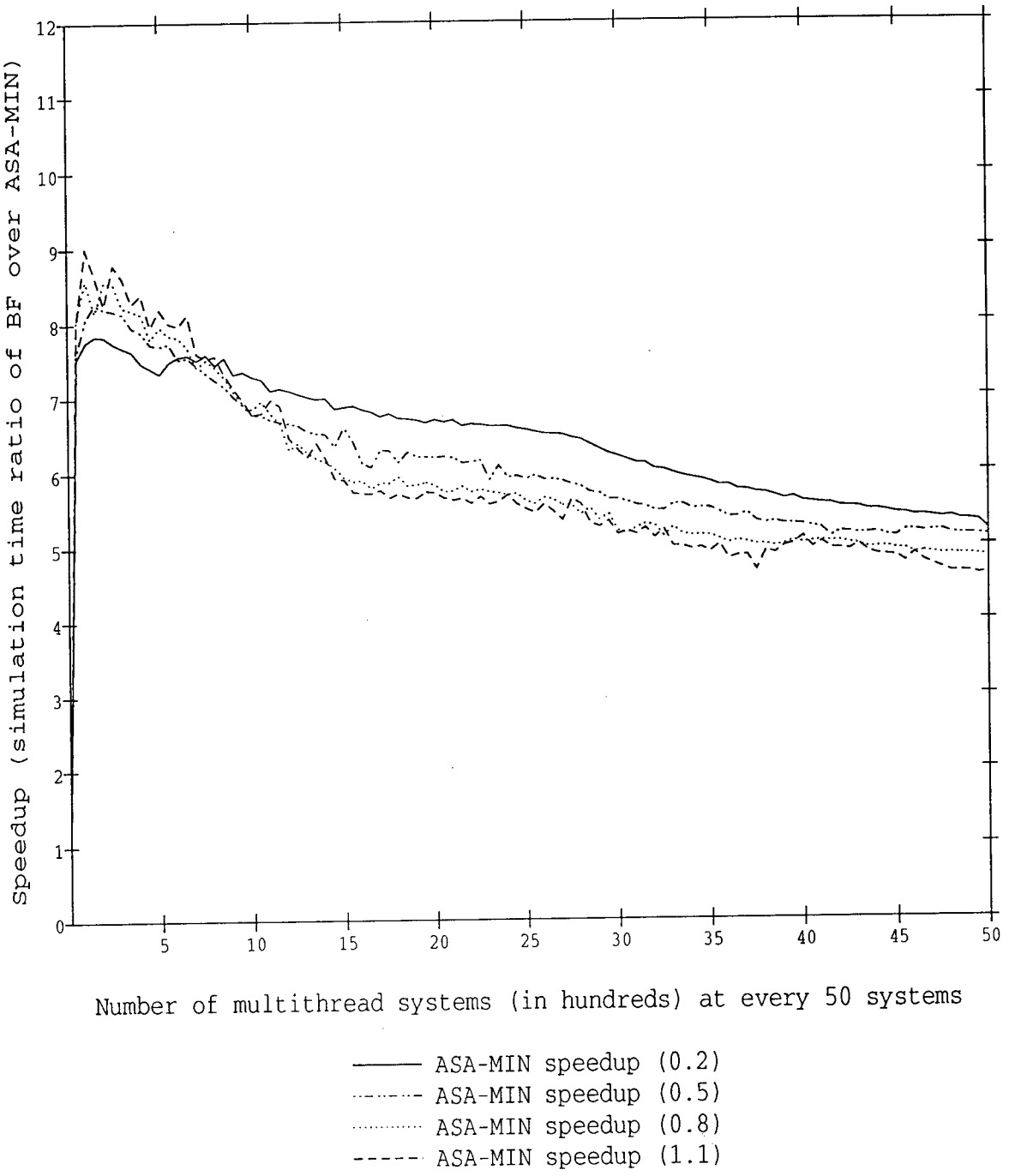


Figure 9: ASA-MIN Concurrent Simulation Method Speedup Factors for Single Server Model (over different utilizations)

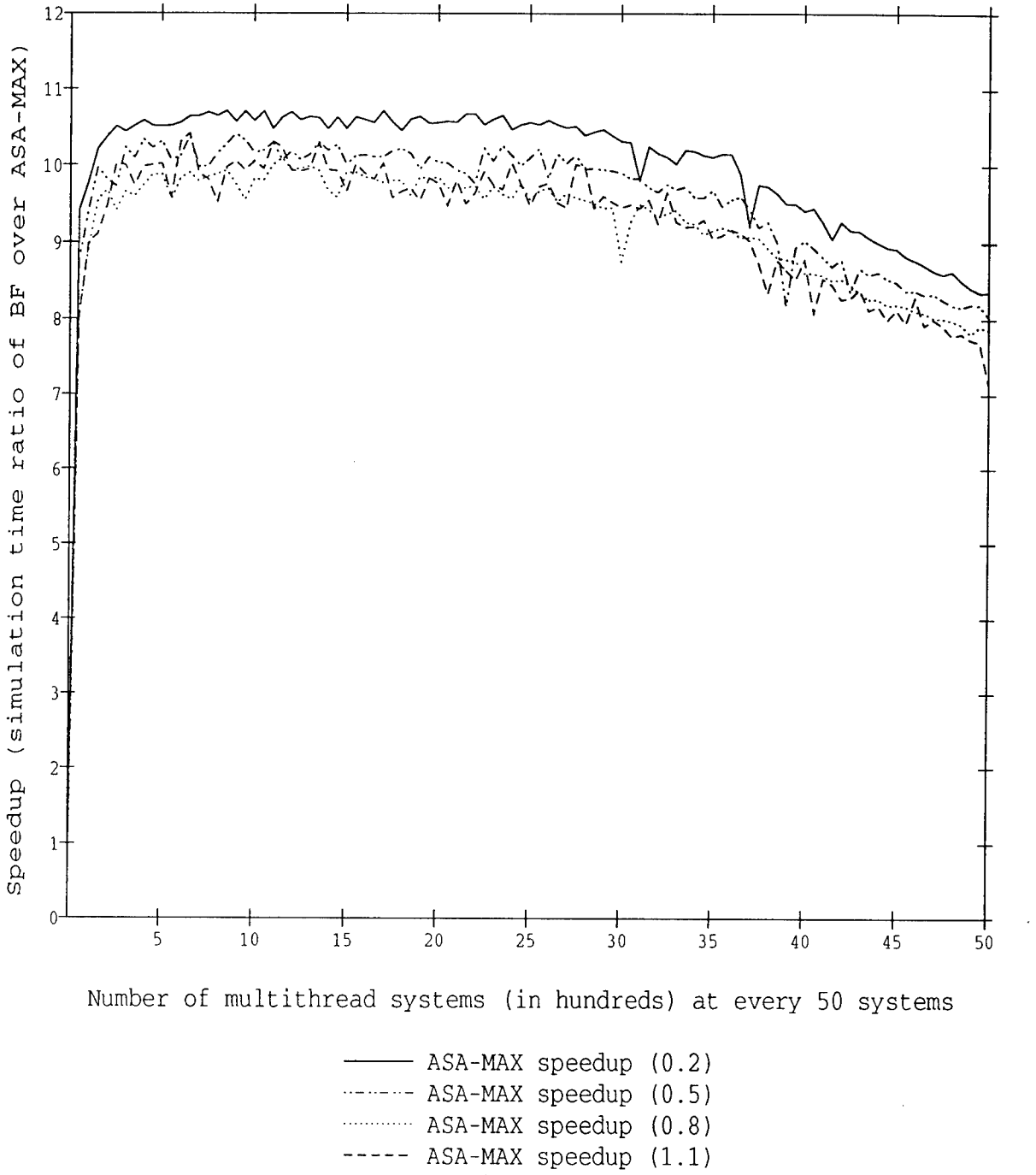


Figure 10: ASA-MAX Concurrent Simulation Method Speedup Factors for Single Server Model (over different utilizations)

3.3.1. CPU Times.

We begin with a set of simulation results intended to measure the CPU time observed for different methods. In all cases, the system is simulated for a total of 5,000 events (arrivals and departures from the six branches in the model). Our intent here is to simulate the system under different buffer allocations, i.e., vectors of the form $[b_1,\ldots,b_N]$, where b_i is the number of buffer slots at processor i.

Figure 11 shows CPU times obtained as a function of the "degree of concurrency", i.e., the number of systems simulated concurrently, which varies from 1 to 10,000. In this plot, the CPU time was recorded every 100th simulated system (by each method). In **Figure 12** this was done every 10th simulated system without showing significant variability in the observed CPU times. We can clearly see that ASA is somewhat more efficient than SC at these parameter settings.

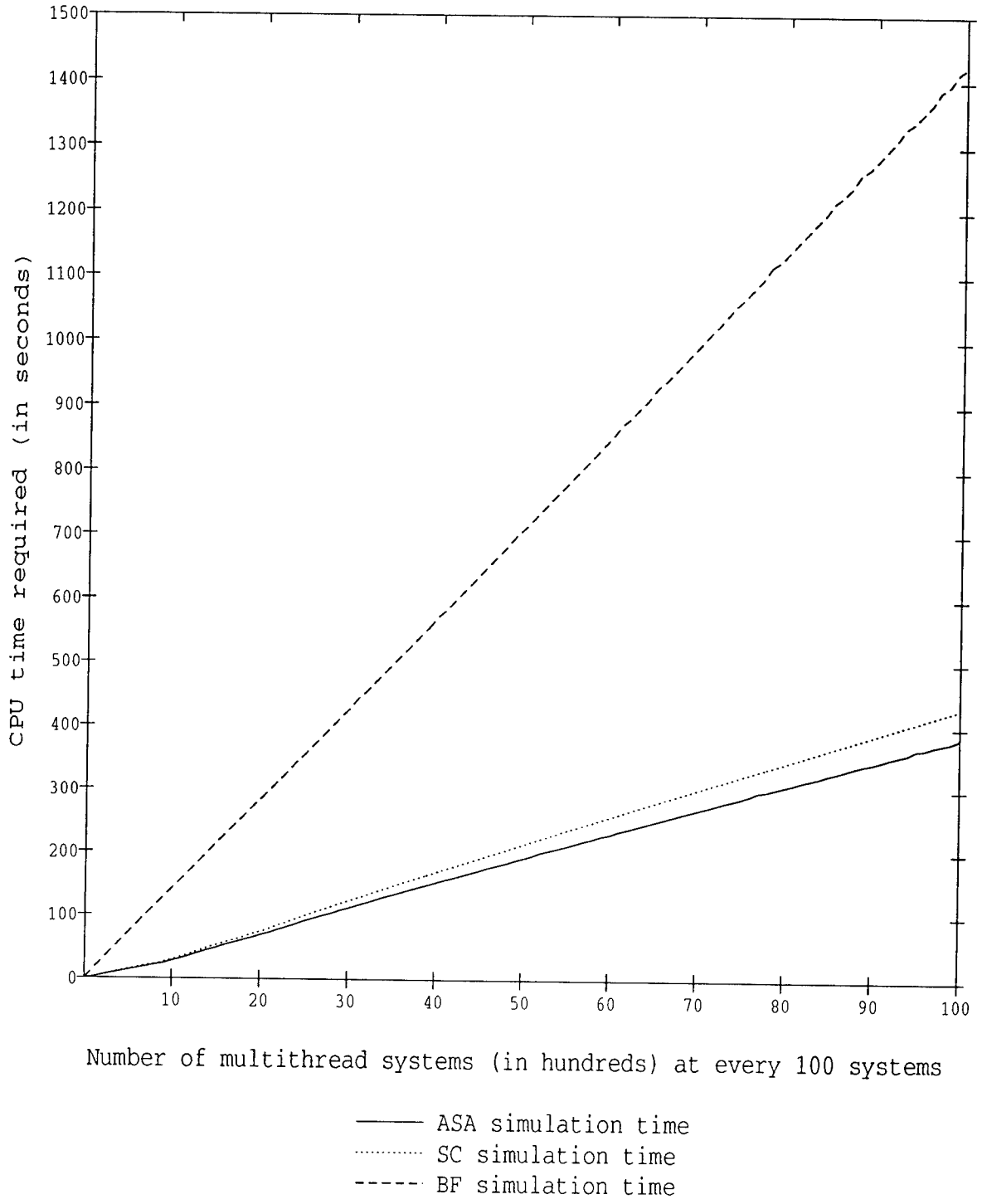


Figure 11: CPU Times for Parallel Server Model (recorded every 100 runs)

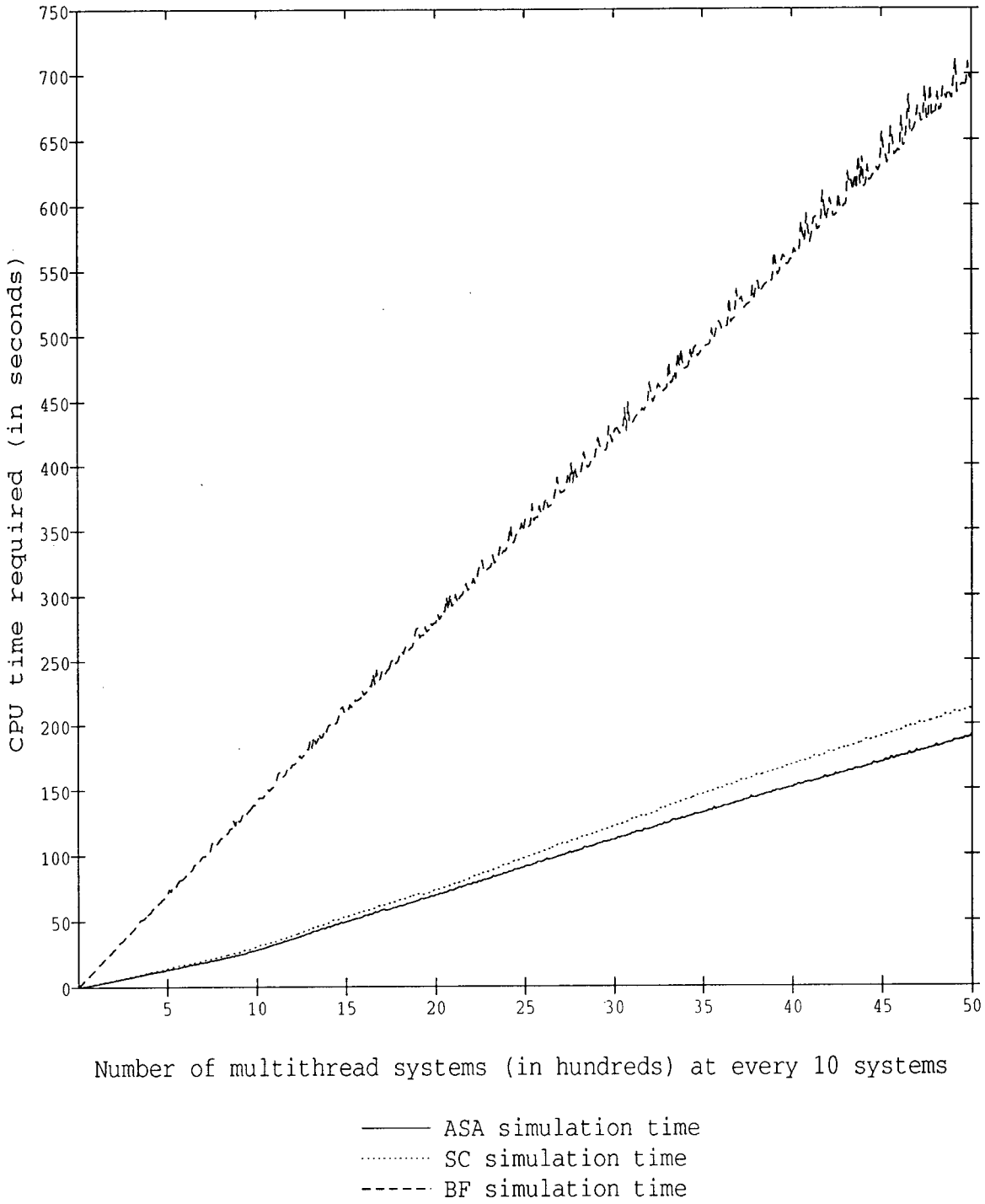


Figure 12: CPU Times for Parallel Server Model (recorded every 10 runs)

3.3.2. Speedup Factors.

Speedup factors corresponding to the results shown in **Figures 11-12** are shown in **Figures 13-14** respectively. As in our earlier results, we observe that the speedup factor reaches a maximum value and subsequently decreases with the degree of concurrency. Again, our conjecture is that for a large number of concurrent simulations, memory limitations cause a large number of page faults, which are purely hardware-dependent. Note also that the speedup factor here reaches a value of about 6, as opposed to about 10 in Section 3.2.2. An explanation of this fact is provided when one considers the four basic functions of a simulator described in Section 2.2 and repeated here:

(F1) Maintain an Event Calendar which is updated through the triggering event mechanism

(F2) Update the simulation clock

(F3) Update the state after every event occurrence

(F4) Generate random variates corresponding to various event lifetimes.

The main benefits from concurrent simulation come from the computational savings mostly in (F4) and to some extent in (F1) and (F2), where only one Event Calendar and one clock are needed in the SC or ASA methods. On the other hand, (F3) is a function that has to be performed in either BF simulation or in SC or ASA. Now, if the system being simulated is such that (F3) is the most computationally intensive component, relative to the other three, obviously this affects the corresponding speedup factor (see also the analysis of Section 2.5 which captures this fact). Thus, if a system involves very complicated state transition mechanisms and relatively few random phenomena to be simulated, the speedup factor may be lower. Conversely, if state transitions are simple to describe and a large amount of random phenomena must be simulated, the speedup benefits become very substantial.

To further quantify these observations, we have repeated the simulation experiment above, except that we allowed for significantly longer runs of 100,000 events per simulation run. We have estimated the CPU time dedicated to different routines in the simulator by observing it over sample periods, and have specifically identified the following five routine types: (1) State updates, (2) Random variate generation, (3) Memory allocation, I/O, and other maintenance routines, (4) Event Calendar updates, (5) Simulation clock updates. The first corresponds to function (F3), the second to function (F5), the third to function (F1), the fifth to function (F2), and the fourth covers all other functionality.

The results for the BF simulation method are shown in **Figures 15-16**. In **Figure 15**, we

plot the fraction (%) of CPU time dedicated to each of the five components above as a function of the number of systems simulated. The values are essentially fixed, since all we do here is repeat a simulation experiment (with different parameter values) in the exact same fashion 5,000 times. In **Figure 16**, the same results are shown over 250 systems only. We can clearly see that more that 50% of the simulation effort goes into random variate generation, whereas state updates represent less that 10% of the effort.

The corresponding results for the SC method are seen in **Figures 17-18**. What is interesting here is the asymptotic behavior of the % CPU time plotted: In **Figure 17**, one sees that almost 100% of the simulation effort is ultimately devoted to state updates! The computation time for the remaining four components becomes relatively unimportant, since it is shared by all concurrently constructed sample paths. Figure 18 gives us a better idea of how fast the sharing due to the SC method becomes efficient over the degree of concurrency desired.

Similar results for the ASA method are shown in **Figures 19-20**. Once again, almost 100% of the simulation effort is ultimately devoted to state updates as the degree of concurrency increases.

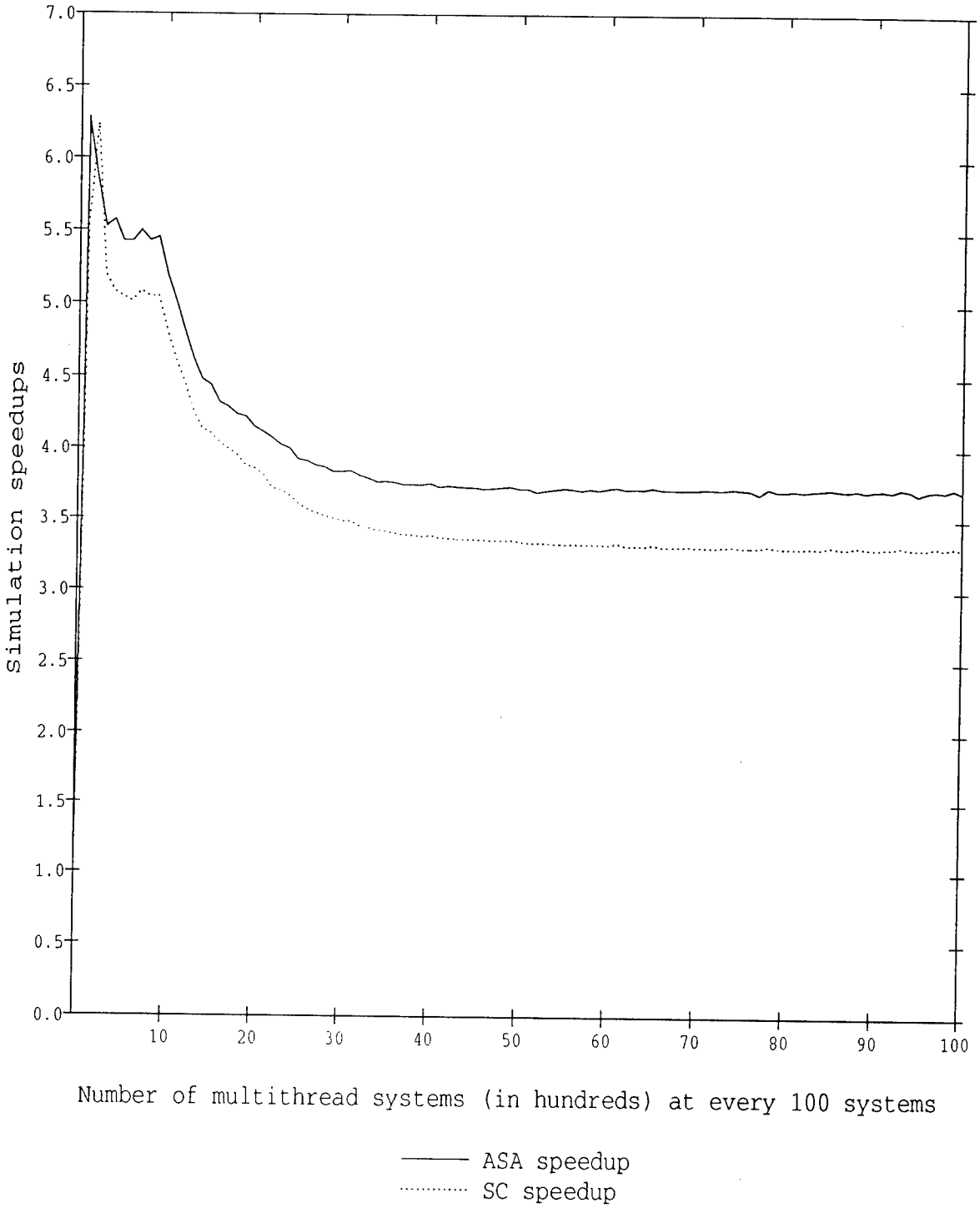


Figure 13: Speedup Factors for Parallel Server Model (recorded every 100 runs)

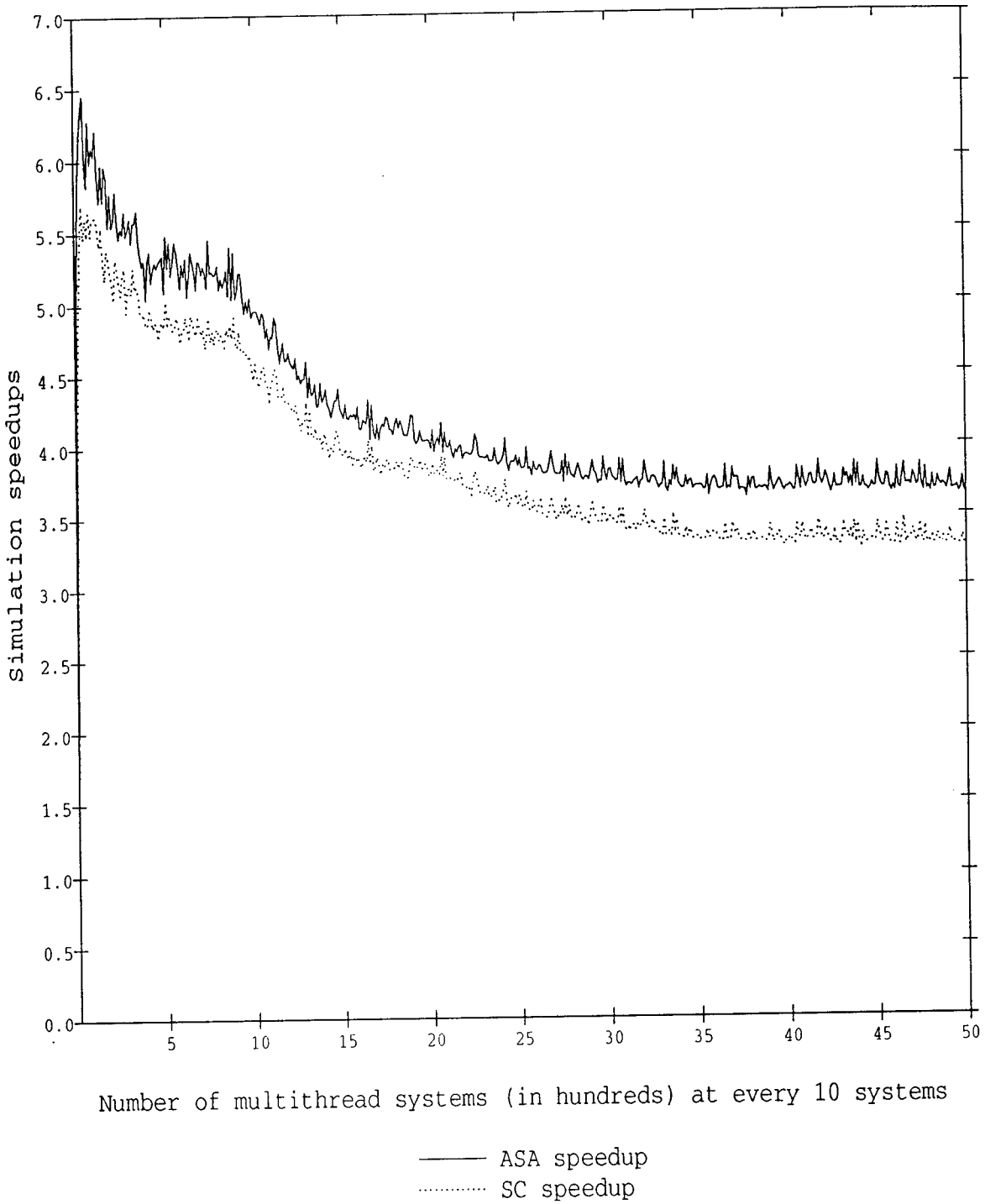


Figure 14: Speedup Factors for Parallel Server Model (recorded every 10 runs)

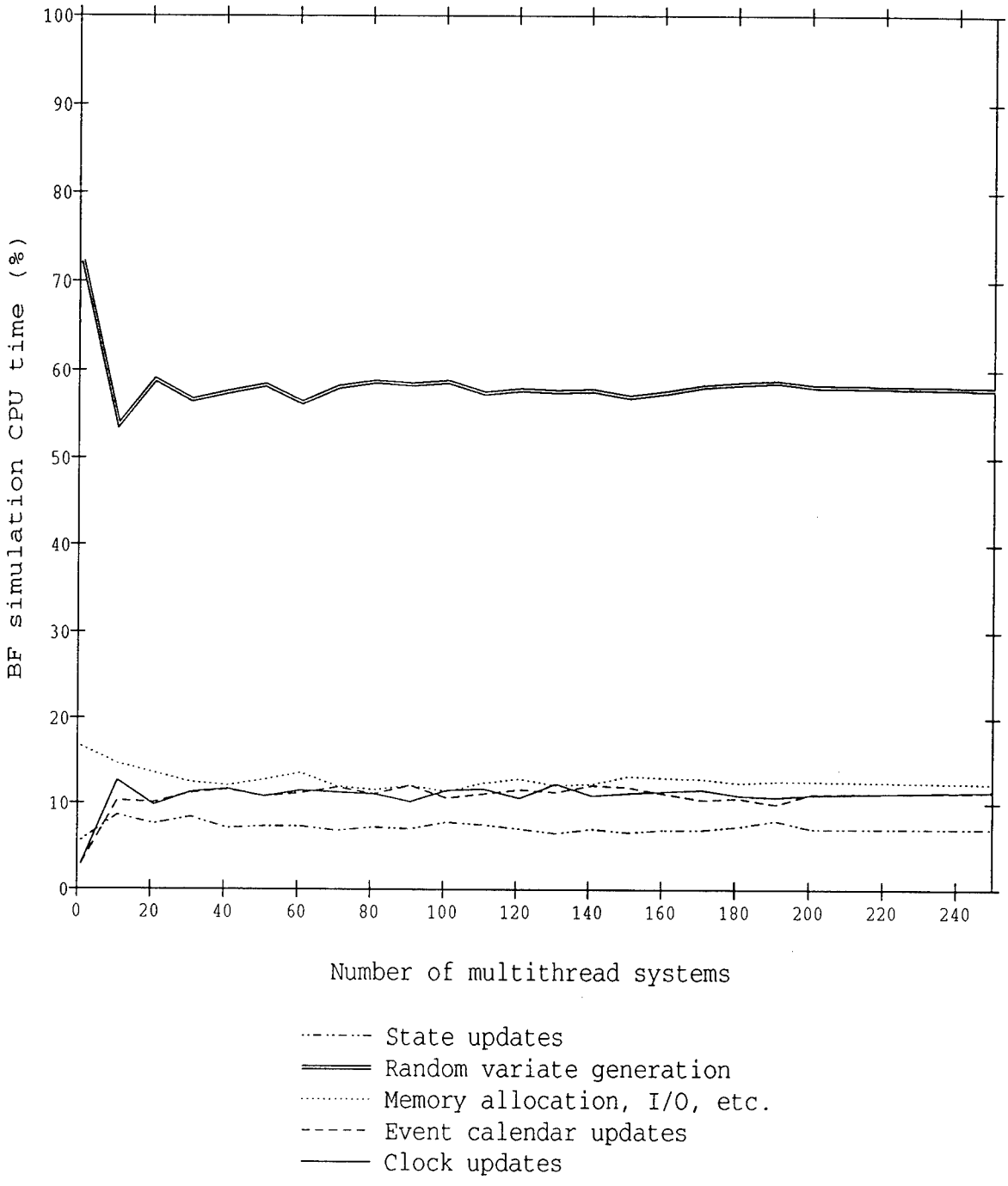


Figure 15: CPU Time Allocation in BF Simulation

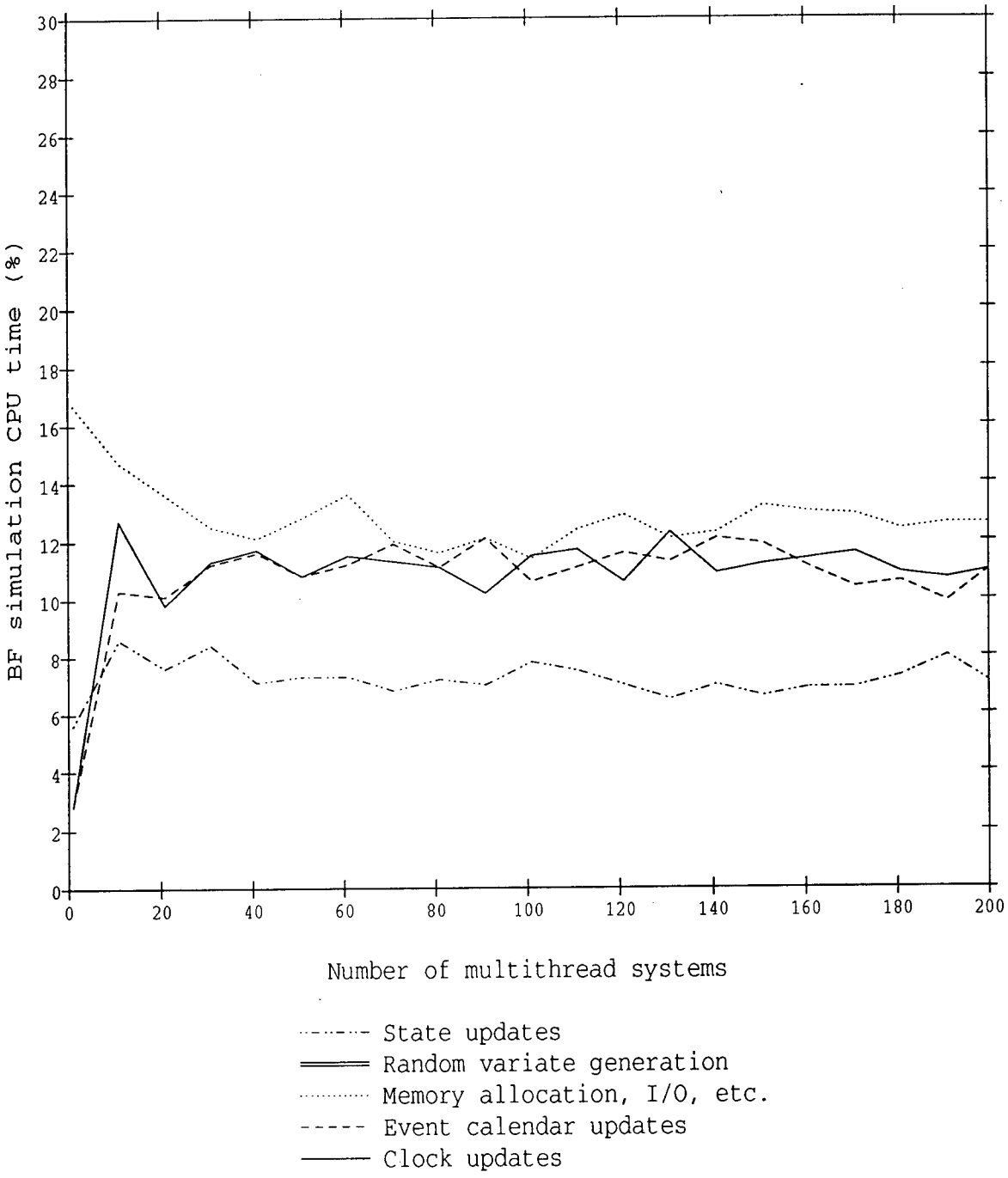


Figure 16: CPU Time Allocation in BF Simulation (magnification)

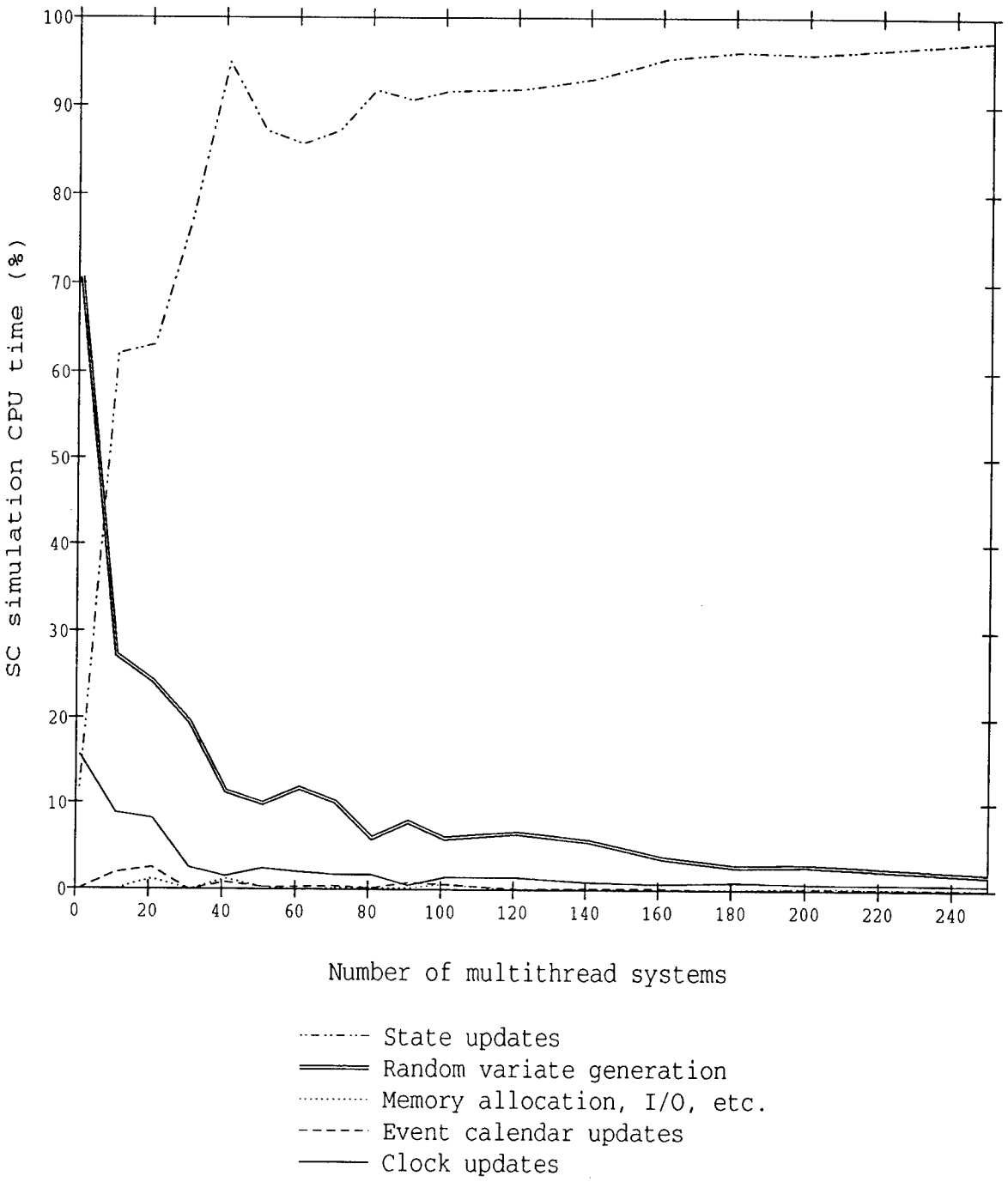


Figure 17: CPU Time Allocation in SC Concurrent Simulation Method

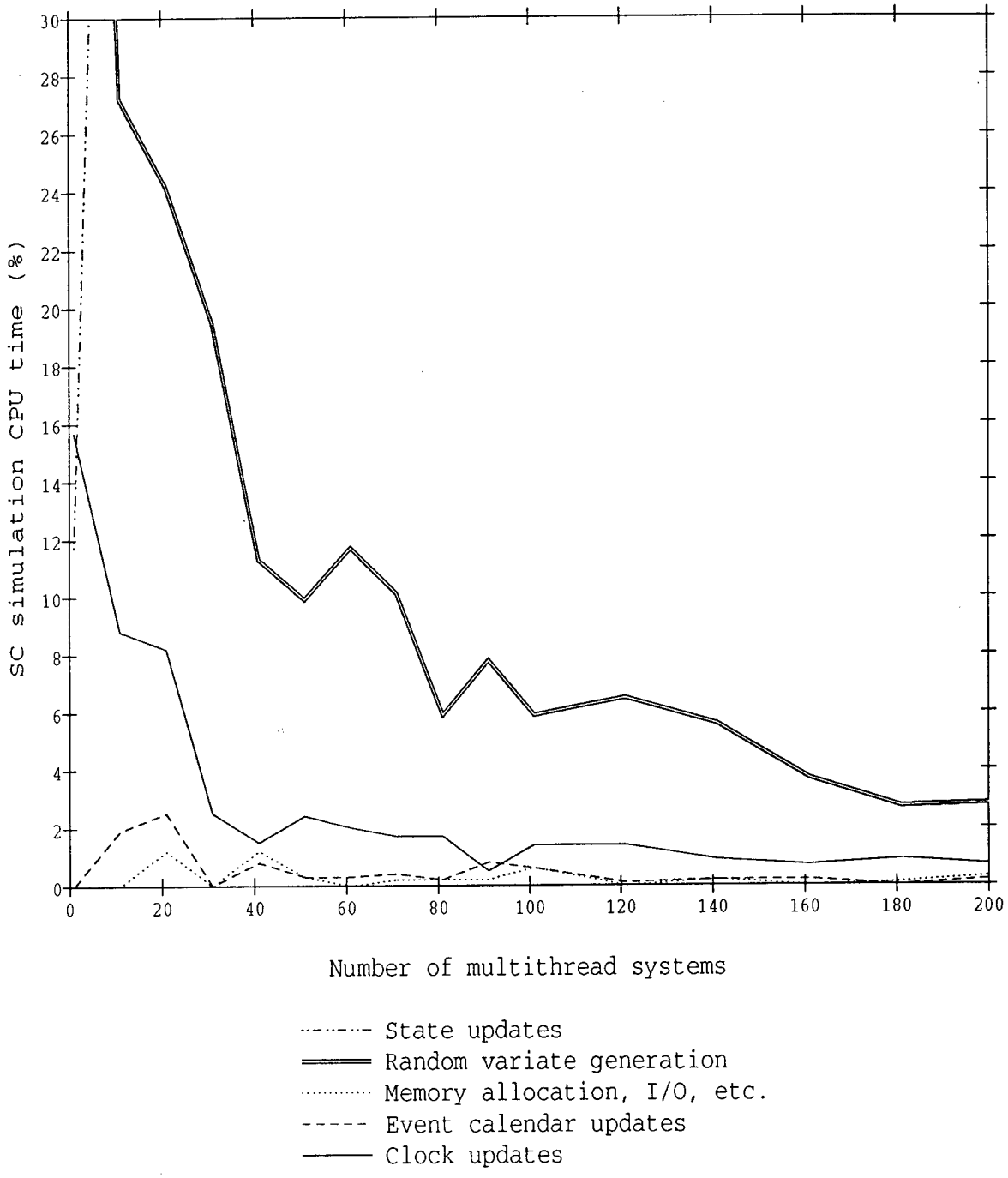


Figure 18: CPU Time Allocation in SC Concurrent Simulation Method (magnification)

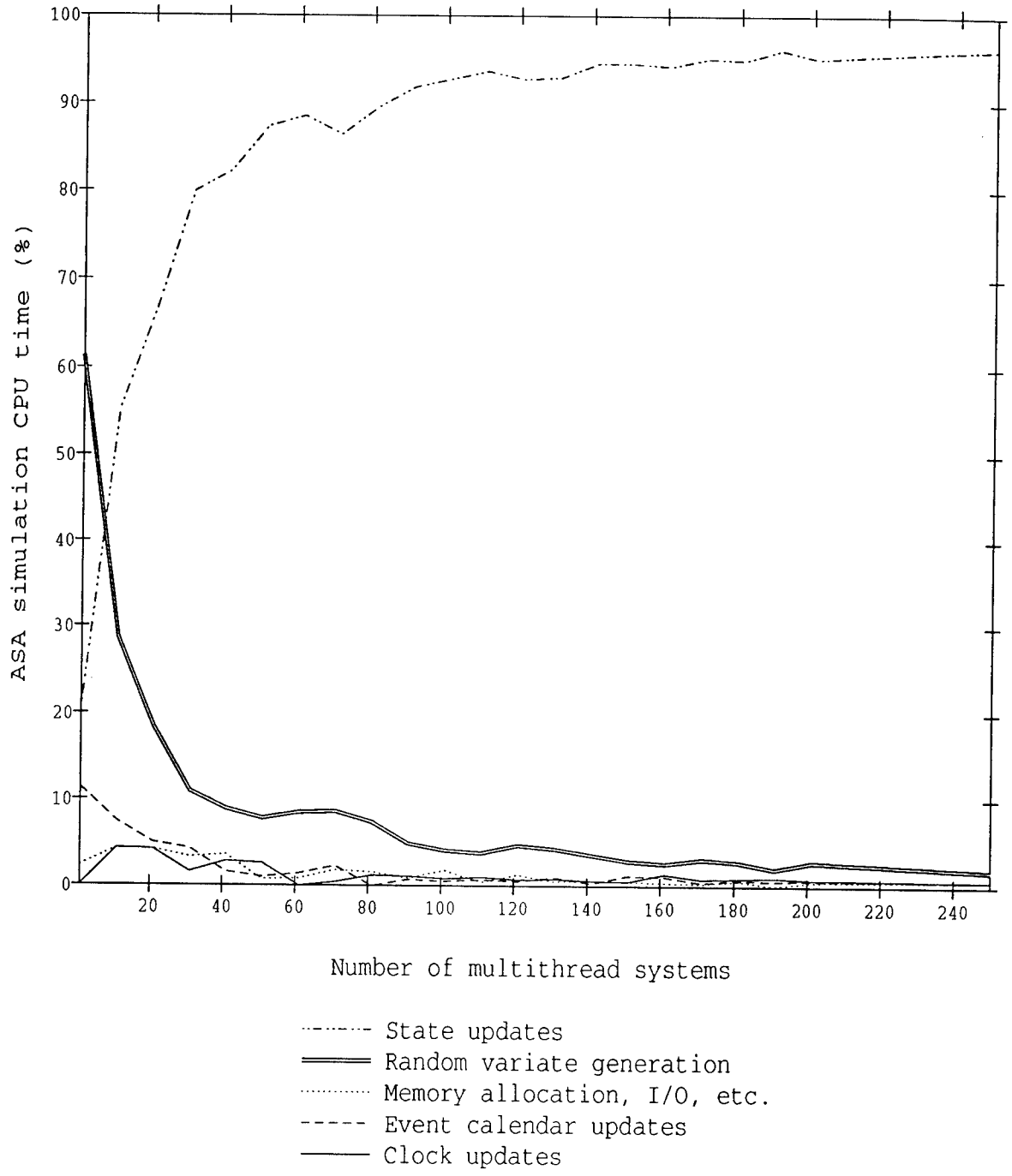


Figure 19: CPU Time Allocation in ASA Concurrent Simulation Method

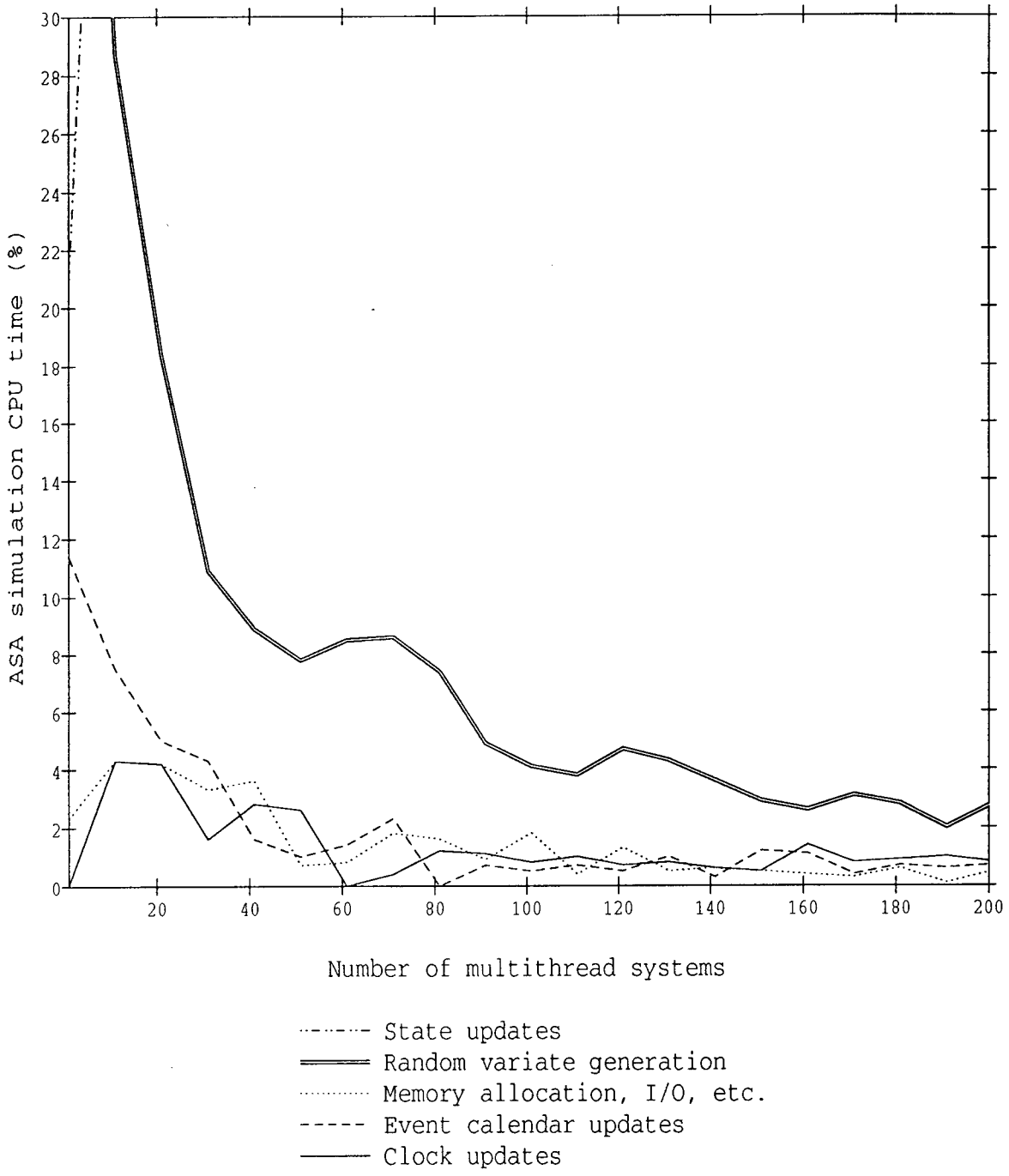


Figure 20: CPU Time Allocation in ASA Concurrent Simulation Method (magnification)

3.4. Software Tools for Concurrent Simulation.

In Section 1.1, the development of interactive capabilities for simulation was identified as a key issue. This requires the development of appropriate software environments for simulation that are only recently beginning to emerge. Although such an effort is not within the scope of this project, some of the concurrent simulation algorithms described above were implemented within the environment of a commercial simulation tool with interactive capabilities. Our objective here was to test the ease of incorporating concurrent simulation methods into existing simulators and test their efficiency in such a setting.

In particular, the results shown in the previous sections were obtained using simulation code that we developed explicitly for the purpose of the efficiency studies. An alternative is to develop such code within an existing simulator. Thus, we adopted the recent commercial simulation tool EXTEND in the Macintosh environment, and developed some simple "objects", i.e., software modules, that can be easily added onto simulation models created through this tool. Our goal here was twofold: (a) Use the resulting software environment for proof-of-concept purposes, and (b) Perform an efficiency study similar to the one reported above in a commercial software simulation setting.

We will limit our results to the simple model considered in Section 3.2, i.e., an M/M/1/k queuing model. The actual EXTEND model is shown graphically in **Figure 21**. In this model, there are two functional blocks used for random variate generation purposes: the one labeled "New customers arrive" and the one marked "Rand" which is used to assign processing times to tasks. The two blocks marked "Exit" are used to count the number of blocked tasks (due to finding a full queue) and the number of departures. The simulation is set up to stop after a given number of departures.

In this effort, we considered only the ASA approach which is much easier to integrate into a conventional simulation tool such as EXTEND. In order to implement the ASA scheme, we created a block referred to as the "ASA block", and connected it to the model of **Figure 21** as shown in **Figure 22**. Specifically, the block was designed to detect every arrival and departure event from the actual model (in which a buffer capacity value is set) and evaluate the state of the system and the numbers of blocked tasks and of departures for any desired number of systems with the buffer capacity value modified.

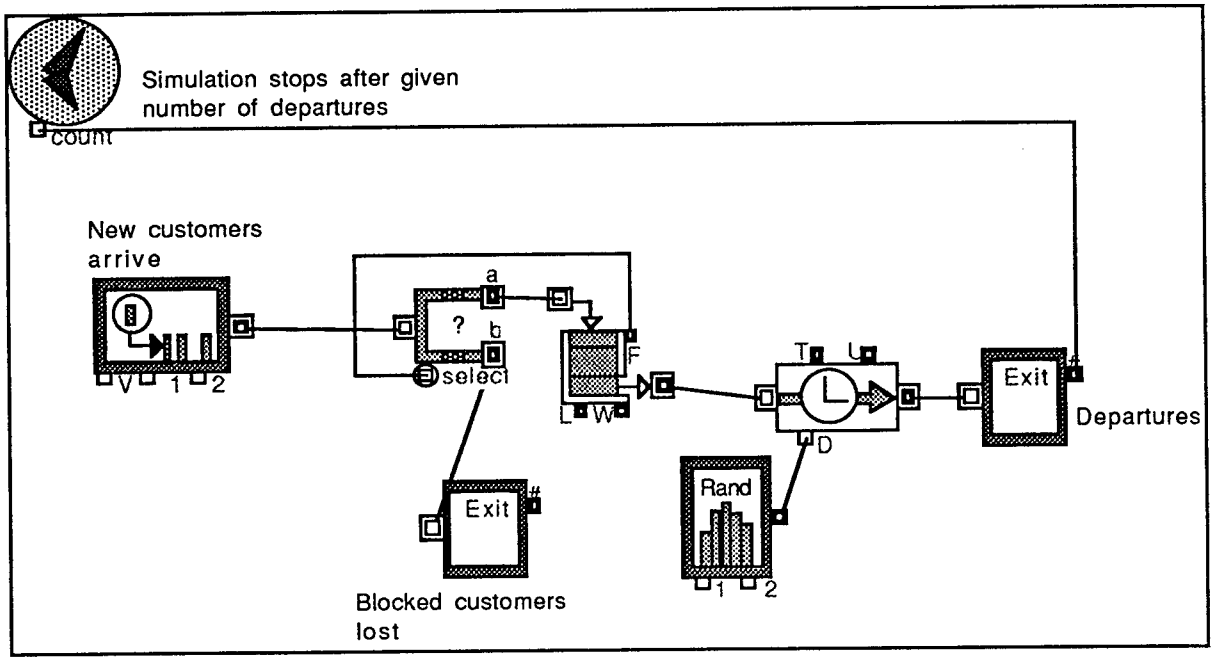


Figure 21. Single Server Model using the EXTEND simulation software

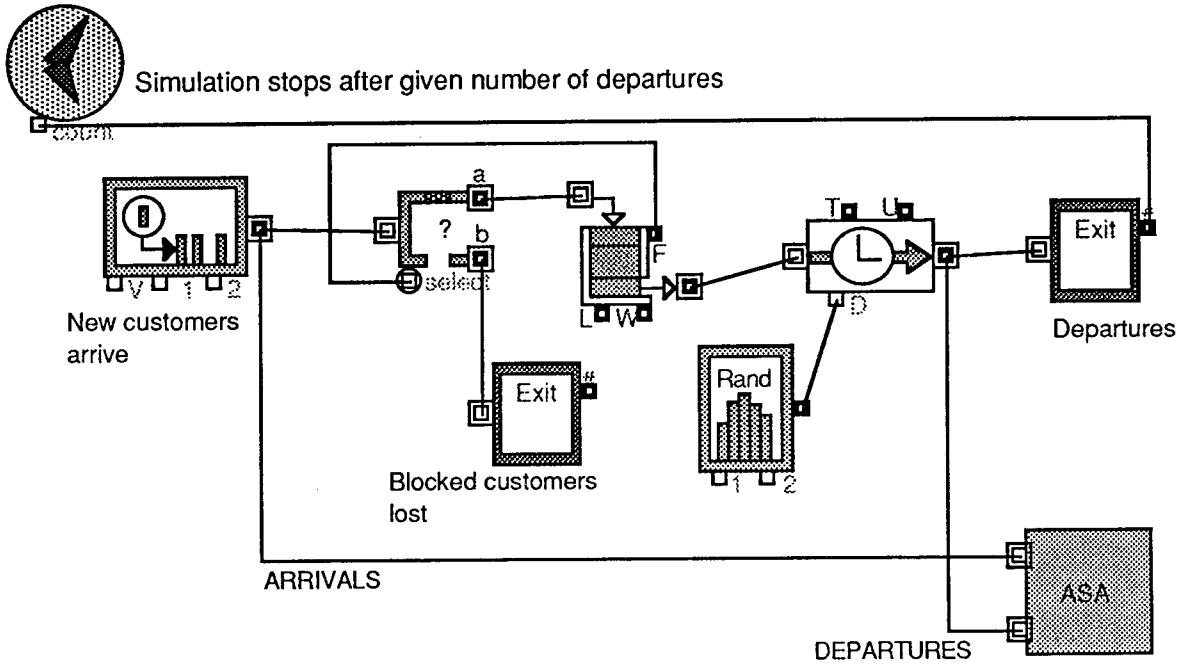


Figure 22. Single Server Model with ASA Block Added for Concurrent Simulation

Two different versions of the ASA block were implemented. In the first, referred to as "ASA1", the ASA procedure used takes advantage of the known structure of the actual system. This "customization" of the ASA block makes it particularly efficient. In the second version, the ASA block (refereed to as "ASA2") was not customized; instead, it was designed to process the arrival and departure events it detected without any additional knowledge of the structure of the system. This obviously reduces its efficiency, but makes it a more general-purpose module one can insert into a large class of models simulated. A few representative results of this study are shown in **Figure 23** (where the total CPU time for the brute force simulation approach and ASA1, ASA2 are shown) and **Figure 24** (where the speedup factors of ASA1 and ASA2 are shown as a function of the degree of concurrency, as in earlier similar plots). In the latter, we can see that the customized ASA algorithm accomplishes higher speedups, as expected.

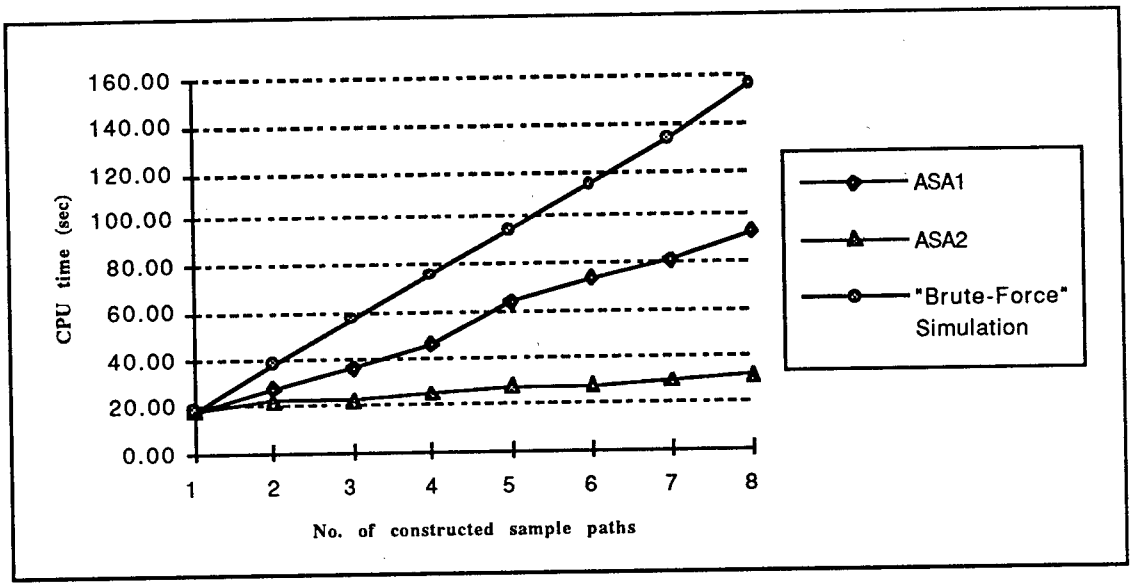


Figure 23. CPU Times for Single Server Model of Figure 21

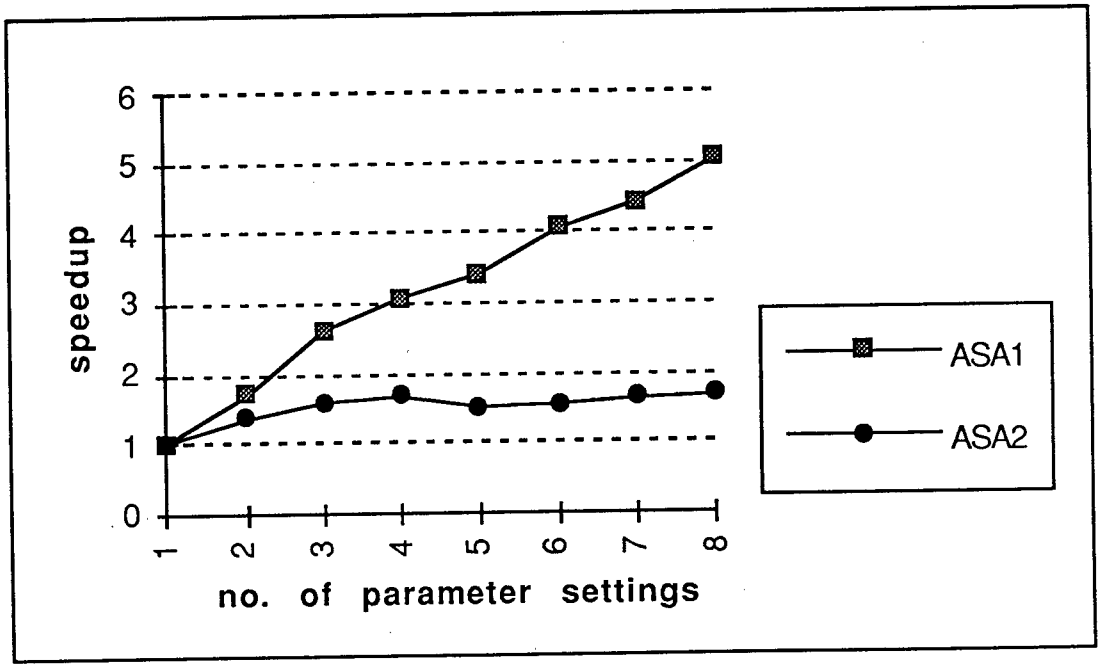


Figure 24. Speedup Factors for Single Server Model of Figure 21

4. OPTIMIZATION SCHEMES USING CONCURRENT SIMULATION.

As mentioned in Section 1.1, the purpose of simulation is often to seek improvements in the design or operational control of a complex system (where analytical tools are simply unavailable), and ultimately optimize it. This requires a computationally intensive process of comparing many alternatives in order to identify the optimal one. In the effort to develop systematic methods for such a process, traditional techniques rely on estimating the *cardinal* values of a performance (or cost) objective function over all possible alternatives. In contrast, *ordinal* optimization is driven by the *relative order* of estimates of the objective function -- not their absolute values. This exploits inherent robustness properties of these order statistics with respect to substantial estimation noise. In practice, this means that short simulation runs may be perfectly adequate to establish an ordering of performance estimates and hence identifying a set of the best alternatives. Combined with the concurrent simulation capabilities described in previous sections, this framework explicitly tackles the problem of complexity by rapidly narrowing down a potentially very large set of alternative designs which are candidates for the global optimum.

Clearly, this principle needs to be put to the test for problems of interest in the C^3I environment, and explicit techniques need to be developed to implement it. This section represents an effort in this direction. In Section 4.1, we will describe the framework of *Ordinal Optimization* (OO) in some more detail and apply it to the testbed problem introduced in Section 3.1. In Section 4.2, we present an explicit optimization scheme, the *Stochastic Comparison* (STC) algorithm, and apply it to the same problem. Finally, in Section 4.3, we present another scheme, the *Stochastic Descent* (SD) algorithm, specifically developed for resource allocation problems, which we also apply to the testbed problem.

4.1. Ordinal Optimization Basics.

For ease of exposition, let us recall the problem introduced in Section 3.1:

$$\text{find } \mathbf{b}^* \in B \text{ to minimize } J(\mathbf{b}) = E[L(\mathbf{b})] \qquad \textbf{(P)}$$

where the vector $\mathbf{b} = [b_1,\ldots,b_N]$ represents an "allocation", b_i is the number of buffer slots at branch i in **Figure 4**, and $b_1+\ldots+b_N = K$. Here, B is the set of all possible allocations. For the problem used in our numerical example, we use 6 parallel branches and $K = 20$ buffer slots, which results in a search space with $|B| = 53{,}130$ possible allocations. Therefore, in a simulation-based approach for solving **(P)**, 53,130 simulations have to be performed, each of which may be quite long to obtain a sufficiently accurate estimate of $J(\mathbf{b})$.

In ordinal optimization one possible starting point is to consider the set of all possible performance values $J(\mathbf{b})$ for all $\mathbf{b} \in B$ and define a subset G_g of B to contain the top $g/|B|$ values of $J(\mathbf{b})$ (see also [4],[21]). This is referred to as the set of "good enough" performance points, thus softening the original requirement for determining a point $\mathbf{b}^*$. Next, the search space is uniformly sampled and M elements selected. The corresponding performance values are estimated (through simulation) and the estimates are ordered. A set S_h, referred to as the "selected" set is defined by the points that give the top h values of these estimates. Now, given an integer k, $k \in \{1,\ldots,h\}$, the *alignment probability* $P(|G_g \cap S_h| \geq k)$ is the probability that at least k elements in the selected set S_h belong to the "good enough" set G_g. Because of the robustness of order statistics with respect to the estimation noise involved, it is often the case that short sample paths can provide alignment probabilities very close to 1 for small values of k. The idea is that if one can quickly narrow down a search to a small set in which at least one "good enough" point exists with high probability, then the optimization process can proceed within S_h to determine the optimal point within this smaller set. Clearly, by adjusting the parameters g, M, and h, one can obtain different values of $P(|G_g \cap S_h| \geq k)$, $k = 1,\ldots,h$. Depending on the confidence level required, given by $P(|G_g \cap S_h| \geq k)$, and the desired proximity to optimality, given by g, the ordinal optimization approach aims at identifying points in B with sufficiently high $P(|G_g \cap S_h| \geq k)$, $k = 1,\ldots,h$ for a given g.

In this section, we illustrate the principles of this approach through our testbed problem. We select a value for M to uniformly sample over the search space of all allocations B. We then exploit concurrent simulation (in this case, we use ASA) to simultaneously obtain all M resulting estimates of a given performance measure $J(\mathbf{b})$ using simulation. We thus obtain a selected set S_h and simply identify the estimate with the smallest value. Our numerical results are for the same model as the one specified in Section 3.3: the arrival rate considered was fixed at $\lambda = 10$ tasks/sec. Each arriving task is routed to the ith branch with probability c_i, $i = 1,\ldots,6$. In particular: $c_1 = 0.1$, $c_2 = 0.2$, $c_3 = 0.1$, $c_4 = 0.2$, $c_1 = 0.2$, $c_1 = 0.2$. The service rates at the branches were set to $\mu_1 = 3$ tasks/sec, $\mu_2 = 2$ tasks/sec, $\mu_3 = 2$ tasks/sec, $\mu_4 = 3$ tasks/sec, $\mu_5 = 3$ tasks/sec, $\mu_6 = 2$ tasks/sec. We have limited ourselves to a Markovian model so as to be able to solve problem **(P)** analytically, hence checking the validity of our approach. By selecting the performance measure $J(\mathbf{b})$ to be the total blocking probability of tasks, the optimal allocation $\mathbf{b}^*$ was determined by evaluating the steady-state blocking probability of the corresponding Markov chain over all 53,130 possible values of the vector $\mathbf{b}$. We found that $\mathbf{b}^* = [2, 4, 2, 4, 4, 4]$ and $J(\mathbf{b}^*) = 0.095081$. The worst allocation was found to be [0, 0, 20, 0, 0, 0] with corresponding performance 0.393333.

Following the discussion above, we first selected 1000 allocations at random (out of 53,130). In **Table 1**, the last column shows the top-10 allocations among these and their performance

obtained from the analytical model. By using the same ASA concurrent simulation algorithm as in Section 3.3, we then simulated all 1000 systems and recorded performance estimates as a function of the simulation run length. The first seven columns in **Table 2** show our results for simulation run lengths determined by the total number of events counted, varying from only 10 to 10^7. In each column, the top-10 allocations determined by the concurrent simulations are shown. The number in parenthesis is the actual order of the corresponding allocation analytically determined. Note that when the runs are so short that very limited information is actually available, the performance estimates are useless in a cardinal sense; however, one can still use them to create an ordering (in case of identical estimates, the order is determined randomly). We can observe that with simulation runs as short as 100 events, one can still determine some of the best allocations (all but one within the top 10%), including the second best one showing as the sixth estimated best in the list. With 10^6 events, all top-10 allocations are identified in the correct order, except for the 7th one (this is just a random effect with no special significance to be attributed to the fact that it is the 7th ordered allocation).

The process above started out by limiting the search space from 53,130 to a samples subspace of 1000 allocations. In **Table 2**, we show results where this is no longer the case. Instead, the last column displays the *actual* order of the top-10 allocations among the 1000 samples. Note that this random sampling by itself has identified 10 allocations in the top 0.18% of all possible choices. Then, note that even with as few as 100 events we can identify an allocation in the top 3.2% of all choices (the 1680th one). With 10^5 events, we can identify the one in the top 0.18% (the 52nd one).

To better visualize the speed with which ordering of estimates becomes accurate, we have plotted the relationship between *actual order* of an allocation and *estimated order* (as determined through the ASA concurrent simulation algorithm). In **Figure 25**, this is shown for simulation run lengths of only 10 events over the subspace of 1000 allocations used in **Table 1**. Thus, if a point in this plot has coordinates, for example, (10, 100) this means that it gives an allocation estimated to be the 10th best, but is actually the 100th best. Obviously, if all points were to form a line with a slope of 1, this would give the ideal situation where all estimated orders correspond to actual orders; we shall refer to it as the "ideal alignment line". In **Figure 25**, where the run length is extremely short (10 events), the points appear randomly scattered. In **Figure 26**, the run length is still, by all standards, very short (1000 events), but one can already see an alignment being formed around the "ideal alignment line". In **Figure 27**, the run length is increased to 10,000 events and all points are clearly aggregated around the "ideal alignment line". **Figures 28-29** correspond to run lengths of 10^6 events and 10^7 events respectively. **Figure 30** is a magnification

of **Figure 29** that indicates an almost perfect order alignment over the top 50 allocations.

In summary, by exploiting concurrent simulation techniques and the speed with which ordering alignments take place, one can rapidly reduce an extremely large search space to a much smaller set within which the optimal point lies with high probability. As already mentioned, this involves a softening of the original hard optimization problem **(P)**: determining an allocation within the top $x\%$ with probability y. One can then select as small a value of x and as large a value of y to trade off accuracy with computational complexity. An important aspect of the OO framework is the explicit quantification of the relationship between x and y. Preliminary results indicate that the strength of OO lies in the fact that by softening the optimization requirement by a very small amount (i.e., specifying a very small value of x), the corresponding value of y can be kept very close to 1. This analysis is outside the scope of this project and is currently the subject of ongoing research.

Table 1

Performance Estimates Ordering vs. Observation Duration

Top 10 designs of a set of 1,000 randomly selected ones

Observation duration (as the number of events in the sample path)							Analytical values
10	100	1,000	10,000	100,000	1,000,000	10,000,000	
.000000 (188)	.148148 (188)	.113208 (33)	.092695 (4)	.094331 (1)	.098223 (1)	.098402 (1)	.098738 (1)
.000000 (413)	.148148 (413)	.113208 (97)	.092695 (1)	.095057 (2)	.098281 (2)	.098575 (2)	.098950 (2)
.000000 (59)	.148148 (59)	.113208 (12)	.092695 (6)	.095093 (3)	.098538 (3)	.098780 (3)	.099121 (3)
.000000 (319)	.148148 (319)	.113208 (13)	.093059 (2)	.095637 (7)	.099109 (4)	.099397 (4)	.099730 (4)
.000000 (512)	.148148 (589)	.113208 (94)	.094512 (12)	.095818 (4)	.099109 (6)	.099397 (6)	.099730 (5)
.000000 (589)	.148148 (592)	.113208 (2)	.094512 (5)	.095818 (6)	.099145 (5)	.099414 (5)	.099730 (6)
.000000 (842)	.148148 (100)	.113208 (64)	.094875 (28)	.096036 (5)	.099297 (8)	.099664 (8)	.100073 (7)
.000000 (592)	.148148 (268)	.113208 (4)	.095238 (13)	.096362 (10)	.099297 (9)	.099664 (9)	.100075 (8)
.000000 (100)	.148148 (552)	.113208 (117)	.095238 (8)	.096507 (8)	.099423 (7)	.099702 (7)	.100075 (9)
.000000 (268)	.148148 (82)	.113208 (17)	.095238 (9)	.096507 (9)	.099477 (10)	.099743 (10)	.100092 (10)

(1) The number in parentheses is the corresponding analytical ordering of that design in the set of 1,000 selected designs.

(2) If there are two or more designs with a same performance estimate, their ordering is then determined by pre-assigned random attibute associated with each design.

Table 2

Performance Estimates Ordering vs. Observation Duration

Top 10 designs of a set of 1,000 randomly selected ones

Observation duration (as the number of events in the sample path)							Analytical values
10	100	1,000	10,000	100,000	1,000,000	10,000,000	
.000000 (1680)	.148148 (1680)	.113208 (243)	.092695 (84)	.094331 (52)	.098223 (52)	.098402 (52)	.098738 (52)
.000000 (5645)	.148148 (5645)	.113208 (717)	.092695 (52)	.095057 (58)	.098281 (58)	.098575 (58)	.098950 (58)
.000000 (435)	.148148 (435)	.113208 (120)	.092695 (84)	.095093 (70)	.098538 (70)	.098780 (70)	.099121 (70)
.000000 (3541)	.148148 (3541)	.113208 (130)	.093059 (58)	.095637 (91)	.099109 (84)	.099397 (84)	.099730 (84)
.000000 (8189)	.148148 (11142)	.113208 (687)	.094512 (120)	.095818 (84)	.099109 (84)	.099397 (84)	.099730 (84)
.000000 (11142)	.148148 (11238)	.113208 (58)	.094512 (85)	.095818 (84)	.099145 (85)	.099414 (85)	.099730 (85)
.000000 (24168)	.148148 (738)	.113208 (474)	.094875 (214)	.096036 (85)	.099297 (94)	.099664 (94)	.100073 (91)
.000000 (11238)	.148148 (2653)	.113208 (84)	.095238 (130)	.096362 (96)	.099297 (95)	.099664 (95)	.100075 (94)
.000000 (738)	.148148 (9593)	.113208 (871)	.095238 (94)	.096507 (94)	.099423 (91)	.099702 (91)	.100075 (95)
.000000 (2653)	.148148 (635)	.113208 (141)	.095238 (95)	.096507 (95)	.099477 (96)	.099743 (96)	.100092 (96)

(1) The number in parentheses is the corresponding analytical ordering of that design in the whole population of 53,130 designs.

(2) If there are two or more designs with a same performance estimate, their ordering is then determined by pre-assigned random attibute associated with each design.

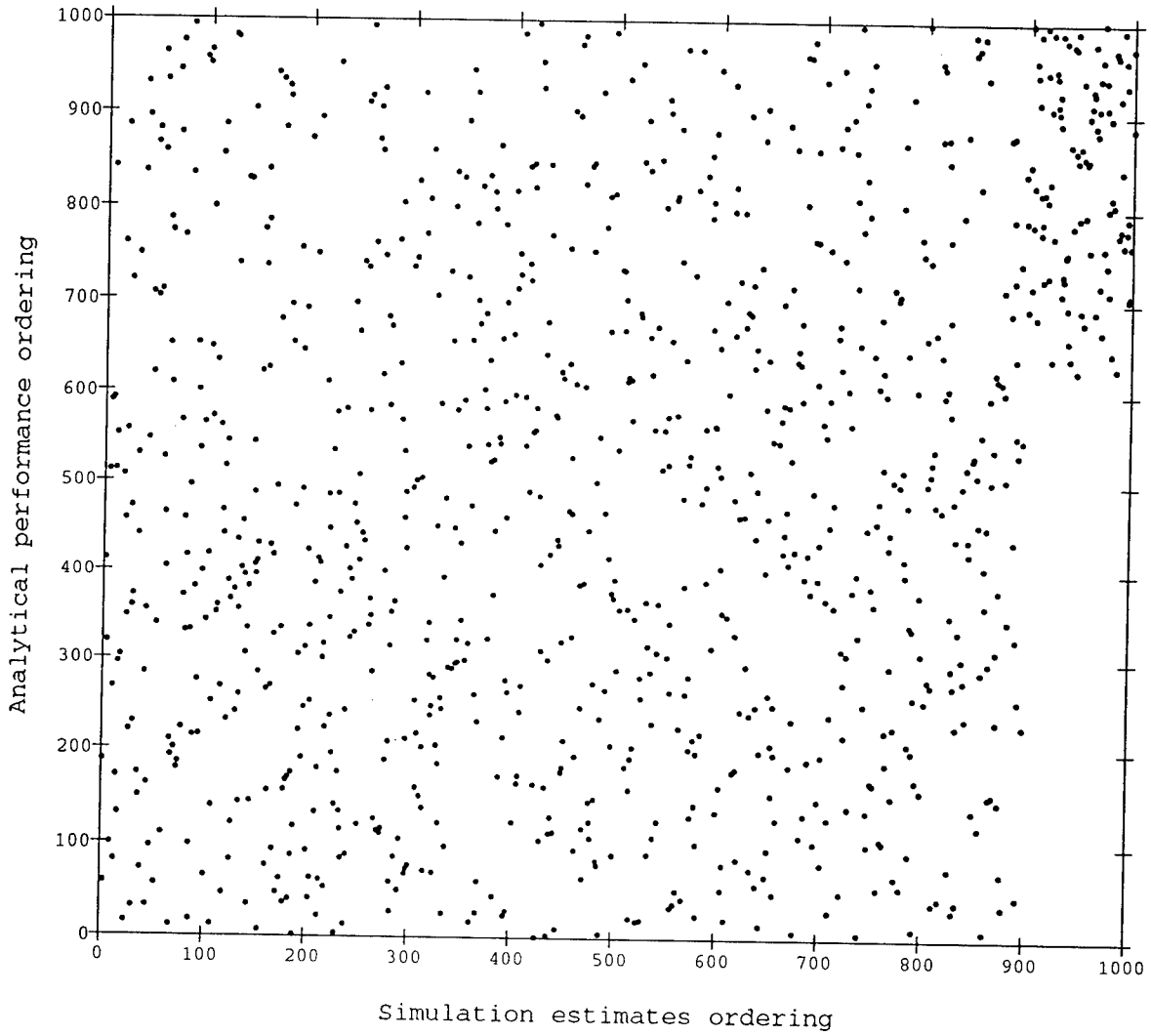


Figure 25: Actual vs. Estimated Performance Ordering of Top 1000 Allocations (simulation run length = 10 events)

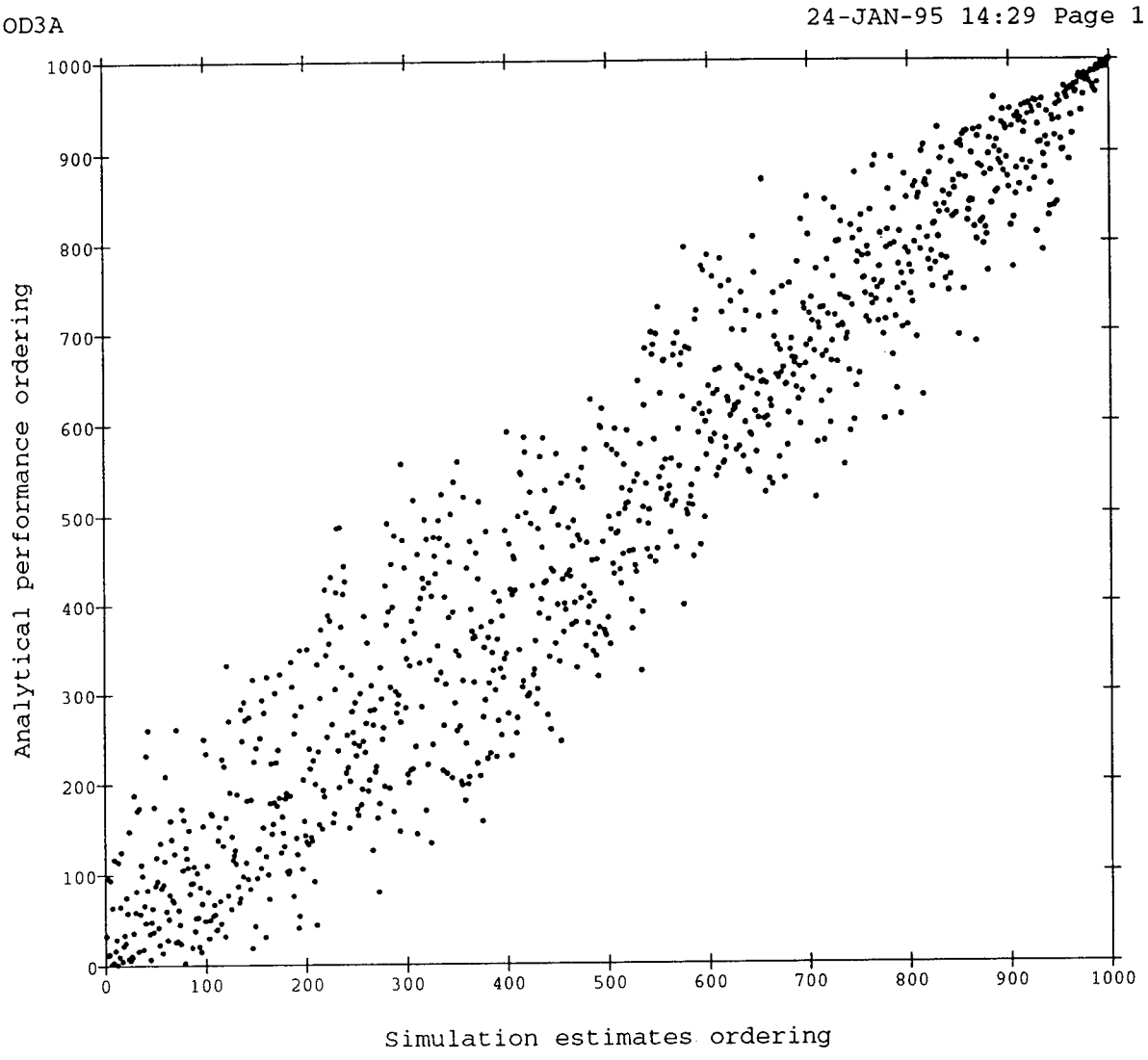


Figure 26: Actual vs. Estimated Performance Ordering of Top 1000 Allocations (simulation run length = 10^3 events)

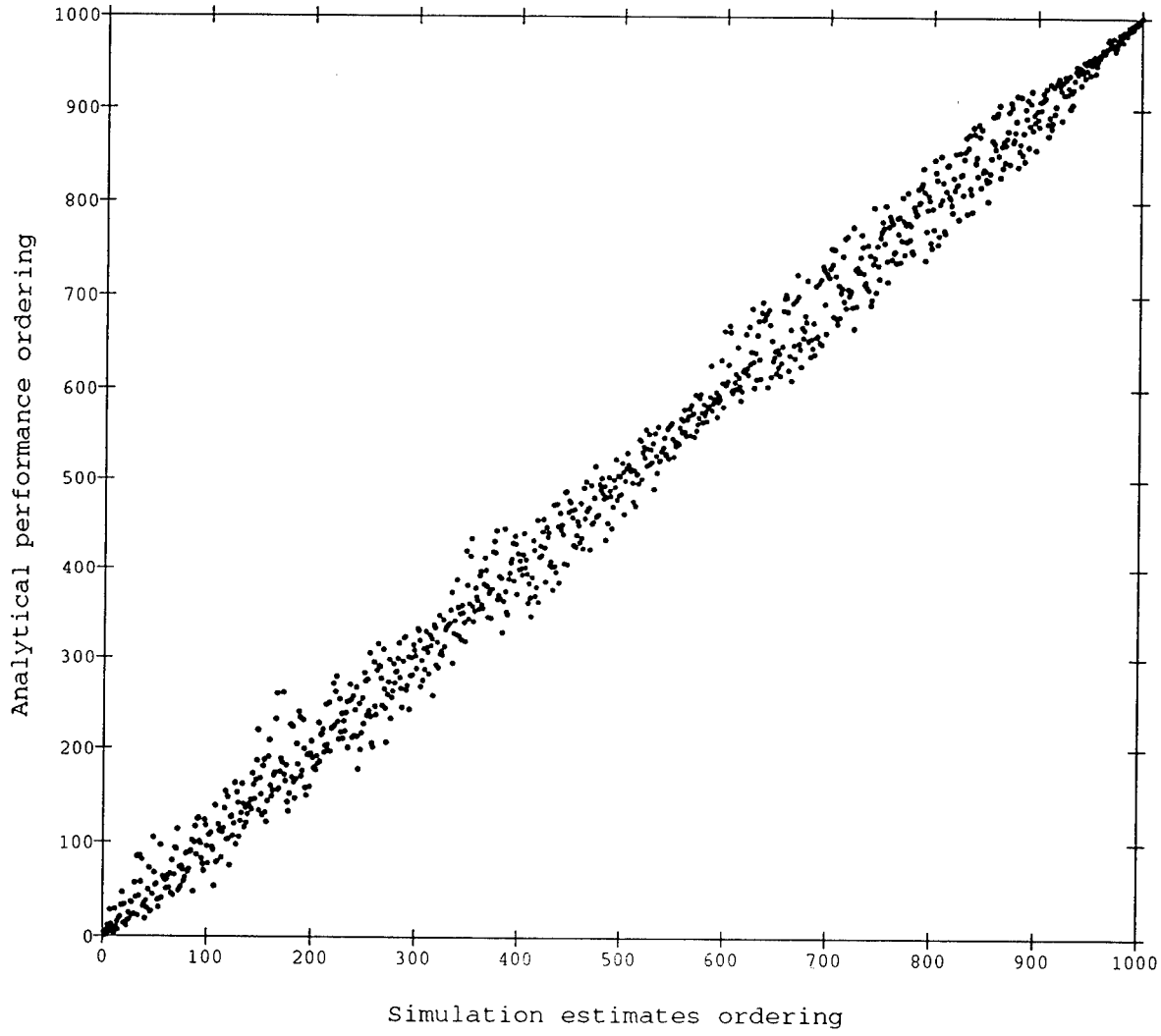


Figure 27: Actual vs. Estimated Performance Ordering of Top 1000 Allocations (simulation run length = 10^4 events)

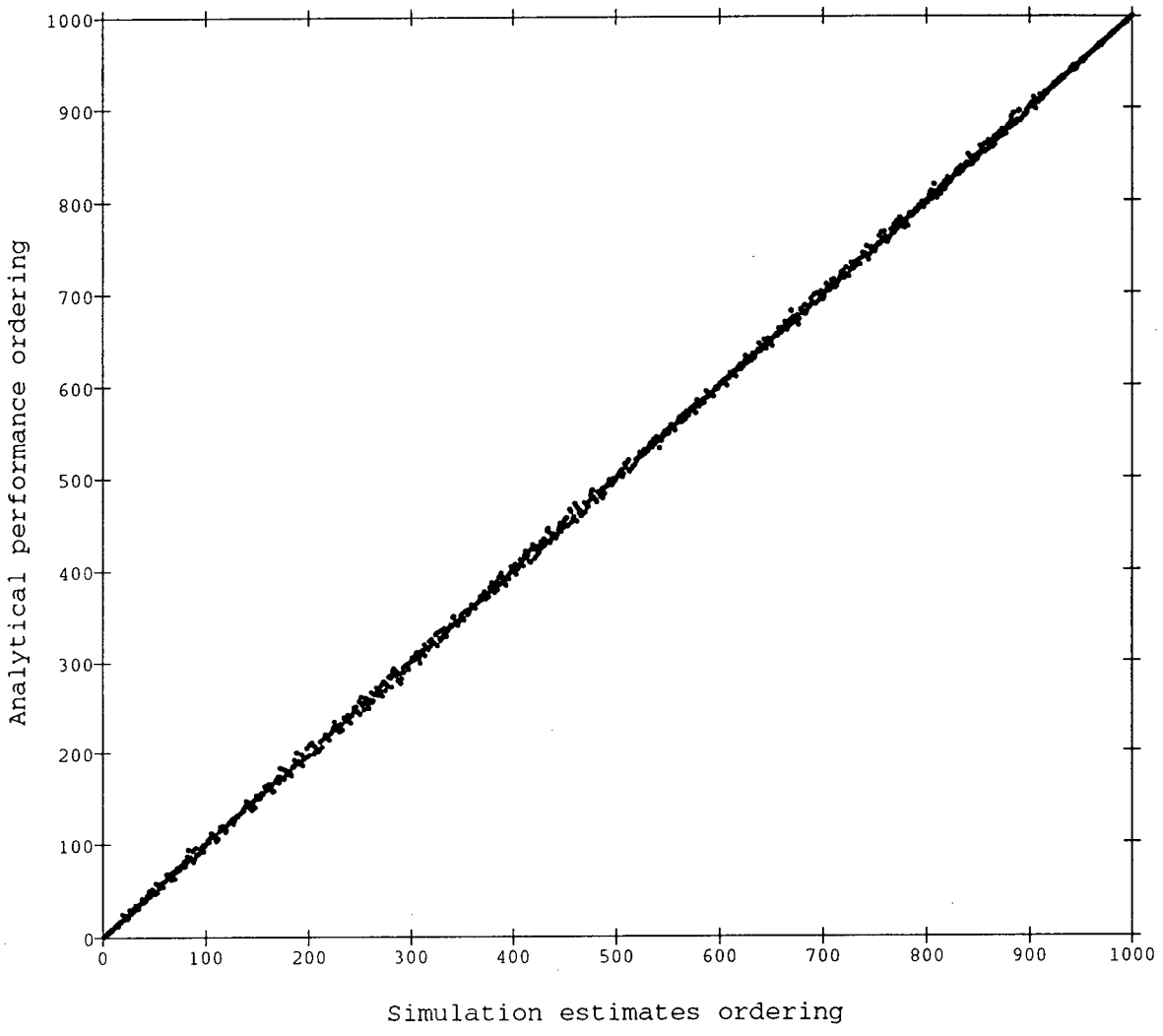


Figure 28: Actual vs. Estimated Performance Ordering of Top 1000 Allocations (simulation run length = 10^6 events)

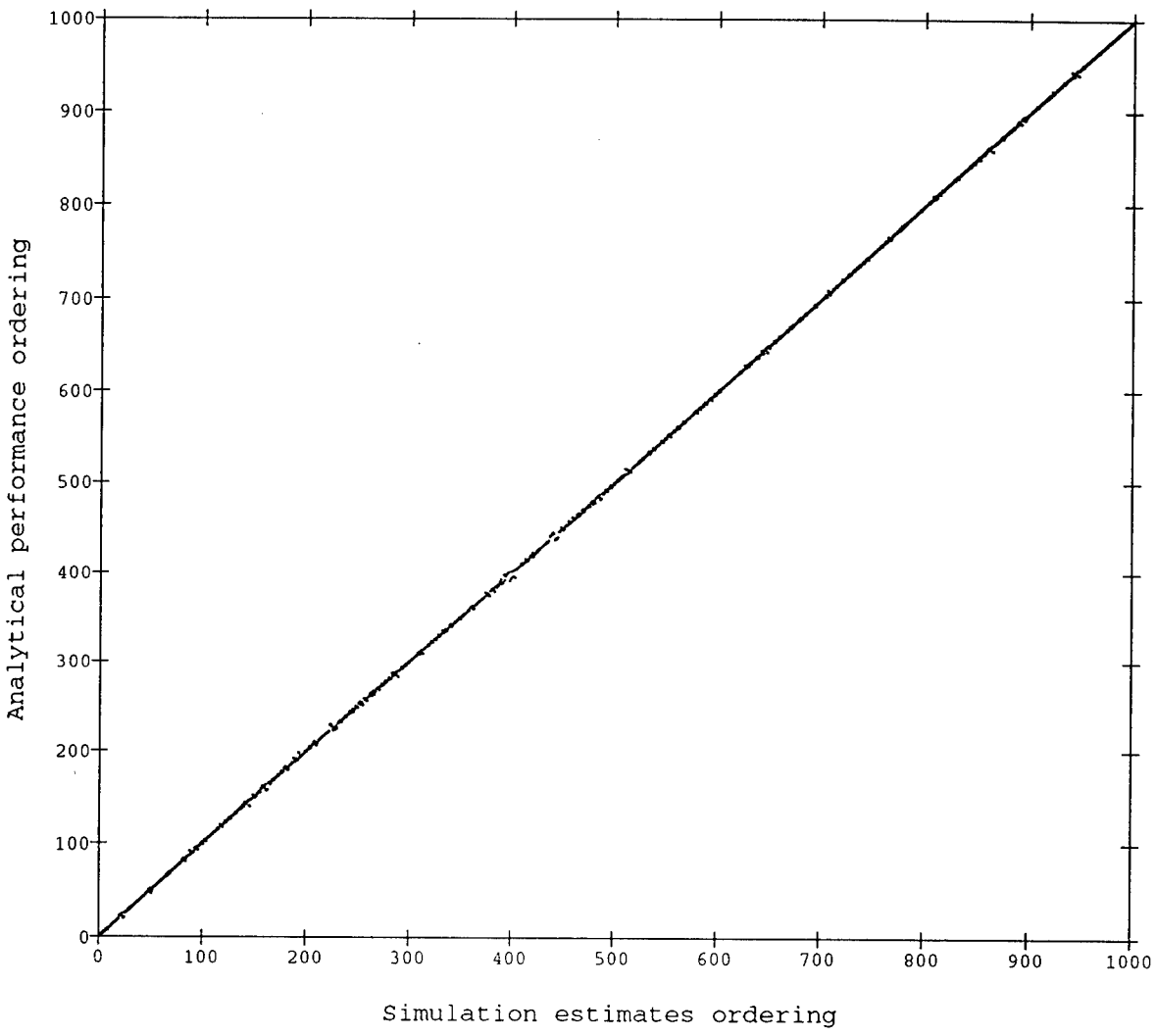
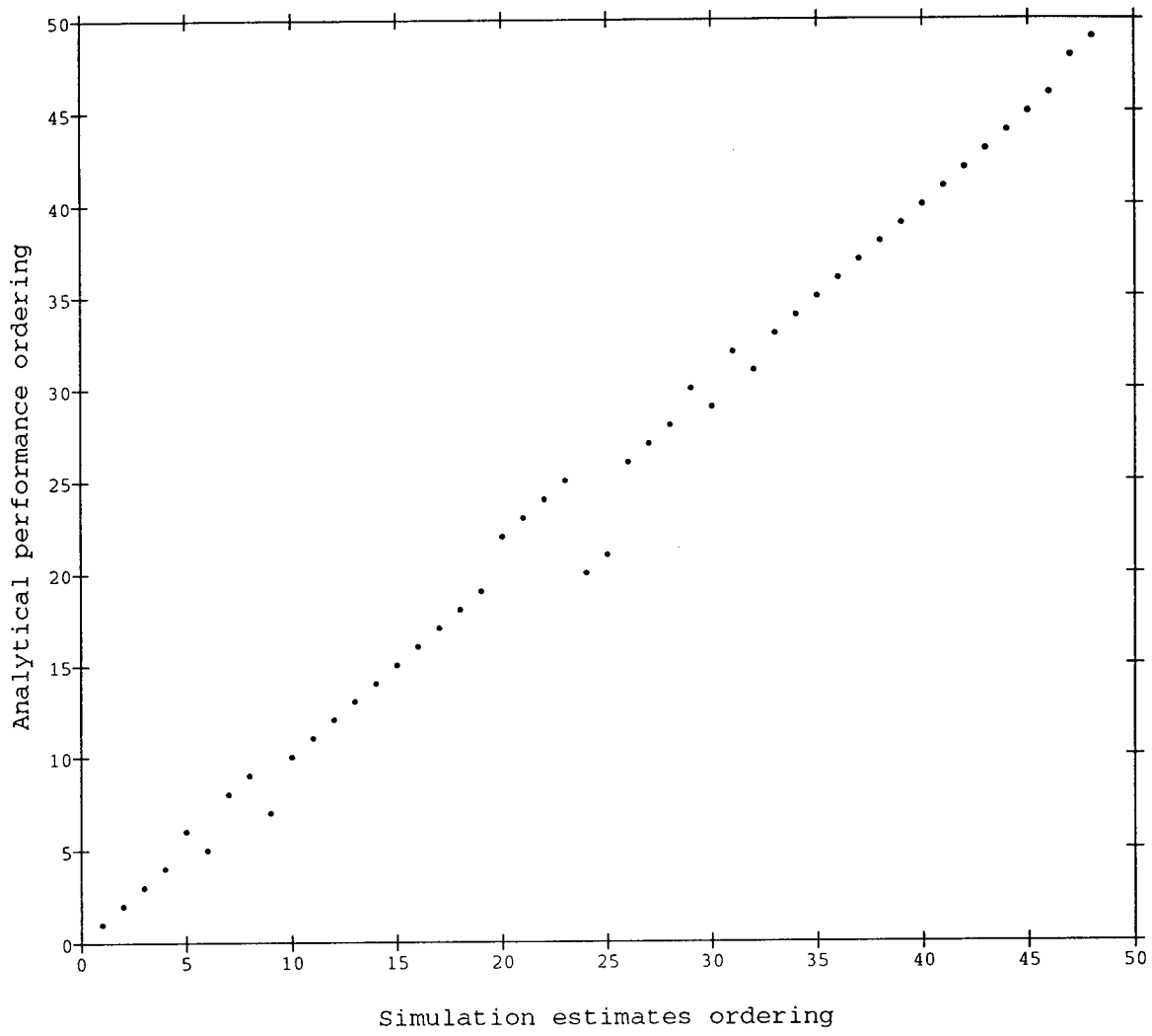


Figure 29: Actual vs. Estimated Performance Ordering of Top 1000 Allocations (simulation run length = 10^7 events)



Each estimate is based on observation of a sample path of 10,000,000 events

Figure 30: Magnification of Figure 29 for the Top 50 Allocations (simulation run length = 10^7 events)

4.2. Stochastic Comparison Algorithms.

The OO approach discussed in the previous section is based on a softening of the requirement for the determination of the global optimum of a problem. In this and the next section we will discuss two schemes which, while exploiting the OO principles and concurrent simulation, still provably guarantee the determination of a global optimum for problem **(P)**.

The first scheme is based on the Stochastic Comparison (STC) algorithm proposed in [22] (see also [42]). For our problem, this algorithm iterates on allocations, where the allocation vector at iteration k is denoted by $\mathbf{x}_k$ and allocation vectors in B are denoted by $\mathbf{b}_i$, $i = 1,\ldots,|B|$:

Stochastic Comparison Algorithm

1. Initialize: $\mathbf{x}_0 = \mathbf{b}_i$, and $k = 0$

2. Given $\mathbf{x}_k = \mathbf{b}_i$, choose a next candidate allocation $\mathbf{z}_k$ from B-$\{\mathbf{b}_i\}$ with probabilities $R(i,j)$, $\mathbf{b}_j \in B$-$\{\mathbf{b}_i\}$

3. For a selected $\mathbf{z}_k = \mathbf{b}_j$, set

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{z}_k & \text{with prob. } p_k \\ x_k & \text{with prob. } (1 - p_k) \end{cases}$$

where $p_k = [\mathrm{P}[L(\mathbf{b}_j) < L(\mathbf{b}_i)]]^{M_k}$

4. Replace k by k+1 and go to step 2.

In step 3, the probability p_k is actually not calculable, since we do not know the underlying probability functions. However, it is realizable in the following way: both $J(\mathbf{b}_i)$ and $J(\mathbf{b}_j)$ are estimated M_k times. If $L(\mathbf{b}_j) < L(\mathbf{b}_i)$ every time, then we set $\mathbf{x}_{k+1} = \mathbf{z}_k$. Otherwise, we set $\mathbf{x}_{k+1} = \mathbf{x}_k$. This corresponds to M_k independent "trials" where p_k is the probability of M_k successes. It can be shown [22] that this algorithm converges to an optimal allocation as long as the sequence $\{M_k\}$ increases logarithmically.

The *Generalized* STC (G-STC) algorithm that we developed is a modification of the procedure above (see [23]), with the single candidate allocation $\mathbf{z}_k$ in step 2 replaced by a *set of allocations*, $C_k = \{\mathbf{b}_j : \mathbf{b}_j \in B\text{-}\{\mathbf{b}_i\}\}$. Using a concurrent simulation technique, a set of estimates $\{L(\mathbf{b}_j)$: $\mathbf{b}_j \in C_k\}$ is concurrently obtained. These estimates are ordered and the smallest (assuming we are minimizing) is selected as the candidate new allocation $\mathbf{z}_k$ used as before in step 3.

To place the STC and G-STC algorithms in the context of related work in the field of stochastic optimization, we note that they are significantly different from algorithms such as Simulated Annealing [43], which relies on accurate evaluation of a performance measure, and closer in nature to the Stochastic Ruler algorithm in [44]. For more details see [22],[41].

The STC and G-STC algorithms were applied to our testbed problem. The concurrent simulation technique used was the ASA Event Matching algorithm of Section 2.4.2. Results are shown in **Figure 31**, where the optimal performance value (analytically determined) is also indicated, as a function of allocation updates with 500 events observed between updates. Note that both algorithms rapidly approach the optimal performance, but they generally take a long time to converge to it. This is typical behavior of such algorithms as also reported in [22],[24].

An interesting question that arises is "how can we compare the performance of the STC vs. G-STC algorithm?" In **Figure 31**, it appears that G-STC generally converges faster, but a way of quantifying this behavior is not obvious. A reasonable measure of the performance of such an iterative algorithm is the *area under the corresponding curve*, which represents the *accumulated cost* of the algorithm relative to the optimal performance. Using this metric, **Figure 32** shows the accumulated costs corresponding to **Figure 31** for the two algorithms, which clearly shows the improvement generated through G-STC.

Another way to evaluate the behavior of the two algorithms is shown in **Figure 33**. Here, on the vertical axis we show allocations ordered from the best (at the bottom) to the 53,130th (worst), with only the top 8500 allocations shown. Starting with a random initial allocation, we have plotted the updates performed by the STC and G-STC algorithms. Observe that within approximately 500 updates (equivalently: 250,000 events), both algorithms determine one of the top-100 allocations. A magnification of this plot is shown in **Figure 34**, where we can see that the G-STC algorithm determines one of the top-20 (i.e., within 0.03% of the optimal) allocations.

Since the STC and G-STC algorithms depend on a randomly selected initial allocation and on the randomness of the simulations whose data they use to perform allocation updates, we have repeated the process above three additional times, each time showing the performance of the two algorithms and accumulated costs. The results shown in **Figures 35-37** are similar to those of **Figures 31-32**.

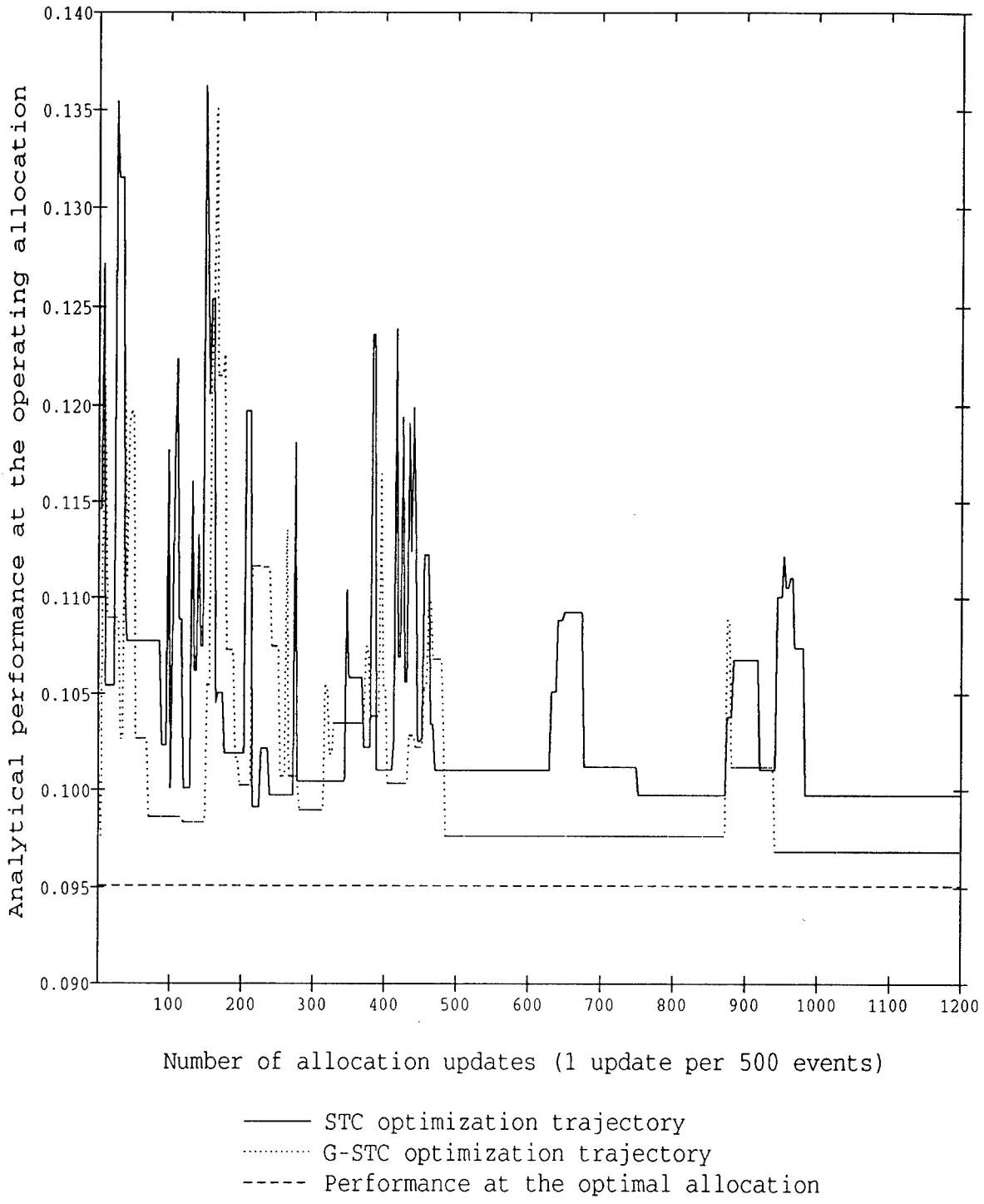


Figure 31: STC and G-STC Algorithms: Convergence to Optimal Performance

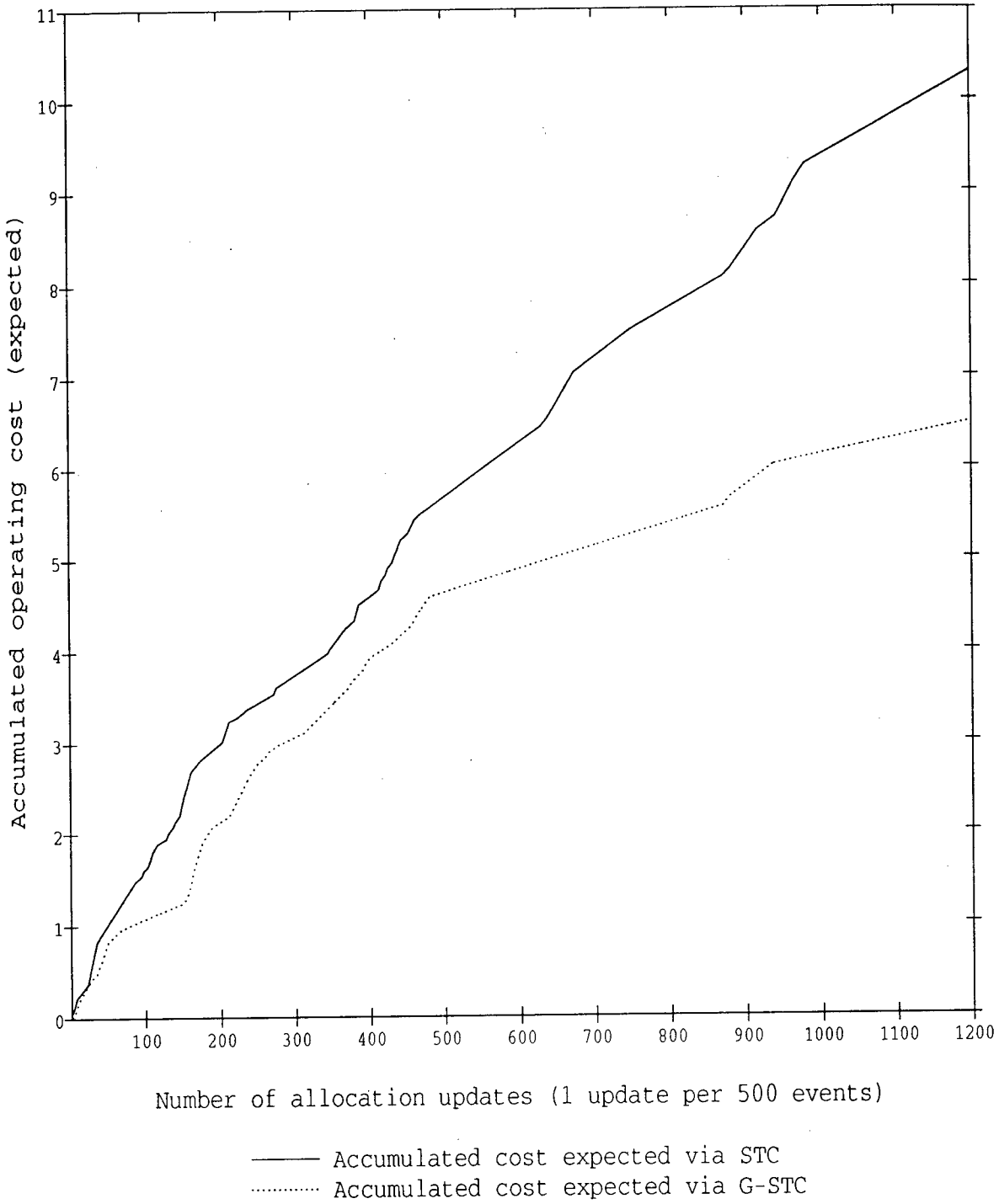


Figure 32: STC and G-STC Algorithms: Accumulated Coast Comparison

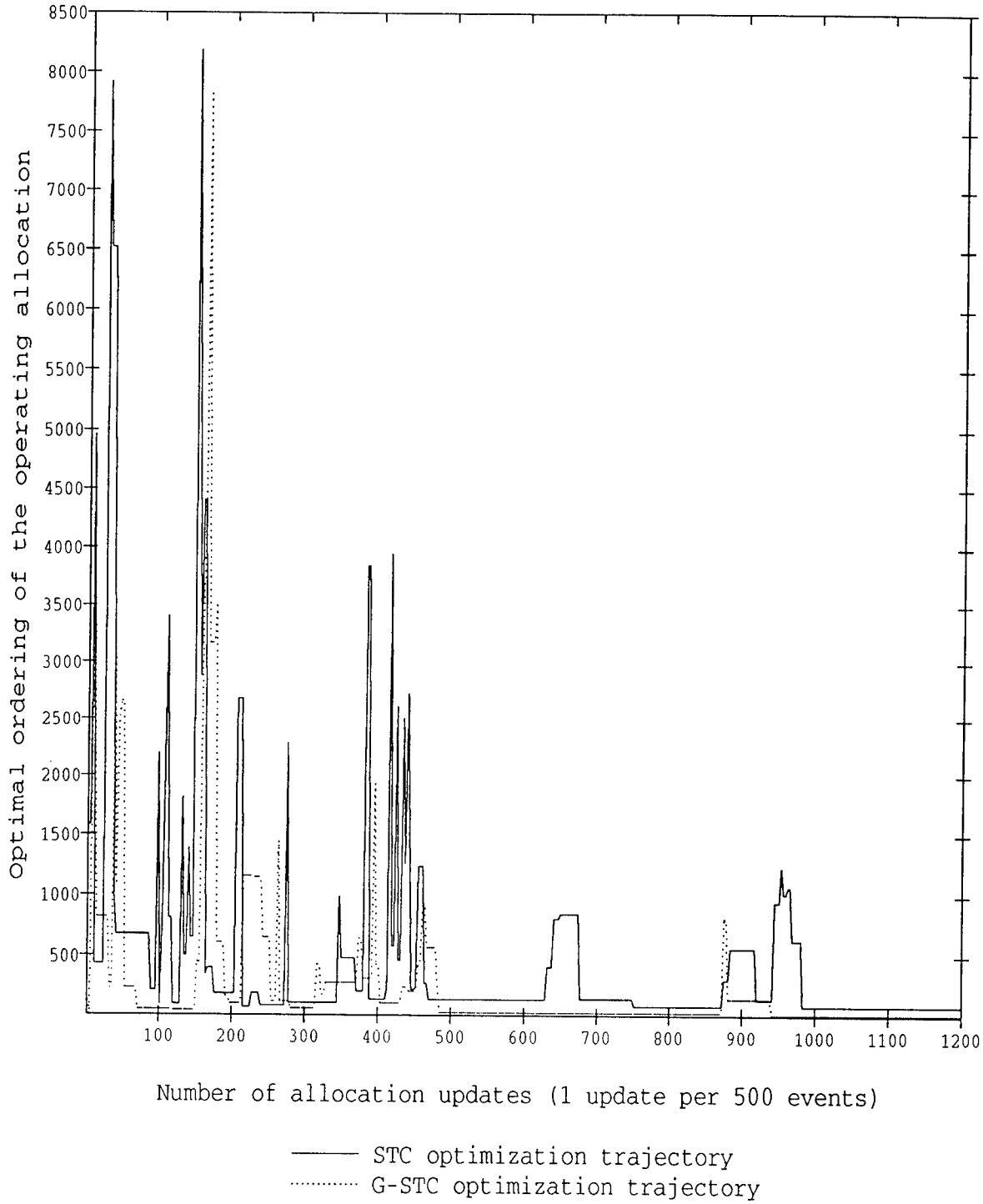


Figure 33: STC and G-STC Algorithms: Convergence to Optimal Allocation

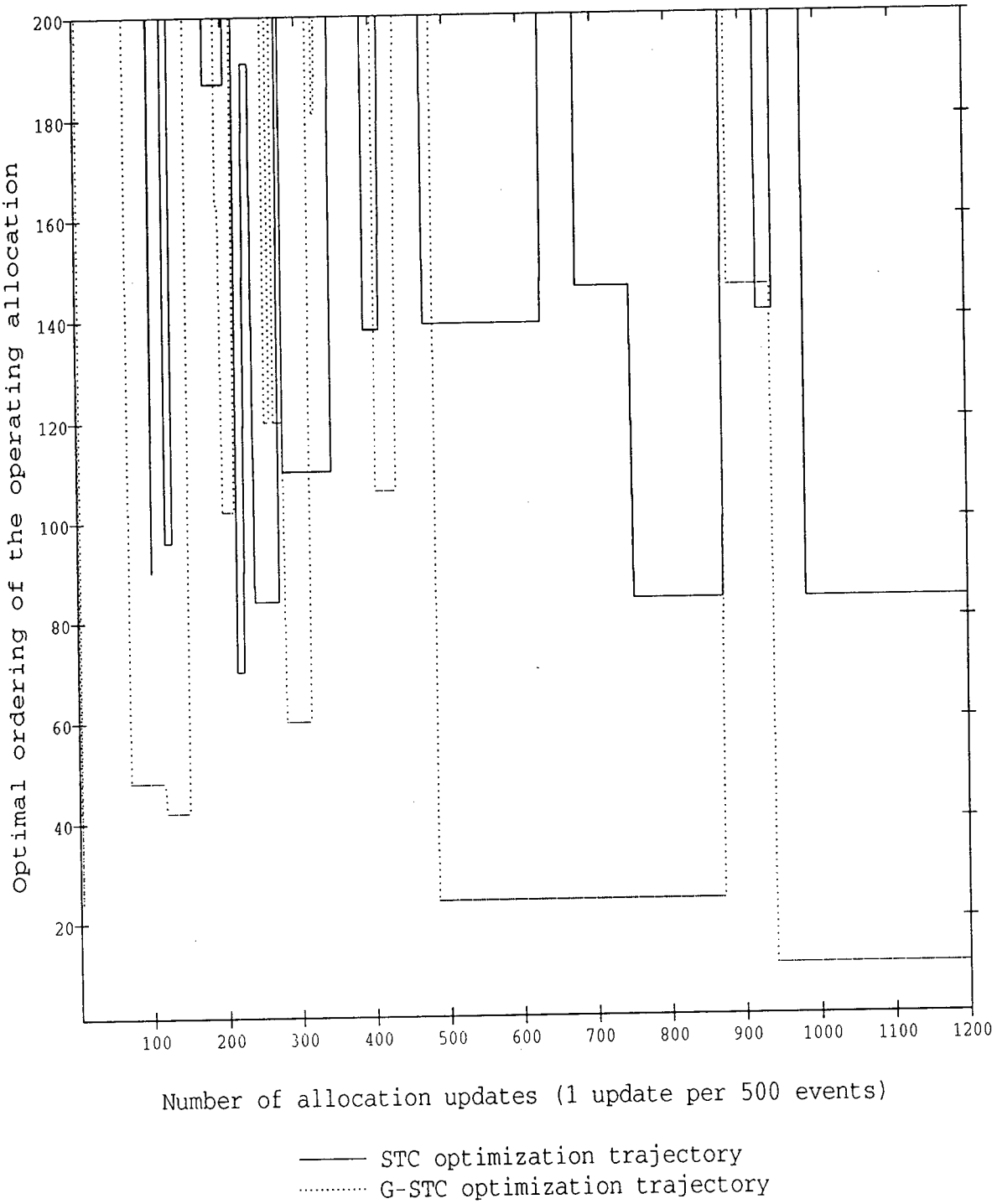


Figure 34: Magnification of Figure 33

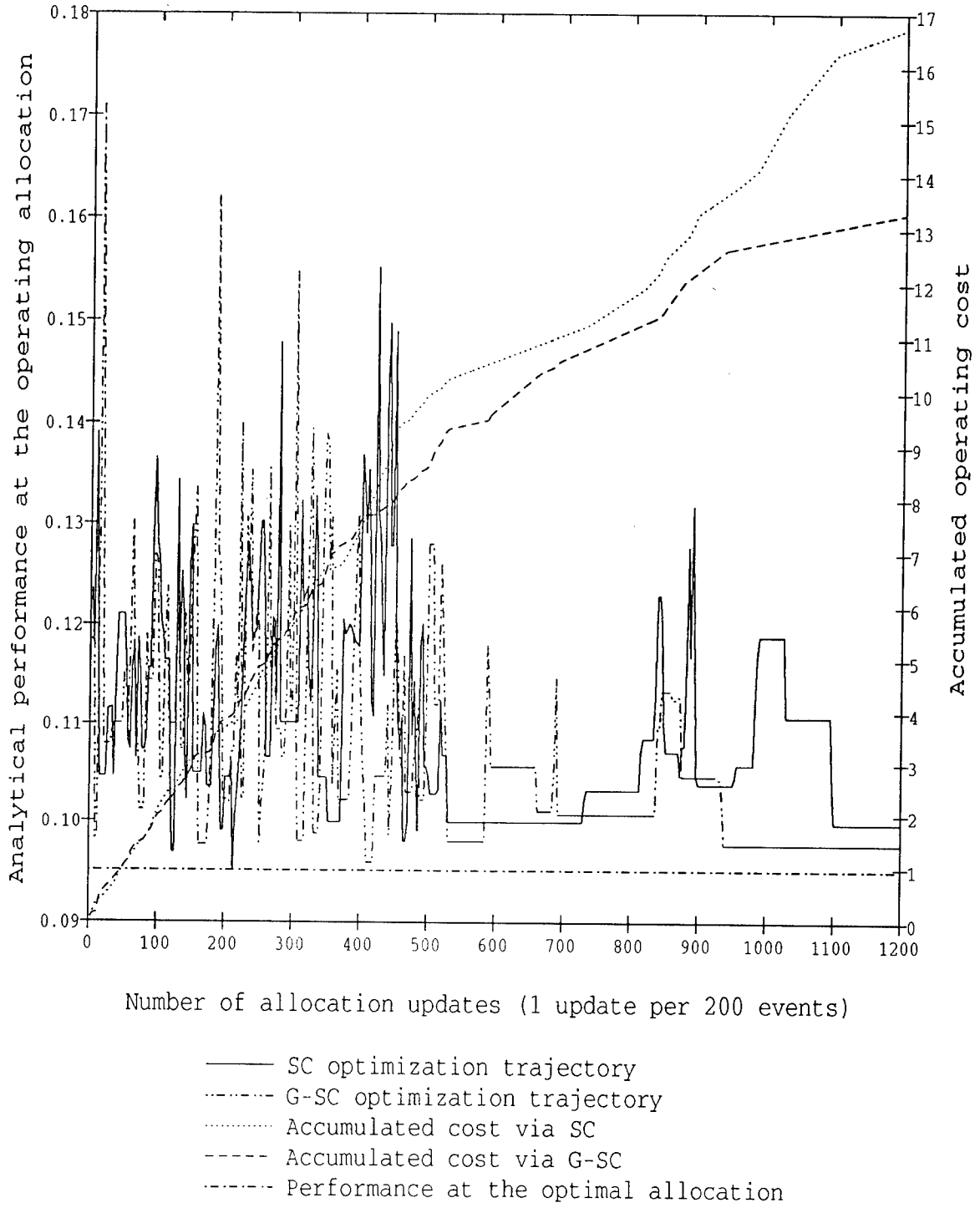


Figure 35: STC and G-STC Algorithms (Simulation Experiment 2)

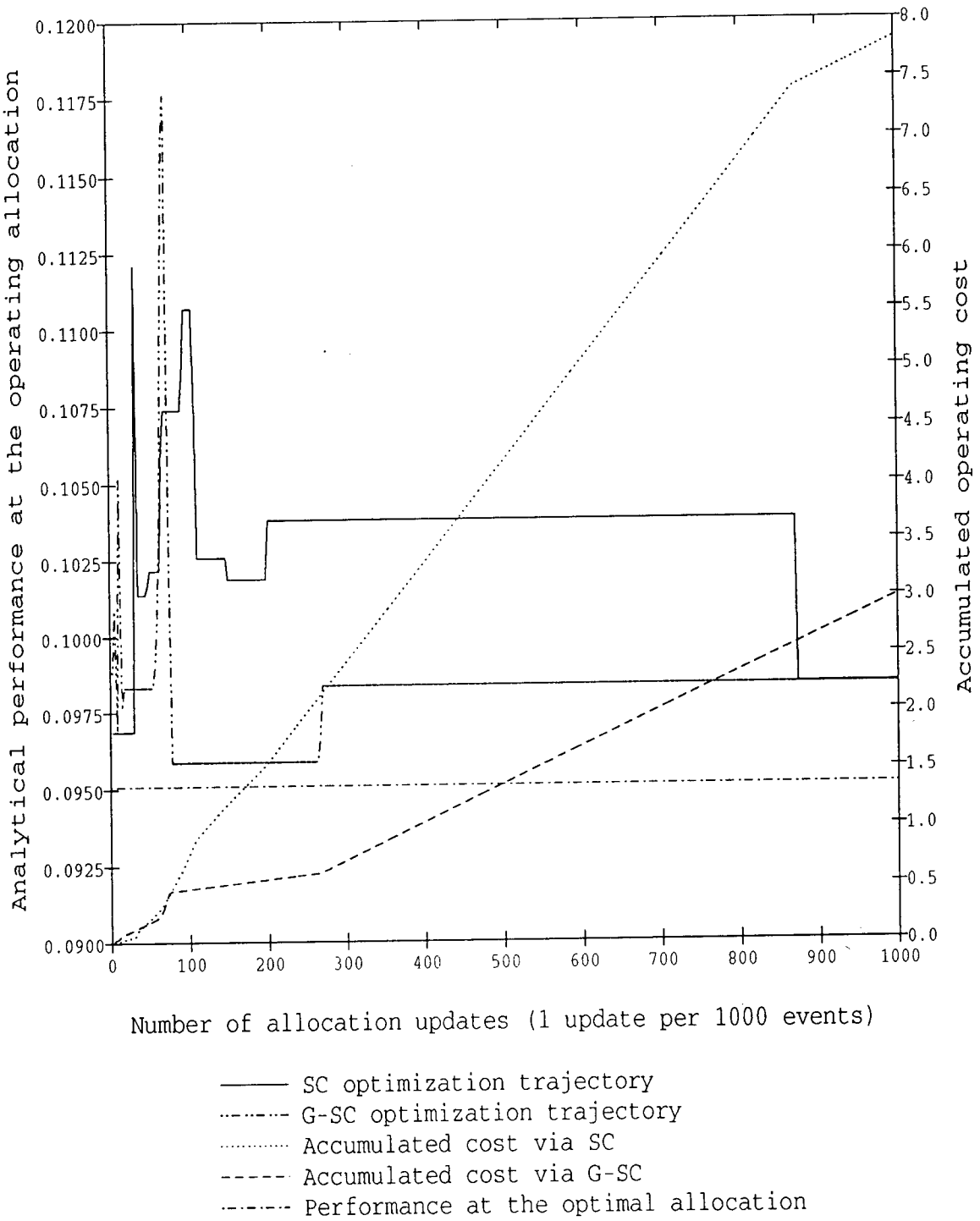


Figure 36: STC and G-STC Algorithms (Simulation Experiment 3)

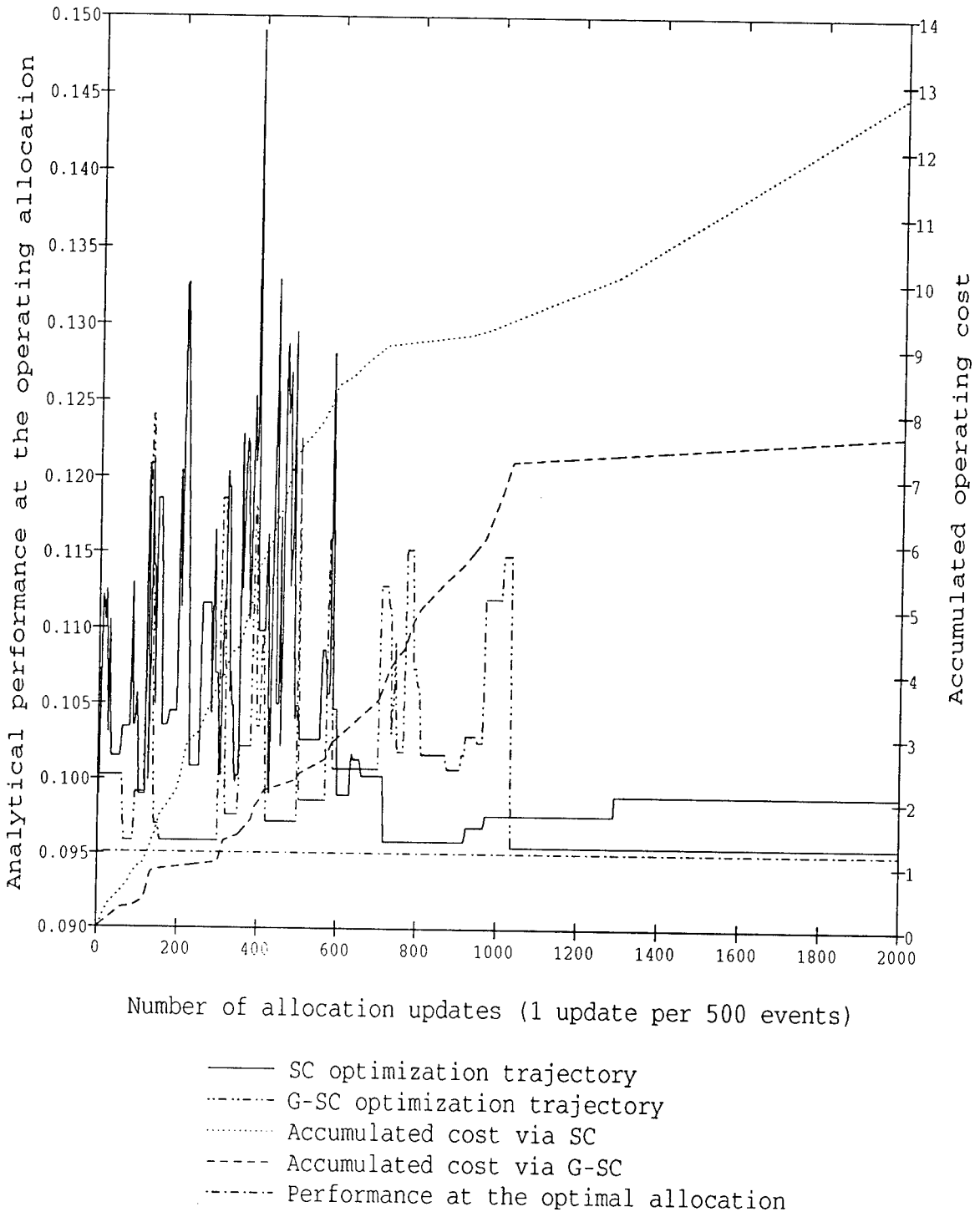


Figure 37: STC and G-STC Algorithms (Simulation Experiment 4)

4.3. Stochastic Descent Algorithm.

The second optimization scheme is based on an algorithm developed in [25] for deterministic models. Here, an allocation vector is denoted by $\mathbf{b} = [n_1,\ldots,n_N]$, where n_i is the number of buffer slots allocated to branch i. The allocation at the kth iteration is denoted by $\mathbf{b}^k$. We also define

$$\Delta L_i(n_i) = L_i(n_i) - L_i(n_i - 1), \quad i = 1,\ldots,K$$

where we have assumed that the performance measure is additive, i.e., $L(n_1,\ldots,n_N) = \Sigma_i L_i(n_i)$.

1. Initialize: $\mathbf{b}^0 = [n_1^0,\ldots,n_N^0]$, $C^0 = \{1,\ldots,N\}$, $k = 0$ and evaluate $[\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)]$.
   If $|C^k| = 1$ go to step 4, else go to step 2.1.

2.1. Set $i^* = \arg\max_{i \in C^k}\{\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)\}$

2.2. Set $j^* = \arg\min_{i \in C^k}\{\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)\}$. Evaluate $\Delta L_{i^*}(n_{i^*}^k - 1)$, $\Delta L_{j^*}(n_{j^*}^k + 1)$

2.3. If $\Delta L_{j^*}(n_{j^*}^k + 1) < \Delta L_{i^*}(n_{i^*}^k)$, go to step 3.1, else go to step 3.2.

3.1. Update allocation: $n_{i^*}^{k+1} = n_{i^*}^k - 1$, $n_{j^*}^{k+1} = n_{j^*}^k + 1$, $n_m^{k+1} = n_m^k$ for all $m \in C^k$, $m \neq i^* j^*$
Replace k by k+1 and go to step 2.1.

3.2. Replace C^k by C^k-$\{j^*\}$. If $|C^k| = 1$ go to step 4, else go to step 2.2.

4. $\mathbf{b}^* = [n_1^k,\ldots,n_N^k]$. Stop.

Under technical conditions presented in [25], this algorithm converges to the actual optimal allocation. However, in the stochastic setting we are considering, the algorithm needs to be modified to take into account that the comparison in step 2.3 involves noisy estimates obtained through simulation. The new algorithm is what we refer to as the *Stochastic Descent* algorithm (see also [23]). The major difference is a randomization mechanism for determining the set C^k at every iteration. In particular, we define a vector $\mathbf{P}^k = [P_1^k,\ldots,P_N^k]$ and determine the set C^k by including branch i with probability P_i^k. Moreover, we define a sequence $\{q_i^k\}$, $k = 0,1,\ldots$ for all i, which represents a "penalty" associated with branch i at the kth iteration.

Stochastic Descent Algorithm

1. Initialize: $\mathbf{b}^0 = [n_1^0,\ldots,n_N^0]$, $C^0 = \{1,\ldots,N\}$, $k = 0$
and $\mathbf{P}^0 = [P_1^0,\ldots,P_N^0]$, $P_i^0 = 1$ for all $i \in C^0$. Evaluate $[\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)]$.

2.1. Set $i^* = \arg\max_{i\in C^k}\{\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)\}$

2.2. Set $j^* = \arg\min_{i\in C^k}\{\Delta L_1(n_1^k),\ldots,\Delta L_N(n_N^k)\}$. Evaluate $\Delta L_{i^*}(n_{i^*}^k - 1)$, $\Delta L_{j^*}(n_{j^*}^k + 1)$

2.3. If $\Delta L_{j^*}(n_{j^*}^k + 1) < \Delta L_{i^*}(n_{i^*}^k)$, go to step 3.1, else go to step 3.2.

3.1. Update allocation: $n_{i^*}^{k+1} = n_{i^*}^k - 1$, $n_{j^*}^{k+1} = n_{j^*}^k + 1$, $n_m^{k+1} = n_m^k$ for all $m \in C^k$, $m \neq i^*, j^*$
Set $\mathbf{P}^{k+1} = \mathbf{P}^k$ and evaluate C^{k+1}. Replace k by k+1 and go to step 2.1.

3.2. Set $P_{j^*}^{k+1} = P_{j^*}^k - q_{j^*}^k$ and evaluate C^{k+1}.

4. If $P_i^k > 0$ for one i only, stop. Else, replace k by k+1 and go to step 2.1.

A rigorous convergence analysis of this algorithm is the subject of ongoing research. In this project, it has been applied to the same buffer allocation problem as in previous sections, and does converge to the actual (analytically obtained) optimal allocation $\mathbf{b}^*$. In **Figure 38**, we can see how the algorithm rapidly converges to the optimal performance as a function of allocation updates with 5000 events observed between updates. This is more clearly shown in **Figure 39**, where in the vertical axis we show allocations ordered from the best (at the bottom) to the 53,130th (worst), with only the top 13,000 allocations shown. Starting with a random initial allocation, we have plotted the updates performed by the Stochastic Descent algorithm. Observe that convergence to the optimal allocation occurs after a remarkably fast 3 iterations.

Since the speed of convergence depends on the initial allocation chosen (which was done randomly in **Figure 39**), we have repeated the simulation experiment above with an initial allocation explicitly selected to be the 53,130th, i.e., the worst possible one. As shown in **Figures 40-41**, convergence now occurs after 17 iterations (equivalently, 85,000 events). This is to be compared to a brute force approach which would require a minimum of 53,130 iterations or the STC and G-STC algorithms of the previous section which are also significantly slower. The reason for the performance of the Stochastic Descent algorithm lies in the fact that it is designed to fully exploit the structure of the resource allocation problem on hand, whereas the STC and G-STC algorithms are much more general-purpose in scope.

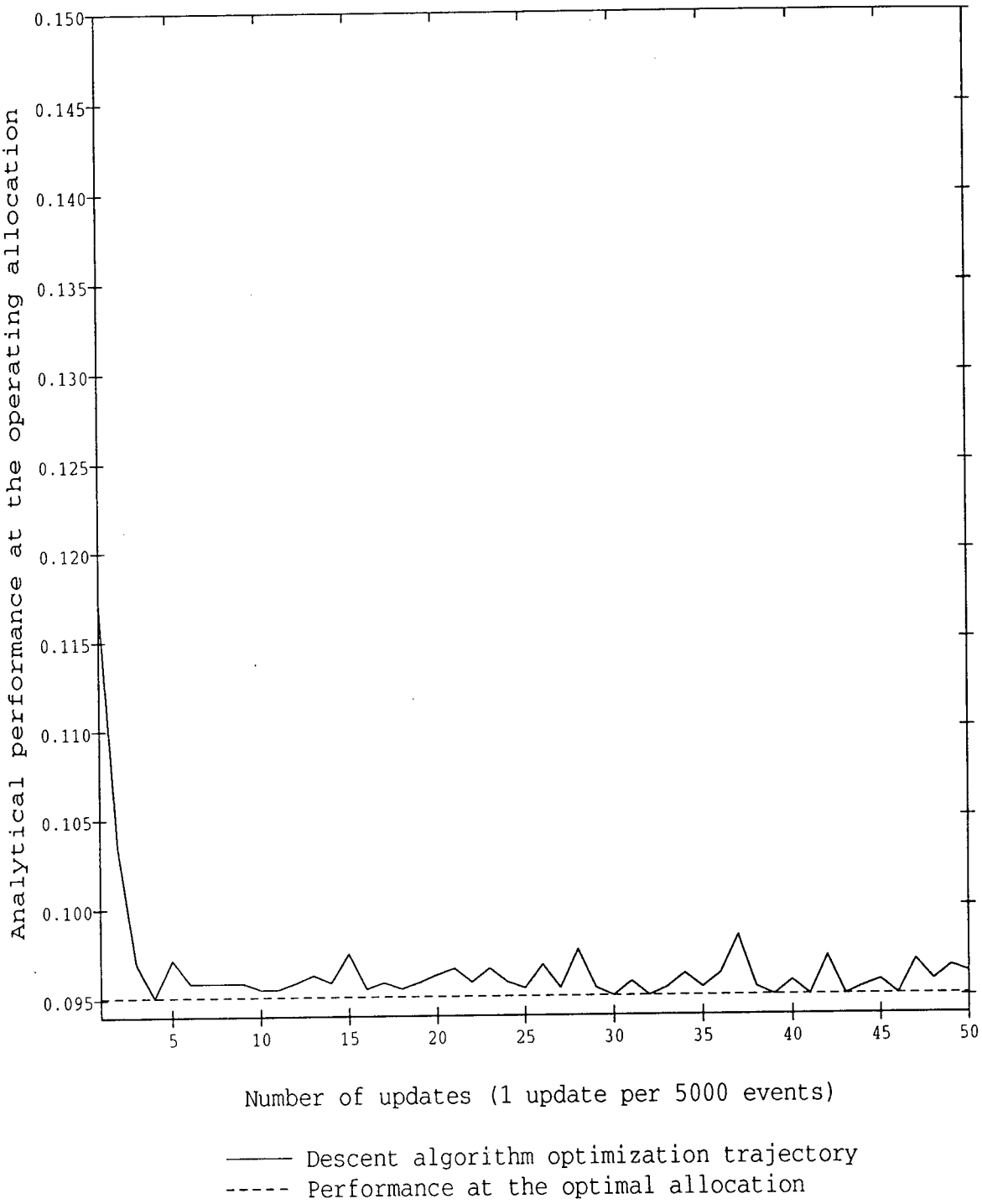


Figure 38: Stochastic Descent Algorithm: Convergence to Optimal Performance

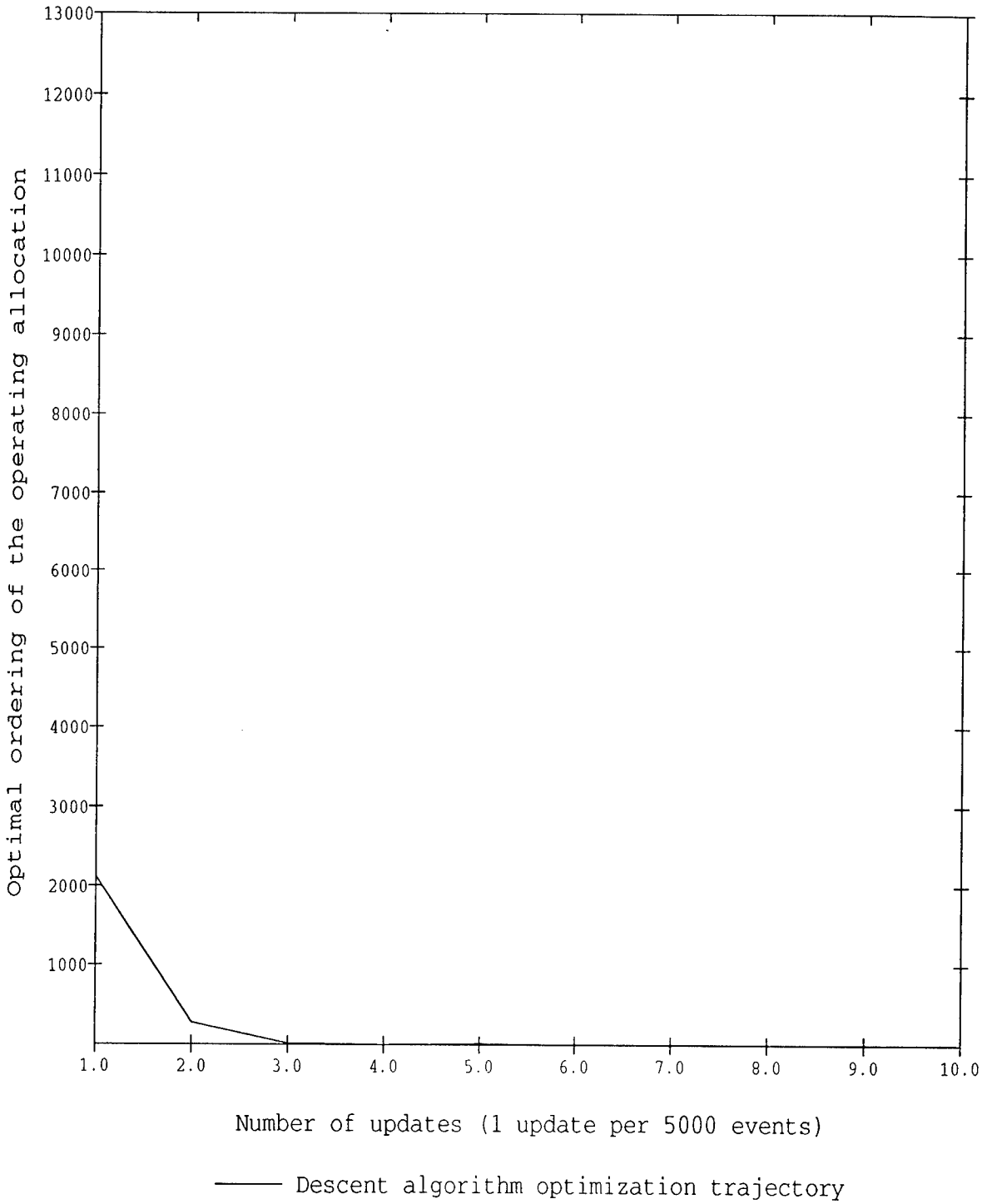


Figure 39: Stochastic Descent Algorithm: Convergence to Optimal Allocation

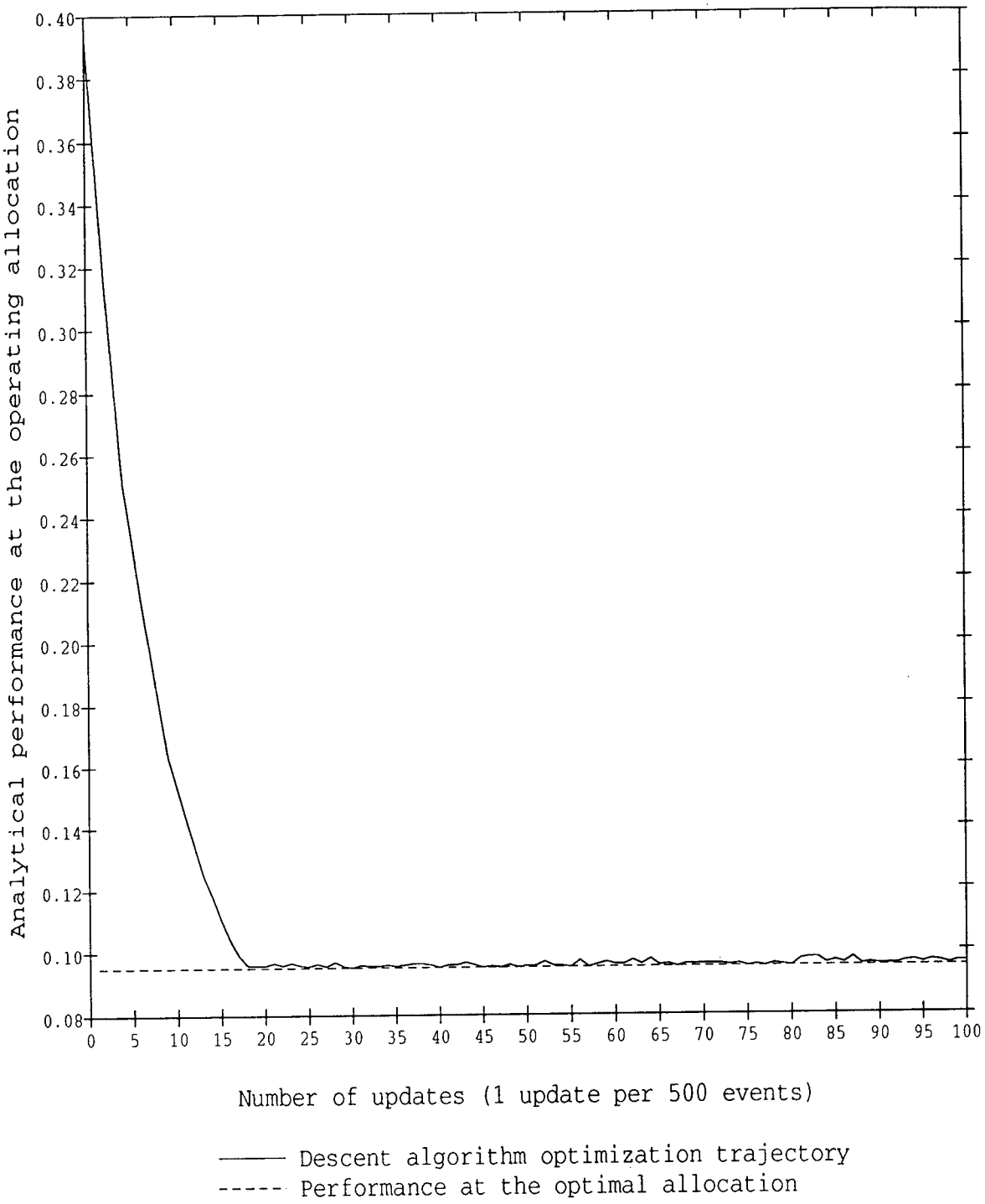


Figure 40: Stochastic Descent Algorithm: Convergence to Optimal Performance (Worst Case)

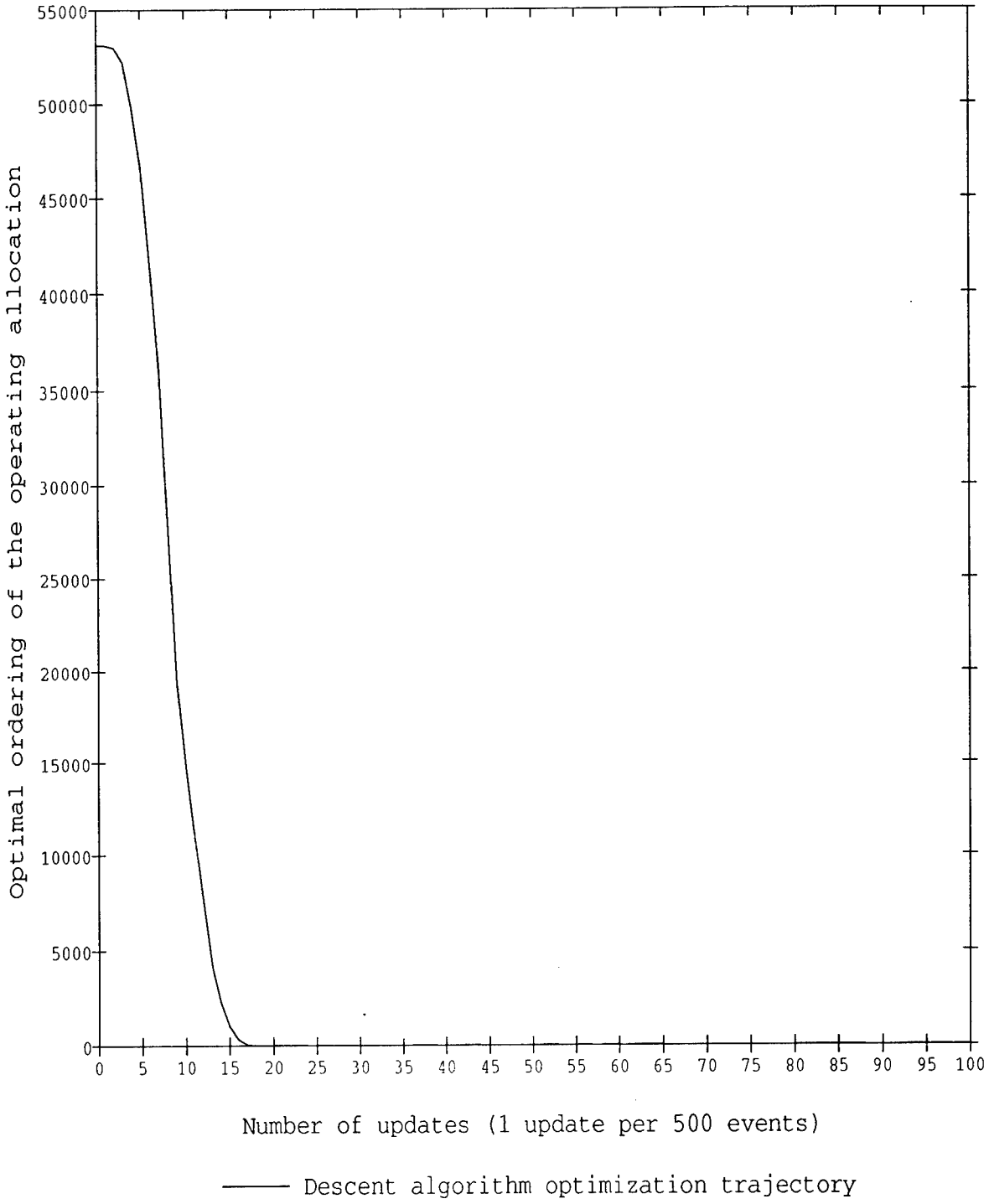


Figure 41: Stochastic Descent Algorithm: Convergence to Optimal Allocation (Worst Case)

5. STOCHASTIC FIDELITY ISSUES IN HIERARCHICAL COMBAT SIMULATION MODELS

The basic elements of a combat simulator include a terrain simulation, an attrition calculation, a high-resolution engagement simulation, and a Forward Edge of Battle Area (FEBA) calculation. Among these elements, the engagement simulation and the related attrition calculation form the fundamental dynamical part that is the driving force of the whole system.

Combat simulation models usually have a hierarchical structure. It would be ideal if the high-resolution battalion level engagements can be simulated directly at the bottom level of the hierarchy. However, this would require long execution times, since such a high-resolution simulator would be called thousands of times for each replica of a battle simulation. Thus, these high-resolution simulations are typically carried out separately. The results are then aggregated into some form of a weapon-attrition score board, which summarizes the statistics of the high-resolution simulation and estimates parameters such as target availability, shooting rate, probability of kill given a hit, target priorities, etc. These parameters are then used to determine the coefficients for a set of nonlinear equations that relate the average number of weapons on each side and attrition for each type of weapon. This approach is usually referred as "calibration". The calibrated equations are solved numerically to get the average attrition. Due to the consideration of execution time, usually the dynamic process of the battle is aggregated in the time axis as well to form time-average quantities. In this process, one usually assumes an exponential decrease of the weapon numbers. This approach has been the working horse for some military analysis units and is an excellent piece of work for the case where computing power is rather limited (see, for example, Concept Evaluation Model, U.S. Army Concept Analysis Agency (CAA), 1985). Nowadays, with the available computing power increased significantly, one can expect significant improvement. In the following 5 subsections, we discuss the crucial issue of preserving the stochastic fidelity in a hierarchical battle simulation model with CAA' s Concept Evaluation Model as the concrete example:

5.1. Overview of A Hierarchical Battle Simulation Model;

5.2. Mathematical Review of the Simulation Model;

5.3. Stochastic Differential Equation Modeling for High Level Attrition Process;

5.4. Scenario Grouping Approach;

5.5. DDA Neural Net for Distribution Learning and Cluster Analysis;

5.6. Summary.

5.1.Overview of A Hierarchical Battle Simulation Model.

Our work on stochastic fidelity preservation is based on a concrete hierarchical battle simulation

model, the COSAGE-ATCAL-CEM model. COSAGE, ATCAL and CEM are the abbreviations of COmbat SAmple GEnerator, ATtrition CALculation and Concept Evaluation Model, respectively. A version of this model is used in the United States Army Concept Analysis Agency. Conceptually this model can be described in the following **Figure 42.**

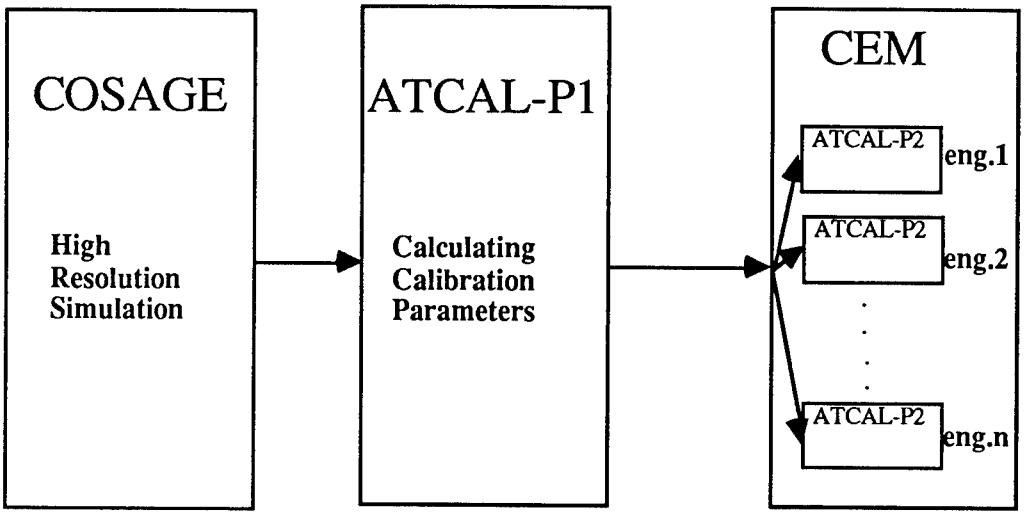


Figure 42. A hierarchical battle simulation model

In the following we explain each component in this system in more detail.

COSAGE: COSAGE is a high-resolution stochastic simulation model of combat between two forces. Typically, the Blue force is sized as a division, and the Red force is scaled from a fraction of a division to a combined arms army. The model simulates periods (normally 48 hours) of combat and produces expenditures of ammunition by round type and losses of personnel and equipment. Maneuver unit resolution is typically down to Blue platoon versus Red company. In the case of close combat, resolution is to the individual equipment and weapon level.

- **Input and Output of COSAGE**
  - **Input:**

Forces, and their organizational structures;

Equipment, weapons, personnel and the characteristics (e.g., strengths, availability, types, densities);

Munitions types and technical characteristics;

Probability values used in simulation;

Priotities,sequences,rules of engagement (tactics and doctrine)

Sensor definition;

Operation concepts.

• **List of COSAGE input data files:**

Air defense sensor; Aircraft munitions; Battery; Category type unit; Counterfire radar; Decision; Equipment; Fire direction center; Forward area rearming and refueling point; Forward observer; High explosives lethal area; Illumination; Mine; Munitions; Orders; Passive detection base; Phased off-line attrition; Posture, environment and mission; Probability of kill; Rules of engagement; Sensor; Smart munitions; Smoke; Sub munitions; System; Tactical aircraft; Target report; Battery type; Battlefield type; Sensor type; Unit; Unmanned air vehicles; Visibility; Weapon.

• **Output**

Unit movement;

Ammunition expenditures;

Equipment and personnel losses (Killer/Victim scoreboard);

Shot summary

Equipment initial density

• **List of COSAGE output files:**

Replication shot summary; Posture shot summary; Killer/victim summary; Direct fire summary; Indirect fire summary; TACAIR/AIRDEF; Target report; Mines; Helicopter/SMARTMUNS report.

COSAGE is a discrete event simulation model with stochastic phenomena modeled through events and processes. A single COSAGE run uses 53 million U[0,1] random numbers.

Representative combat samples can be produced from COSAGE using skillful and ingenious construction of input files. COSAGE can be used to simulate:

(1) Most types of offensive and defensive operations.

(2) The operations of US, allied, joint, combined (excluding naval), and threat forces from the individual to a combined arms army.

(3) The effects of terrain, weather, and obscurants on combat operations.

(4) The movement of combat and combat support units on the battlefield.

(5) The performance and effects of most current and future weapons and equipment.

(6) The major aspects of target acquisition such as ground surveillance radar, forward observers, flash and sound sensors, counter mortar and counter battery radar, and remotely piloted vehicles.

(7) The direct fire battle effects due to small arms, tanks, fighting vehicles, antitank guided missiles, and mines.

(8) The indirect fire effects from tube artillery, mortars, and multiple launch rockets --- to include the effects from conventional and improved conventional munitions, precision

guided munitions, laser guided munitions and mines.

The objectives of COSAGE are:

(1) To calculate expected ammunition expenditures and equipment/personnel losses for both sides.

(2) To provide a record of killer/victim relationships to the ATCAL for calibration.

(3) To provide a record of losses and expenditures to the simulation post processors.

Generally, six different combat operations are modeled for a study using one or more of the following types of terrain.

- **Terrain Types:**

A. Flat to gently rolling with minimum obstacles.

B. Gently rolling terrain with some obstacles and vegetation.

C. Mountainous with steep slopes and/or dense forestation or swamps.

- **Combat Operation Types:**

(1) Blue Intense Defense (I): The blue division in a prepared defense against an attacking Red force.

(2) Blue Delay (D): The Blue division conducting a delay or a defense on alternate or successive defense positions against an attacking Red force.

(3) Blue Hasty Defense (H): The Blue division in a hasty defense against an attacking Red force.

(4) Static (L): The Blue division and a Red force are at parity and are both in defensive positions. Both sides are conducting patrols, probes, and reconnaissance.

(5) Blue Attack (F): Multiple Blue divisions conducting an attack against a Red division in a prepared defense.

(6) Red Hasty Defense (N): Multiple Blue divisions conducting an attack against a Red division in a hasty defense.

The results of the division combat simulations are called combat samples. Combat samples represent the expected results, during a theater campaign, for division combat for 2 days of the posture simulated. The combat sample is not intended to represent the first high-intensity period nor the last period when intensity is expected to be low.

Combat samples are analyzed to ensure that the military aspects of combat are adequately and faithfully portrayed in the model. The ammunition expenditures and equipment losses for both the Red and Blue forces are determined, and killer/victim relationships are established.

ATCAL: A key function of any combat simulation is the calculation of losses of equipment and personnel in the engaged forces. In simulations of small-scale engagements, this can be

accomplished through a detailed treatment of all shooters and their potential targets, with each firing opportunity examined as to its feasibility and outcome. The high-resolution simulation (resolved down to the interactions between individual weapons) can be done readily at battalion level but only with great computer time and resource penalties at division level. Although this approach gives the most realism, it cannot be considered for the large-scale combat occurring at theater level . Here one must use attrition equations to compute the interactions between Blue and Red weapons. These attrition equations are of various forms, all attempting to relate numbers of shooters and numbers of targets to losses in engagements. Attrition equations always contain parameters which are difficult to determine, such as allocations of fire to various target types or maximum rate at which a given type of shooter can kill a given type of target. They also make use of aggregation, the grouping together of weapons and targets in some logical way in order that the losses inflicted by each of the groups upon each group in the opposing force can be computed. The Concepts Evaluation Model (CEM) aggregates the weapons into Blue and Red heavy armor (tanks), light armor (APCs), soft (dismounted personnel), artillery, helicopter, and fixed-wing categories. Other theater models aggregate into vehicle types: several tank types, several APC types, several artillery types, etc. When the parameters of an attrition equation are chosen in such a way that aggregated combat results agree with the results of a high-resolution simulation, for a particular mix of weapons on the two sides, the equation is said to be calibrated. A large part of chronic dissatisfaction with large-scale simulations is due to the failure of attrition equation results (with a fixed set of parameters) to track well with high-resolution results (or with judgemental expectations) as the composition of the forces is varied.

ATCAL is a method for calibrating a set of attrition equations to the results of sample high-resolution simulations. It uses auxiliary equations to feed the main attrition equations, modifying their parameters and thereby accounting for considerably more battlefield detail than heretofore. This added flexibility permits better portrayal of the results of force variations. The method uses high resolution results and provides useful side information in addition to the loss-by-cause table (commonly refereed to as a killer-victim scoreboard). This side information is the allocation of fire among all shooter and target types, the expenditures of ammunition, the relative importance of all weapons, and the force ratio in the engagement.

The starting point of the new methodology is a pair of new attrition equations, one for point fire and one for area fire. Both equations compute losses by cause and are therefore fundamentally different from attrition equations which operate with aggregated firepower. The point-fire equation has two main calibration parameters: kills per round fired at the target and availability of a target to a shooter. These two parameters, each depending upon shooter and target type, can be uniquely

determined from two main output matrices of the high-resolution simulation: the firing matrix (shots fired by each shooter type against each target type) and the attrition matrix (or killer-victim scoreboard). Calibration for the point fire part consists of using the ATCAL methodology "backward" to determine the two parameters. To determine losses for a force of some different composition from the calibration force the ATCAL methodology is run "forward". The area fire equation utilizes two main calibration parameters also: the kills per round (given that the round is associated with the target) and a response factor for each tube type, which states the volume of fire delivered to satisfy the demand for fire (the demand being an internal ATCAL calculation). There are no time steps in ATCAL --- the entire engagement is treated as a single time step and average numbers of participants are used, instead of starting numbers, to produce a decrease in fire as losses go up.

Discussion of ATCAL: The ATCAL model is an implementation of a method which employs hundreds of interconnected equations. We are to describe the main equations of ATCAL and the reasoning behind; and to show how the equations are coupled together. The coupling is different in each of the two distinct parts of ATCAL. The two parts, called Phase I and Phase II, are what was referred to above as running the model backward and forward --- for calibration of parameters and for prediction of results when new forces are employed. In our report, we concentrate on ATCAL phase II.

• **Importance of Weapons:** Fire allocation is part of the ATCAL process. Since allocation goes preferentially to the more important targets, a means of quantifying the importance of targets is a necessary first step in fire allocation. The importance of weapons comes from a nonlinear operation on the killer-victim scoreboard and uses only the scoreboard itself. Since the scoreboard is a key element in ATCAL, the importance can be readily computed within ATCAL for the engagement at hand.

• **Vehicles and their average number:** Each vehicle, used to denote a killable entity on the battlefield, can be both shooter and target and, in general, each will have several weapon types on it. The average numbers of vehicles of each type in the engagement are used in the attrition equations to produce a dynamic model which responds appropriately to changes in engagement length. For an engagement with heavy attrition of some systems, it is very wrong to use the number present at the beginning in computing fire allocations. Fire allocation to a target type should decrease as the target is depleted. Similarly, the fire from that depleted vehicle type should decrease. Using an average number, which can assume arbitrarily small values, in the attrition equations is a way of taking a vehicle out of the picture in response to its depletion. The straight line average cannot assume arbitrarily small values, therefore the average is computed from an

assumed exponential decrease in weapon count during the course of the engagement.

Suppose the starting number of weapons is N (see **Figure 43**). After time t, the remaining number becomes $N(t) = Ne^{-\lambda t}$. Therefore, if we denote the attrition by ΔN, the average number of weapons during the engagement period T, $\bar{N} = \frac{1}{T} N \int_0^T e^{-\lambda t} dt$ can be expressed as

$$\bar{N} = -\Delta N / \ln(1 - \frac{\Delta N}{N}) \tag{21}$$

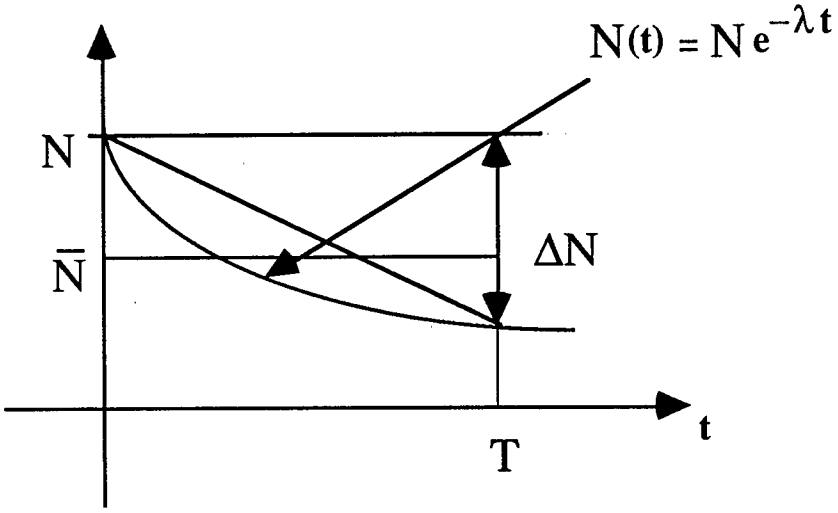


Figure 43: Attrition Process over Time

Note that in the above equation ΔN and $\bar{N}$ depend upon each other in an inverse sense (an increase in one results in a decrease of the other), it is clear that there must be a solution in which the two are in equilibrium. Therefore, in ATCAL phase II, we take $\bar{N} = N$ initially and iterate to solve for ΔN.

We note that there are actually some justifications for the exponential decay assumption for the point fire. Denote by R and B the average strength of the red army and blue army, respectively. Then according to the Lanchester equation,

$$\begin{cases} \dot{R} = -c_r * B \\ \dot{B} = -c_b * R \end{cases}$$

from which we can get

$$\begin{cases} R(t) = R_0 * e^{-\sqrt{c_r * c_b}\,t} \\ B(t) = B_0 * e^{-\sqrt{c_r * c_b}\,t} \end{cases}$$

with $\frac{B_0}{R_0} = \sqrt{\frac{c_b}{c_r}}$.

• **Target priorities:** Each weapon or round type must have its targets prioritized so that the model can compute allocations of fire to targets for point fire or allocations of rounds by type for area fire. Target priority is computed as the product of kills per round and target importance. Thus a target only becomes lucrative to a weapon if it is both important in itself and more easily killed than other targets available to the weapon.

• **Attrition equation for direct fire:** Consider a property of a shooter-target pair to be target availability, defined as the fraction of time a particular target (of a type) can be fired upon by a particular shooter (of a type). The quantity can be thought of as averaged over all shooters and targets (of the types in question). Denote this quantity by A_{ijk} --- the fraction of time that k-th target is available for j-th weapon of i-th vehicle. If there are $\bar{N}_k$ targets of type k, the fraction no target is available is $(1-A_{ijk})^{\bar{N}_k}$, so $1-(1-A_{ijk})^{\bar{N}_k}$ is the fraction of time at least one target is available. Now each weapon j of all the $\bar{N}_i$ shooters is capable of firing $(RATE)_{ij}$ rounds during the engagement, and it will fire at vehicle k if no other target with higher priority is available. If the vehicles with higher priorities are index by k', and the probability of kill per round is P_{ijk}, then we can write the attrition equation for point fire as

$$(\Delta N_k)_{ij} = \bar{N}_i (RATE)_{ij} P_{ijk} [1-(1-A_{ijk})^{\bar{N}_k}] \prod_{k'} [1-A_{ijk'}]^{\bar{N}_{k'}} \qquad (22)$$

Rewrite equation (21) as

$$\Delta N_k = (1-e^{-\Delta N_k/\bar{N}_k}) N_k \qquad (23)$$

Given the starting number of vehicles, the calibrated parameters $(RATE)_{ij}$, P_{ijk}, A_{ijk}, we can then solve the nonlinear equations (22) and (23) by iteration to obtain ΔN_k. The importance index k' changes along with each iteration. This process is summarized as the following steps:

• **Iterative Algorithm for ΔNk and N̄k (ATCAL II)**

Step 0: Set $\overline{N_k^0} = N_k, (t=0)$;

Step1: Calculate $(\Delta N_k^{t+1})' = f(\overline{N_i^t}, \overline{N_k^t}, \overline{N_{k'}^t})$ using (22);

Step2: $\Delta N_k^{t+1} = [1-e^{-(\Delta N_k^{t+1})'/\overline{N_k^t}}] N_k$;

Step3: Calculate $\overline{N_k^{t+1}} = \overline{N_k^t} * \Delta N_k^{t+1} / \Delta N_k^t$;

Step4: Check if $|\overline{N_k^{t+1}} - \overline{N_k^t}|/N_k < \varepsilon$ or $t \leq 15$. If not, set $t = t+1$ and go back to step 1.

Remarks:

1. Note that the purpose of the iterations above is to solve the nonlinear equations, not to simulate the evolution of the system over time. In another word, there is no time steps in ATCAL; the entire engagement is treated as a single time step and average numbers of participants are used, instead of starting numbers, to produce a decrease in fire as losses go up.

2. The weapon importance changes during the course of the iteration. This is due to the fact that the importance of a weapon is unknown to us and it depends on the number of kills and the importance of kills inflicted by this weapon. Therefore, as the attrition happens, the importance of the weapons is also adjusted. Eventually, the weapon importance converges because the attrition values converge to their solution.

• **Area Fire Attrition Equations:** As the model sequentially processes the weapon type on each shooting vehicle, it encounters an indicator which tells it whether the weapon is to be processed with point-fire or area-fire. The attrition equations are different for these two types of fires and so are all the parameters that go into them.

The problem for area-fire is to account for (a) the amount of firing that is to be done; (b) the apportionment of the firing among the different round types; (c) the effects of the firing on the target arrays.

The computation of the attrition due to area-fire involves many intermediate steps, which we do not describe here. We merely list the attrition equation in the following and note that the principle of solving the nonlinear equations for area-fire is no different from that for point fire. The area-fire attrition equation is

$$(\Delta N_k)_{ij} = (RSPNS)_i\,(DEMAND)_i\,(Allocation)_{ij}\,(FRAC)_{ijk}\,(\bar{N}_k/N_k)L_{ijk} \qquad (24)$$

where $(RSPNS)_i$, L_{ijk} are calibration parameters obtained from ATCAL phase I, while $(DEMAND)_i$, $(Allocation)_{ij}$, $(FRAC)_{ijk}$ are obtained from the above and another calibration parameter $(BIAS)_{ij}$ in the intermediate steps. With the assumption that the number of vehicles decreases exponentially, we can still use equation (23) and thus solve the nonlinear equations (23) and (24) recursively to obtain the attrition due to area-fire.

The flow chart of the ATCAL Phase 2 main cycle is shown in **Figure 44**.

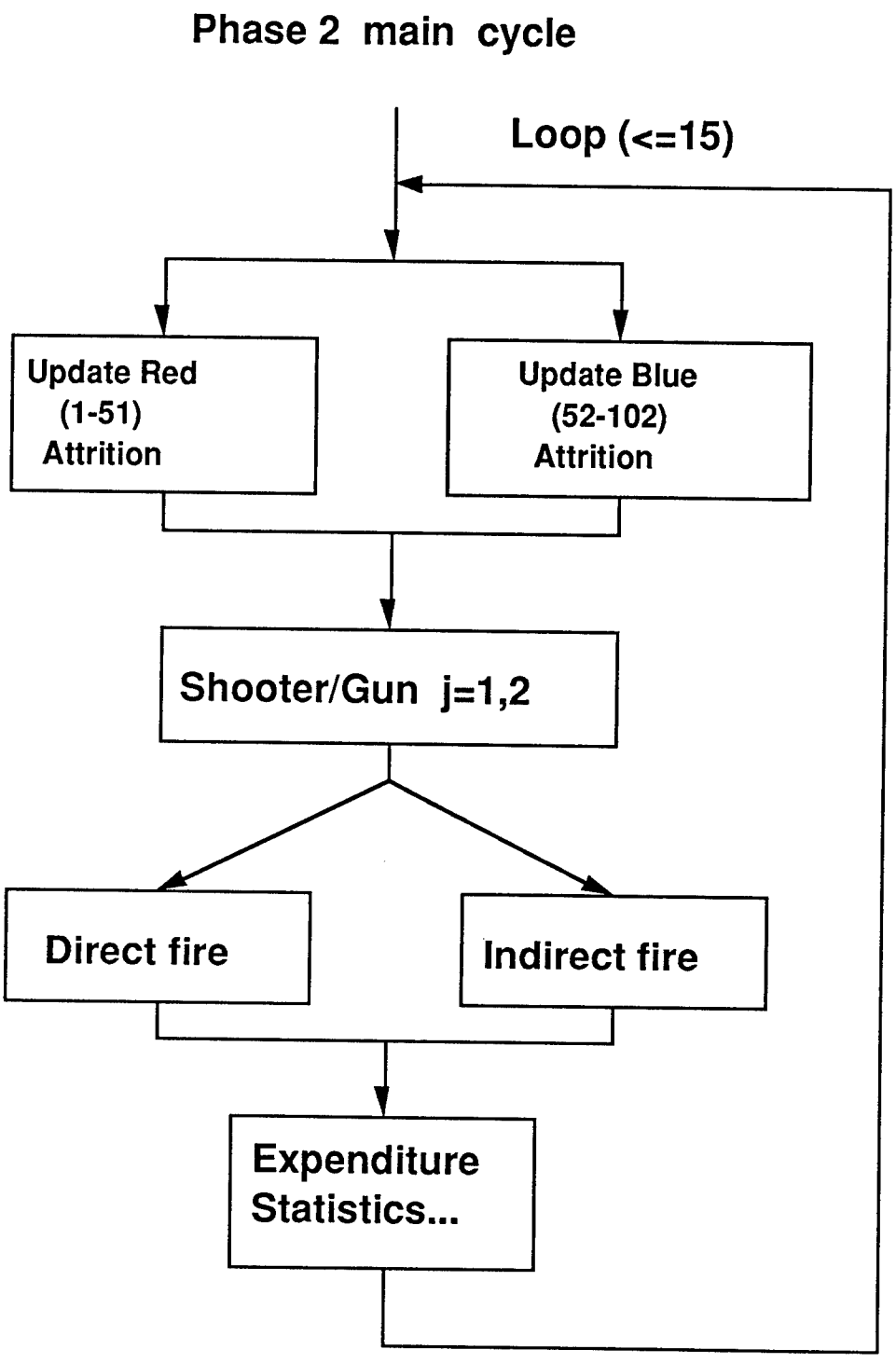


Figure 44. Flow Chart for ATCAL PHASE II

5.2. Mathematical Review of CEM and STOCEM.

• **A comment on the nonmonotonic behavior of CEM:** The nonmonotonic behavior or "structural variance" has occurred in the CEM, stemming from the use of decision thresholds; a slight increase in strength of one side may alter the force ratio that is compared to a decision threshold, resulting in a different allocation of forces and a significantly different outcome. This phenomenon deserves much more research but it is beyond the scope of this project. Next we try to review the underlying philosophy of the COSAGE-ATCAL-CEM from its mathematical formulation. We start with the ultimate goal of a hierarchical battle simulation model.

• **What do we really want from CEM?** What we wish to obtain through a battle simulation model is the high resolution simulation estimates for the mean (and higher moments of, if possible) attrition at the theater level. We denote the mean attrition for the "throughout high resolution simulation (THRS)" as

$$E[g(w)]$$

where $g(w)$ is the sample THRS output with seed w.

• **CEM approach:** The COSAGE-ATCAL-CEM approach to this problem can be divided into two steps:

a) Run COSAGE to get samples of "calibration" parameters such as

probability of kills per round $P(\omega)$;
availability (ω);
firing rate $R(\omega)$;
response factor $F(\omega)$;
bias $B(\omega)$;.

Then estimate their mean values P, A, R, F, B (ATCAL I)

b) Use a Lanchester-like attrition equation to calculate the mean attrition, i.e., approximate $E[g(\omega)]$ by

$$\Delta N=f(P, A, R, F, B) \quad \text{(ATCAL II)}$$

We now discuss the limitations of the CEM approach.

• **General Limitation of CEM approach:** Current CEM model has fixed engagement duration and can not be interrupted. More seriously, although CEM accepts different initial mixtures as input, the execution of ATCAL Phase II inside CEM uses the output from COSAGE with a single initial posture. This is due to the fact that setting up an initial mixture for COSAGE takes a long time and a lot of effort. Needless to say this situation is not satisfactory and more research is needed to handle this. Finally, as we explained before, in ATCAL Phase II one

averaged the attrition over the whole duration of the engagement and assumed exponential decrease of the attrition. Since each engagement always consists of an initial search period with not much firing, an intensive firing period and an ending period when one side or both sides decide to quit the engagement, to average over time is hardly descriptive of the attrition process. In summary the current CEM approach has the following limitations:

(1) fixed engagement duration,

(2) no interruptibility,

(3) fixed initial mixture for a given posture in COSAGE..

(4) averaged over time

Next we discuss the central issue of our report, the preservation of the stochastic fidelity in CEM model.

• **Preservation of Stochastic Fidelity Issues in CEM Approach:** The following are the two major points in these issues.

- COSAGE runs generate drastically different scenarios, and using the overall mean of the calibration parameters over all the scenarios as the input parameter to ATCAL Phase 2 may not give good attrition estimates.
- COSAGE sample shows that the attrition dynamics are clearly different during different engagement periods. So use of the average over time may not be appropriate.

These points have been observed and there have been attempts to overcome some of these problems. One such approach being considered is the STOchastic CEM (STOCEM) approach. We now discuss this approach.

• **STOCHASTIC CEM (STOCEM) approach:** The stochastic fidelity issue has been noticed in the battle simulation society and a certain amount of effort has been devoted to resolve the issue. Stochastic CEM (STOCEM) is one such approach. The basic idea of STOCEM is to use each COSAGE sample as the input to the ATCAL Phase 2. The output of the CEM is considered as one sample of the stochastic output. The final result is the distribution of, say, the attrition, rather than their mean values as in the case of CEM. 10-12 replications from COSAGE is considered good enough in the current version of STOCEM. For each replication, STOCEM still uses the original calibration procedure. Experimental results show that sometimes the original CEM outputs lie out of the confidence interval of the corresponding STOCEM results.

We think STOCEM is a commendable effort. However since for each replication STOCEM still uses the original calibration procedure, one has to ask the fundamental question whether the mean value-based Lanchester Law is valid for samples. To answer this question we need to review the mathematical description of the STOCEM approach.

• **Review of STOCEM Approach:** STOCEM begins by running COSAGE to get samples for "calibration" parameters such as probability of kills per round $P(\omega)$; availability (ω); firing rate $R(\omega)$; response factor $F(\omega)$; bias $B(\omega)$; others. Then, STOCEM uses a Lanchester-like attrition equation to obtain samples of the attrition, i.e., approximate $g(\omega)$ by $f(P(\omega), A(\omega), R(\omega), F(\omega), B(\omega))$. Finally, when a mean value is desired, STOCEM estimates $E[g(\omega)]$ by averaging over $f(P(\omega), A(\omega), R(\omega), F(\omega), B(\omega))$.

The major question we have for the STOCEM approach is whether the Lanchester attrition equations are valid for samples of COSAGE. One could argue that there is considerable averaging in COSAGE over many possible battle scenarios already, so the Lanchester Law could be applied to the COSAGE output. However to average over scenarios generated from a given initial mixture along time is different from averaging over scenarios with more rational weights. This is because the weights attached to the former approach are determined by the battle evolution along time. Such an average could be more wrong than uniform weighted average, let alone a rational choice of the weights. The latter should be what we would like to achieve.

In summary we have the following observations regarding the STOCEM approach.

- Lanchester-like equations are based on the idea that average attrition is proportional to enemy's average strength, which makes sense only for the mean values.
- Note $E[f(X)] \neq f(E[X])$ where f is ATCAL II which does not make much sense to the samples X!
- Even if we know the distribution of P, A, etc., we still don't know how to incorporate them (or just the second moment) into the attrition equation to reflect their impact on $E[g(\omega)]$
- Over the range of starting points that ATCAL II must address, the calibration parameters remain constant for the posture.
- CEM 12 hours engagement is actually a kind of average over (unspecified) phases of a 48 hour combat process.

Mathematically the fundamental issue with the preservation of the stochastic fidelity lies in the inequality $E[f(x)] \neq f(E[x])$ with nonlinear $f(\cdot)$. We give several examples to illustrate this inequality. Our first example is extremely simple:

Example 1: $f(x) = x^2 \quad \Rightarrow \quad E[x^2] = s^2 + E[x]^2$

Our second example is closer to the equations used in ATCAL for computing the attrition.

Example 2: $f(x) = a^x, \quad 0 < a < 1$

Note that in ATCAL we need to use the power function to compute probabilities involved in the

attrition calculation. In the following we illustrate the inequality $E[f(x)] \neq f(E[x])$ for several different distributions for x.

Example 2a.

$$\Pr(X=1)=p, \quad \Pr(X=0)=1-p$$

$$E[X]=p$$

$$f(E[X])=a^p$$

$$E[f(X)]=pa^1+(1-p)a^0=pa+1-p$$

Example 2b.

$$\Pr(X=k)=q^k p, \quad q=1-p, \quad k=0,1,\text{K}$$

$$E[X]=\frac{q}{p}$$

$$f(E[X])=a^{\frac{q}{p}}$$

$$E[f(X)]=pa^0+qpa^1+q^2pa^2+\text{K}=\frac{p}{1-qa}$$

Example 2c.

$$\Pr[X=k]=\frac{l^k e^{-l}}{k!}, \quad for\ k=0,1,\text{K}$$

$$E[X]=l$$

$$f(E[X])=a^l$$

$$E[f(X)]=\frac{l^0 e^{-l}}{0!}a^0+\frac{l^1 e^{-l}}{1!}a^1+\frac{l^2 e^{-l}}{2!}a^2+\text{K}=e^{-l}e^{la}=e^{-l(1-a)}$$

We now turn to another approach for preserving the stochastic fidelity. This approach has been studied in [26,27].

5.3. Stochastic Differential Equation Modeling for High Level Attrition Process.

Several researchers have suggested to use the Ito type of stochastic differential equations to model the attrition process in battle simulation. The main idea is to let the deterministic part of the stochastic differential equation model the Lanchester attrition computation process and let the Wiener process-driven part represent the stochastic component of the engagement. [26] and [27] are two nice references on this methodology.

One key issue of implementing this idea on a digital computer is the discretization of the continuous Ito stochastic differential equation. Fortunately, in the last 15 years numerical solutions for stochastic differential equations have drawn considerable research attention. It is now feasible to simulate stochastic differential equations on a digital computer efficiently. Note that the discretization of Ito stochastic differential equations is not as trivial as for the deterministic

differential equations, due to the sophistication of the definition of the Ito calculus. [28] is a nice reference for this issue. We now briefly discuss the mathematical formulation of this approach.

The basic Stochastic Lanchester Attrition Equation takes the following form.

$$dx_i = f(x_1,x_2,...,x_n)dt + g_{ij}(x_1,x_2,...,x_n)dw_i(t), \quad i= 1,..., n;$$

where $x_i(t)$, $i = 1,2,..., n$ is the strength of the i-th weapon. The method consists of the followig steps:

- Choose $f(\cdot)$ and $g(\cdot)$ to fit the mean and the covariance of the lower level output;
- Choose a numerical scheme to simulate the system equation.

We point out that to use stochastic differential equations for attrition modeling we need to discretize it and fit the time varying coefficients. This last requirement makes it difficult to implement in practice. On the other hand we believe that to fit the time varying coefficients eventually yield essentially the same thing as the following "scenario grouping" scheme, which is easier to implement. So we now turn to the discussion of the scenario grouping approach.

5.4. Scenario Grouping Approach

• **Basic Idea:** Since different random seeds in COSAGE shall result in different scenarios, the correct thing to do is to group the results of COSAGE into scenarios, S_1, S_2, ..., S_n, and calculate

$$E[g] = E[g|S_1]*Pr(S_1)+E[g|S_2]*Pr(S_2)+...+E[g|Sn]*Pr(Sn)$$

For each scenario S_i, we can get the calibration parameters P_i, A_i, etc. and then use $f(P_i, A_i, ...)$ to approximate $E[g|S_i]$.

Professor S.M. Robinson (University of Wisconsin) made the following remarks in [29]: "Note that this use of an expected value performance measure is not at all the same thing as the common use of "expected value models" in which stochastic elements are individually and systematically replaced by their expected values. That procedure is invalid as a method for modeling anything, since the outcome can not be reliably related to the average of the outcomes under the individual scenarios, or to any other quantity of interest. Rather, the expected value performance measure that we are using corresponds to use of a Monte Carlo simulation process, but (as we shall see below) with a certain degree of increased structure."

In the following we use a queueing model example to show that different random seeds result in different function $g(\cdot)$.

The example performance function we use is the mean waiting time of the 5th customer in an M/M/1 queue. For each random seed, we get the samples of the interarrival time $a_1(w), a_2(w), \mathrm{K}, a_5(w)$ and the samples of the service time $s_1(w), s_2(w), \mathrm{K}, s_5(w)$. The waiting

time of the 5th customer is the sum of the service time of all the previously arrived customers who are in the same busy period as itself. So if we define Si as the scenario that i customers are in front of the 5th one and are in the same busy period, then we have

$$W_5(\vec{a}(w),\vec{s}(w)|S_i) = f_w(\vec{a}(w),\vec{s}(w)|S_i) = \sum_{k=5-i}^{4} s_k$$

Hence different random seeds not only result in different $\vec{a}(\omega),\vec{s}(\omega)$ but also different functions. We show pictorially in **Figure 45** that different random numbers could generate different scenarios.

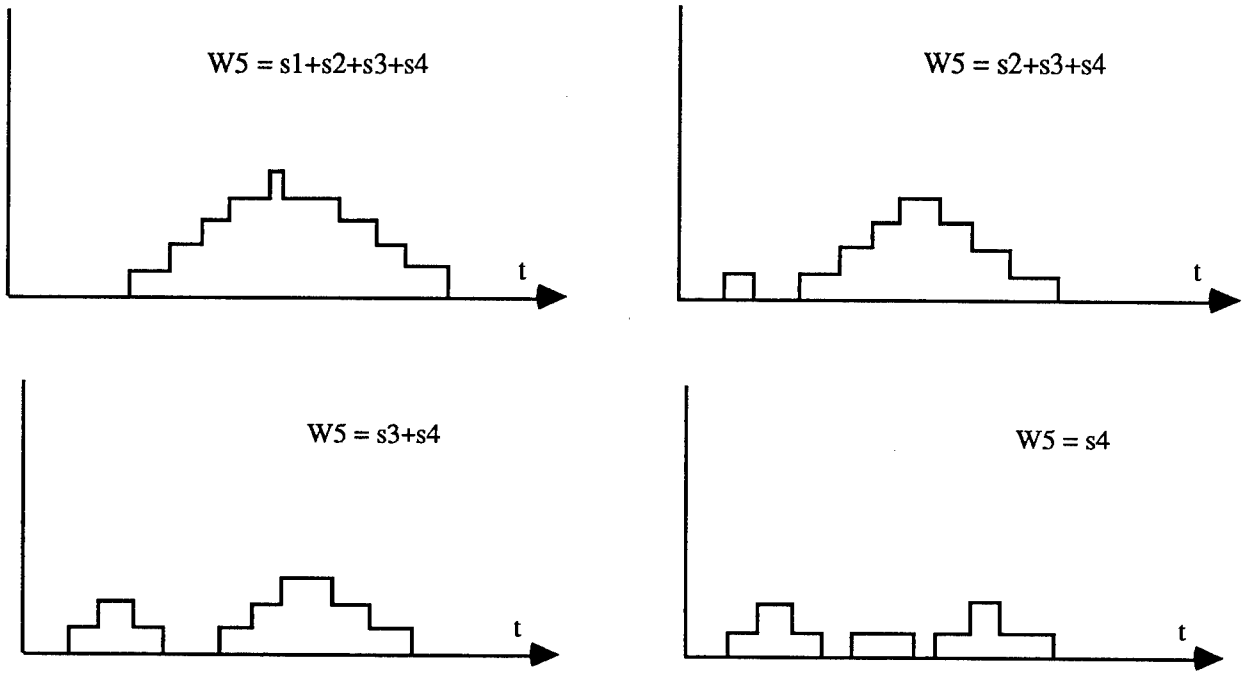


Figure 45. Four Scenarios for A Simple Queueing Sample Path

Now we use a simple queueing example to illustrate the scenario grouping idea. Consider a M/M/1/K queue with utilization r .The probability of buffer full is calculated by the mean formula

$$P_K(\rho) = \frac{(1-\rho)\rho^K}{1-\rho^{K+1}} \qquad (25)$$

This formula is valid only when r is a constant. When used in a simulated model, r has to be estimated accurately. Using (25) with a sample of r is not justified.

We compare this queueing example with the battle simulation model CEM:

•/M/1/K Queue	CEM
$P_K(\rho) = \frac{(1-\rho)\rho^K}{1-\rho^{K+1}}$	$\Delta N = f(P, A, \ldots)$
r	$P, A, \ldots$

The question is : **Can we use (25) when r is a random variable?** Suppose we have a MMPP/M/1/K queue shown in **Figure 46**. We want to estimate r and use it as the calibration paramemter and then use equation (25) to estimate P_K. Suppose also that we have chosen $K = 8$, $m = 10$, and obtained eight estimates of r : 1,2,3,4,5,6,7,8. Now we estimate P_K using four different schemes:

1) Treat all the estimates as one group (a la CEM), i.e. calculate the mean of r, 4.5, then (25) gives $P_K = 0.93 * 10^{-3}$
2) Divide the estimates into two groups: {1,2,3,4} and {5,6,7,8}, compute their respective mean value: 2.5, 6.5 and then use (25) to get
$$P_K = P_K(2.5) * 0.5 + P_K(6.5) * 0.5 = 5.7 * 10^{-3}$$
3) Divide the estimates into four "scenario groups": {1,2}, {3,4}, {5,6}, {7,8} and then use (25) to get
$$P_K = P_K(1.5) * 0.25 + P_K(3.5) * 0.25 + P_K(5.5) * 0.25 + P_K(7.5) * 0.25 = 7.7 * 10^{-3}$$
4) Divide the estimate into eight groups (a la STOCEM), i.e., plug each estimate into (25) and then compute the average. In this case we get $P_K = 8.2 * 10^{-3}$

The question is: Which scheme is "more correct"?

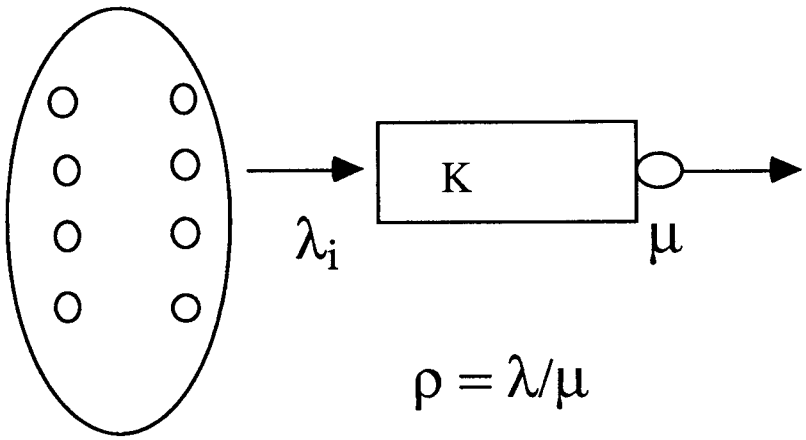


Figure 46. MMPP/M/1/K Queue

Suppose the arrival process is modulated by a two-state continuous time Markov chain. The mean sojourn time of each modulating state is 50, the arrival rate associated with each modulating state is 2.5, 6.5, respectively. By simulation, $P_K = 5.5 * 10^{-3}$. So method 2) above gives a good estimate (Because 3) and 4) did not adequately "smooth" the samples and 1) fails to convey the impact of the two scenarios).

Suppose the arrival process is modulated by a four-state Markov chain. The sojourn time of each modulating state is still 50, the arrival rate associated with each modulating state is 1.5, 3.5, 5.5, 7.5, respectively. By simulation, $P_K = 7.6 * 10^{-3}$. So method 3) above gives a good estimate.

Let us now return to the COSAGE-ATCAL-CEM model. The key question is how to do the grouping. The answer of this question depends on the computational power available to us. Basically we could have four levels of grouping.

- Level 1: Group according to time: first, second, and last period of a battle;
- Level 2: Group according to the ATCAL I output;
- Level 3: Group according to several key quantities in COSAGE output;
- Level 4: Clustering analysis of COSAGE output (using large neural net).

Note that in this approach we also need to estimate the probability of each group. For example, in dealing with different initial mixture, we need to assume these probabilities. To estimate these probabilities from simulated data we need to perform clustering analysis. We consider the following clustering analysis algorithm via iterative partitioning.

- **Iterative Partitioning Clustering Analysis Algorithm :**

1. Select an initial partition with K clusters;
   Repeat Steps 2 through 5 until the cluster membership stabilizes.
2. Generate a new partition by assigning each pattern to its closest cluster center;
3. Compute new cluster centers as the centroids of the clusters;
4. Repeat steps 2 and 3 until an optimum value of the criterion function is found;
5. Adjust the number of clusters by merging and splitting existing clusters or by removing small, or outlier, clusters.
6. Cluster validation: both statistical methods and expert knowledge are needed.

References [30,31] give more details on clustering analysis algorithms.

It is also possible to use a neural network for clustering analysis, since clustering analysis is essentially an optimization problem. We need to explore this type of clustering algorithm in future research. The following subsection describes a large stochastic neural network as a tool to "learn" the statistics of the high resolution battle simulation model COSAGE. Such large stochastic neural

network could also be useful in clustering analysis.

5.5. Use of a Large Stochastic Neural Net to Learn COSAGE

In the scenario grouping approach described above we need to run COSAGE many times to obtain the statistical data. It is very difficult to store these data in appropriate data structures. It is therefore useful if we can build a device that can generate data that obey the same distribution as COSAGE does very quickly when we need them. We propose to use a large size stochastic neural network to do this. The basic idea is like the following:

The terminology "learning" here is the same as "estimation of distributions from samples". We need to estimate the distribution of the COSAGE output. In other words, we need to "draw" the histogram of the COSGAE output. As [32,33] point out, a multilayer feedforward neural network is the only tool to circumvent the complexity problem.

The following is the procedure for using a neural network to obtain the output distribution from the high resolution simulation samples.

- Generate COSAGE samples "off-line" to train the learning system;
- Replacing COSAGE by a trained learning system in "on-line" simulation;
- Learning time would not be excessive: after certain time the "fresh data" from COSAGE do not add much. Higher level dynamics in CEM dominates the final output distribution.
- Verification of the trained model: Use pairwise joint distribution test.

We describe such a stochastic neural network proposed by Wong in 1991 which he named "diffusion machine" [34]:

• **Diffusion Machine** [34]: The state of the machine at time t is denoted by

$$v_i(t),\ i = 1,2,\ldots,\ n;$$

and the output of the neuron is

$$ui = g(v_i),\quad i = 1,2,\ldots,n;$$

$$dv_i = -\frac{\partial E[v(t)]}{\partial v_i}dt + \sqrt{\frac{2T}{g'(ui(t))}}\,dWi(t)$$

where $E[v(t)]$ is the function to be optimized.

Following are some of the examples for function g :

1. $g(x) = \frac{1}{2}(1 + tanh(\frac{x}{a})$

2. $$g(x) = \begin{cases} 1-\dfrac{1}{2(1-x/a)} & x>0 \\ \dfrac{1}{2(1+x/a)} & x<0 \end{cases}$$

To implement the neural network on a digital machine we have to first discretize the stochastic differential equation involved. We adopt the approach developed by Kloeden and Platen [28]. The discretized stochastic differential equation becomes:

$$u_{ij}(k+1) = u_{ij}(k) - \frac{\partial E[v(t)]}{\partial v_i}\Delta t + 2\sqrt{aT}\,(1+|\frac{u_{ij}(k)}{a}|)\,\Delta w(k)$$

$$\Delta w(k) = r(k)\sqrt{\Delta t};$$

$r(k) \in \{+1,-1\}$ is an i.i.d. random sequence with two point distribution.

$a = 0.02$, $\Delta t = 0.01$ or 0.001.

To verify the usefulness of the approach with high dimensional distributions we apply this neural network in an image segmentation problem where the image has 10x10 pixels:

$$V=\{v_{ij},\ i,j,=1,...,10\}$$

The objective function to be minimized is chosen as

$$E(v) = \frac{1}{2\sigma^2}\sum_{ij} v_{ij}^2 - \frac{1}{\sigma^2}\sum_{ij} v_{ij}y_{ij} + \beta[\sum_{i=1}^{9}\sum_{j=1}^{8}(v_{ij}-v_{i,j+1})^2$$
$$+\sum_{i=1}^{9}\sum_{j=1}^{9}(v_{ij}-v_{i,j-1})^2 + \sum_{i=1}^{8}\sum_{j=1}^{9}(v_{ij}-v_{i+1,j})^2 + \sum_{i=1}^{9}\sum_{j=1}^{9}(v_{ij}-v_{i+1,j})^2$$
$$+\lambda\sum_{i=1}^{9}\sum_{j=1}^{9} v_{ij}(1-v_{ij})$$

The segmentation results are quite good, verifying that the neural network based on the diffusion machine proposed by Wong could indeed work well with high dimensional random vectors.

• **A Three Network Learning System:** As Wong proposed, learning, or estimation of a high dimensional distribution, can be achieved by a device consisting of three connected stochastic neural networks. Each network has n nodes, divided into 2 groups:

$$V = \{v_i, i = 1,2,...,n_v\} : \text{Visible Nodes}$$

$$H = \{h_i, i = n_{v+1}, \ldots, n\} : \text{Hidden Nodes}$$

The goal is to find $\{w_{ij}\}$ for the "free running network" such that the stationary $\tilde{p}_0(v;w)$ is as close as possible to a pre-given $p_0(v;w)$.

• **Mathematical formulation of the learning network**: The joint density on VxH for one network is $p_0(v,h;w)$, $v \in V$, $h \in H$, w is the weight. The marginal density on V for one network is

$$p_0(v;w) = \int_H p_0(v,h;w)dh.$$

Our goal is to find w to minimize the "Kullback information"

$$G(w) \triangleq \int_H \tilde{p} \ln\left[\frac{\tilde{p}(v)}{p_0(v;w)}\right]dv.$$

Probability densities of the system can be written in terms of the "energy function" $E(v,h;w)$:

$$p_0(v,h;w) = \frac{e^{-\frac{1}{T}E(v,h;w)}}{\int_{VxH} e^{-\frac{1}{T}E(v,h;w)}dvdh}$$

and the marginal density on v is

$$p_0(v;w) = \int_H p_0(v,h;w)dh = \frac{\int_H e^{-\frac{1}{T}E(v,h;w)}dh}{\int_{VxH} e^{-\frac{1}{T}E(v,h;w)}dvdh} = \frac{A}{B}.$$

If $E(v,h;w)$ is a quadratic function of v,h then we are using Gaussian density to approximate the real density.

• **Derivation of the Learning Operation:** Note

$$\frac{p_0'(v;w)}{p_0(v;w)} = \frac{\frac{\partial p_0(v;w)}{\partial w}}{p_0(v;w)} = \frac{\frac{AB' - BA'}{B^2}}{\frac{A}{B}} = \frac{B'}{B} - \frac{A'}{A}.$$

We have

$$G(w) = \int_V \tilde{p}(v) \ln\left[\frac{\tilde{p}(v)}{p_0(v;w)}\right]dv$$

and

$$\frac{\partial G(w)}{\partial w} = \int_V \tilde{p}(v) \ln\left[-\frac{p_o'(v;w)}{p_o(v)}\right] dv = \int_V \tilde{p}(v)\left[\frac{A'}{A} - \frac{B'}{B}\right]$$

Or,

$$\frac{\partial G(w)}{\partial w} = \frac{1}{T}\int_{VxH}\left[\frac{\partial E(v,h;w)}{\partial w_{ij}}\left[p_o(v,h;w) - p(v)\frac{\tilde{p}_o(v,h;w)}{p_o(v;w)}\right]\right] dvdh$$

$$= \frac{1}{T}\left[E_0\left(\frac{\partial E}{\partial w_{ij}}\right) - \tilde{E}\left(\frac{\partial E}{\partial w_{ij}}\right)\right]$$

Then, we can use the Langevin Machine [34] to minimize G(w):

$$dw_{ij}(t) = -\frac{\partial G(w)}{\partial w_{ij}} dt + \sqrt{2s} dz_{ij}(t).$$

Preliminary experiments have shown that this approach is indeed effective. However, substantially more work in both theory and experiments need to be done before this approach can be effectively used in practice.

Although the learning machine based on Wong's diffusion network could be very useful in collecting distribution information from the samples for high dimensional random vectors, it has to be implemented on parallel machines to be efficient. Actually many similar research efforts have been devoted to the sequential simulation study of neural networks whose unique advantage is the parallelism ! Since it is not feasible to use big parallel machines for our purposes (also due to the realization that big, general purpose parallel machines may not be very efficient for codes that are only partially parallelizable), we have devoted significant effort to the development of a special purpose parallel computing device, based on the Digital Differential Analyzer (DDA) proposed by some Russian scientists in the 1960s for other purposes. We first discuss how to use DDA to implement the basic component in a neural network, the sigmoid function.

• **DDA Implementation of Sigmoid Function:** We choose the sigmoid function as

$$g(u) = \frac{1}{1+e^{-u}}.$$

This is illustrated in the following **Figure 47**.

The basic idea of the Digital Differential Analyzer (DDA) can be described as follows. Suppose we want to calculate the function $y = g(u)$. In the DDA computation, one performs

$$dy = g'(u)\, du\ .$$

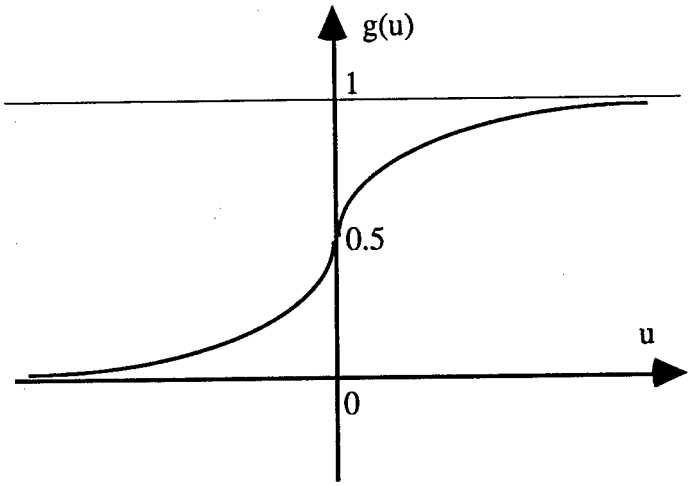


Figure 47. Sigmoid Function

For a simple but useful example let's look at the function $y = e^{u}$. In the DDA computation one does the following instead:

$$dy = e^{u}\, du = y\, du.$$

In the digital implementation one uses the following difference approximation

$$y_n = y_{n-1} + y_{n-1} du$$

with, for example, the following two possible choices of du: :

binary: $du = \{+0.0001, -0.0001\}$,

ternary: $du = \{+0.0001, 0, -0.0001\}$.

The DDA implementation of a sigmoid function can be derived as follows:

$$y = 2g(u) - 1 = \frac{1 - e^{-u}}{1 + e^{u}}$$

$$dy = [\frac{1}{2}(1 + y)(1 - y)]du = wdu$$

$$dw = -ydy$$

$$v = g(u) \quad \begin{cases} v = (y+1)/2 \\ dy = wdu \\ dw = -ydy \end{cases}$$

This implementation can be illustrated in the block diagram shown in **Figure 48**. This DDA sigmoid function was used in the image segmentation problem and the results are very good.

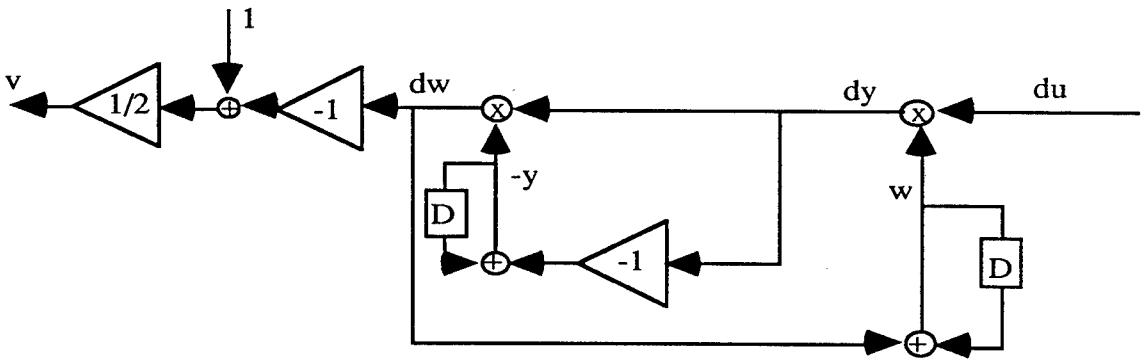


Figure 48. Block Diagram for DDA Implementation of A Sigmoid Function

5.6 Summary.

Through this project we have gained a thorough understanding of the importance of preserving the stochastic fidelity in hierarchical battle simulation models. The Lanchester equation and similar equations can only be used for average values and has little value in preserving the stochastic fidelity. Ito type of stochastic equations can be used to enhance the deterministic Lanchester equation. However the computational complexity involved could be insurmountable. Scenario grouping approach seems to be computationally feasible and bears strong "physical" meaning to render it easy to implement in practice. On the other hand, since we will have to deal with random vectors with huge dimension, stochastic neural networks seem to be the only feasible computation tool for estimating the distribution of the output quantities of the high resolution simulation models. We recommend to implement Wong's diffusion machine on special purpose parallel processors (DDA) for such "learning" purposes. Such a stochastic neural network can also be used for clustering analysis for high dimensional stochastic data, which is also important for the scenario grouping approach. Based on our experience with applications in image processing we believe that the stochastic neural network as we described can be effectively used for the battle simulation models.

6. RATIONAL APPROXIMATIONS FOR STOCHASTIC DISCRETE EVENT SYSTEMS.

The rational approximation approach for performance analysis of stochastic discrete event systems including computer and communication networks is proposed in [35,36]. This approach may form powerful new tools for the metamodeling of large-size stochastic discrete event systems when the so-called "curse of dimensionality" occurs.

The curse of dimensionality is a major difficulty in the analysis and design of many computer/communication systems. Examples among many of such systems are: communication networks consisting of hundreds of nodes; multiprocessor computing systems having thousands of processors; closed queueing systems with the circulating population in the hundreds or thousands; models using phase type distributions involving large or infinite dimensional Markov chains; etc. The essential difficulty to analyze these systems is that the computational complexity grows rapidly with the size of the system or the dimension of the system model. They are otherwise quite tractable. In other words, the concerned performance function can be evaluated accurately when the system size is small. The situation looks so hopeless that the research effort in analytical modeling of such systems is fading away. Simulation becomes the standard approach in the research literature to justify the new designs. In this project, we tried to tackle this problem using the Newton-Pade Approximants. This approach is motivated by our earlier work on applying the Pade approximants to queueing networks [35] and is based on the following observations.

- Performance functions of many computer/communication systems have nice shapes as a function of the system size. They are very often provably monotonic, convex or concave, having known or easily obtainable asymptotic behavior when the system size goes to infinity;
- These functions can be evaluated via analytical formulas when the system size is small. The calculation becomes impractical only when the system size becomes large;
- For functions having features described in 1 and 2, it is possible to obtain simple approximants that are virtually exact for all engineering purposes. In other words, in these situations if we only ask for accurate approximants for large size systems we can have easy solutions to the otherwise impossible problems.

The idea is easy to understand intuitively. Think about a real-valued function defined on the set of non-negative integers, $f(n)$, $n = 0,1,2,\ldots$ Assume, for example, that we can prove this function is monotonically increasing and concave and converges to a known constant when $n \to \infty$. On top of these properties we also know, say, $f(0),f(1),\ldots,f(9)$ exactly. Imagine that we have to draw two different curves going through the 11 known points (including $f(\infty)$) exactly and that are both

monotonic and concave. It is clear that the requirement of monotonicity and concavity significantly narrows down the possibilities. The question is how to combine the given knowledge to form an easily calculable approximant. We have shown in [35-41] that the rational approximants fit a set of initial function values and the asymptotics are very effective.

The performance functions studied in [35-41] include waiting time in some classical queueing systems, processing power for multiprocessor systems, key quantities in the queue inference engine problem, cell-loss probability in high speed communication networks, among others. We believe that this approach could form a powerful tool for many performance analysis problems complementing simulation.

7. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS.

In this section, we summarize the main findings, lessons learned, and recommendations for future directions that have resulted from this project.

• **Concurrent and Parallel Simulation.** Computer simulation has emerged as the only tool of universal applicability when it comes to modeling complex systems. However, systematic design and performance studies of such systems require a large number of time-consuming simulation runs: to ask N "what if" questions, a nominal simulation run is needed, followed by N additional runs, one for each "what if". One of the main goals of this project has been to establish the feasibility of the idea that multiple "what if" questions can be answered from data obtained from *the nominal simulation run alone*. This approach has been termed "concurrent simulation" when applied to conventional sequential computers, in contrast to "parallel simulation" when applied to parallel computers or networks of workstations. We believe this feasibility objective was successfully met through (a) The development of two specific algorithms for implementing concurrent simulation, the *Standard Clock* (SC) method (see Section 2.4.1) and *Augmented System Analysis* (ASA) (see Section 2.4.2), and (b) Software demonstrations and explicit numerical results indicating significant speedup factors provided by these algorithms over conventional repetitive simulation (see Section 3).

We summarize next some lessons we have learned from this work, which also suggest directions for future research:

(1) Implementation of the conceptual algorithms we developed is critical. We have found that the speedups achieved by concurrent simulation are significantly affected by such factors as the particular computer hardware used and the data structures selected. This suggests the need for a synergistic collaboration of hardware and software expertise with the analytical expertise required to develop new simulation techniques.

(2) Parallel simulation (as defined above) promises at least one additional order of magnitude in speedup over concurrent simulation. In this project, we were limited to the use of sequential computers, which gave us speedup factors in the range of 5 to 10 (see Section 3.2 and 3.3). It became clear, however, that the state update component of a simulation process (the one we cannot "parallelize" through concurrent simulation) can become large; yet, this is precisely the component which can be easily and naturally "parallelized" through the use of parallel processors. The speedup analysis performed in Section 2.5 substantiates this observation (see also [18]). We believe that the investigation of our techniques implemented in a parallel computing environment is a direction that holds great promise.

(3) The two main concurrent simulation algorithms studied in this project impose some limitations on the nature of the event lifetime distributions allowed. In particular, while the structure of systems to be simulated may be arbitrarily complex, the SC and ASA methods are based on Markovian assumptions (with some extensions possible, e.g., see [9],[19]). Structural complexity is indeed the major challenge we need to confront, compared to distributional information. What we have found is typical in the development of engineering methodologies: there is a tradeoff between the generality of a technique and its computational efficiency. In the late stages of this project, we initiated the study of a new "universal" concurrent simulation scheme (see Section 2.6) based on a "time warping" idea. The price to pay for such "universal" applicability is a reduction in efficiency, measured through the speedup factor we have defined in this project. Additional research remains to be carried out in this direction, in order to (a) Fully develop explicit algorithms, (b) Study their properties and limitations, and (c) Efficiently implement them, especially in view of observation (1) above.

• **System Optimization.** The ultimate goal in designing complex systems is to optimize their performance. When simulation is used in this process, traditional techniques rely on computing the *cardinal* values of a performance measure over all possible alternatives. In contrast, *ordinal* optimization is driven by the *relative order* of estimates of the objective function -- not their absolute values. The advantage here is that we can exploit inherent robustness properties of these order statistics with respect to substantial estimation noise. In simple terms, the idea we set out to explore in this project is "why waste time to get "good" performance estimates, when relatively "poor" but quickly obtained estimates can be provably adequate to order these performance estimates?" Our results, based on a testbed problem (see Section 3.1), indicate that this novel way of viewing optimization processes indeed provides powerful means for solving combinatorially hard problems. Based on this principle, two specific types of algorithms were presented, *Stochastic Comparison* (STC) (see Section 4.2) and *Stochastic Descent* (see Section 4.3). Both exploit concurrent simulation by extracting performance estimates over multiple alternatives from a single simulation run.

This component of the project has led to some findings of interest which are summarized next:

(1) The combination of ordinal optimization principles and concurrent simulation holds the promise of a new paradigm for simulation-based system optimization, one where so far intractable problems may be solved. Even a simple procedure (such as the one described in Section 4.1) readily revealed the power of this approach to rapidly identify a small subset of alternatives within which the optimal solution to a complex problem lies with high probability.

The more sophisticated techniques developed in Sections 4.2 and 4.3 systematize this approach. Clearly, more research is needed to gain a better understanding of the capabilities and limitations of ordinal optimization and to develop a rigorous and complete framework.

(2) Ordinal optimization involves a basic engineering tradeoff between "near optimality" and "computational complexity": through a small compromise in the quest for "the optimal" large gains result in the effort to find a "good enough" solution. "Near optimality" means that a solution within $x\%$ of the optimal with probability y may be found. "Computational complexity" is measured by the amount of time required to reach such a solution. The crucial finding is that this relationship is highly nonlinear: a very small value of x combined with a value of y very close to 1 can provide a reduction in computational complexity of orders of magnitude. This has been demonstrated in a number of application areas (e.g., see selected papers in [18]). It is precisely this tradeoff that needs to be further studied and explicitly quantified, a task that we see as the main challenge in this area.

(3) Although the key principles of this approach are simple to apply, rigorous analysis involves a high degree of technical complexity. In this report, we have spared the reader of technical details (e.g., in proving convergence of the algorithms developed), a flavor of which may be found in related published work [22]-[25].

• **Stochastic Fidelity in Hierarchical Simulation.** In this project, we studied the hierarchical decomposition of a simulation model as one way to reduce complexity. The analysis of a concrete hierarchical battle simulation model convinced us that the preservation of stochastic fidelity from the high resolution simulator to the more abstract attrition calculation is the crucial issue in hierarchical modeling for large stochastic simulation models. The challenge here is to deal with a huge dimensional model under realistic computation time constraints. Our main recommendation is to use the scenario grouping approach to combine data analysis and expert opinions on the weighting probabilities for each scenario that could be generated by the high-resolution simulator. Such weighting probabilities could also be obtained (more objectively) by some learning and clustering analysis procedures using large size stochastic neural networks. We recommend to implement such stochastic neural networks on special purpose parallel processors made of Digital Differential Analyzers (DDA), since mainstream general-purpose parallel computers are not easily available; in addition, the overhead involved in implementing such special neural networks on general-purpose parallel machines could be unrealistically high. Our preliminary experiments support this recommended approach. We suggest to pursue the directions described in the report, especially a concrete implementation of the scenario grouping approach to

the battle simulation model and the application of stochastic neural networks in learning and clustering analysis for the data generated from this model.

• **Other Means for Tackling System Complexity.** Concurrent/parallel simulation, ordinal optimization, and hierarchical decomposition all represent efforts to tackle the basic problem of system complexity. One component of this project was devoted to exploring alternative methods. One such method is based on the use of rational approximation techniques. We introduced these techniques in the context of computationally complex performance analysis problems in stochastic discrete event systems, such as computer systems and communication networks. Analysis and experiments show that this approach could form a powerful tool for metamodeling of stochastic discrete event systems. In this report, we limited ourselves to a brief overview in Section 6; technical details may be found in the published literature [35]-[41].

Clearly, further research on the theoretical foundations and more concrete application examples based on rational approximation techniques are needed, which were not within the scope of this project. In the course of this work, however, we were led to some observations we hope will form the basis of further investigation:

(1) Metamodeling techniques based on simulation are well worth exploring (see also [45]-[47]) and may hold significant promise when combined with some of the ideas discussed in this report.

(2) Other ways of decomposing a complex system are also worth exploring. In particular, besides their *hierarchical* structure, another common feature of C^3I systems is their *decentralized* nature (i.e., the fact that such systems usually consist of numerous interconnected components intended to operate autonomously whenever possible). Although we did not pursue this direction, we carried out some work [48] demonstrating the natural use of perturbation analysis techniques in this context.

• **Interactive Use of Simulation.** As pointed out earlier, the implementation of many of the techniques studied and developed in this project is an important and nontrivial task, greatly dependent on hardware and software technologies at one's disposal. Although we demonstrated the interactive use of some of these techniques using a commercial simulation environment (see Section 3.4), this task was outside the scope of the project. We do wish to stress, however, that concurrent simulation and ordinal optimization methods are ideally suited for emerging technologies, such as parallel processing, and new software capabilities that include Object Oriented Programming (OOP) and Graphical User Interfaces (GUI).

REFERENCES

[1] Ho, Y.C., and Cao, X., *Perturbation Analysis of Discrete Event Dynamic Systems,* Kluwer Publishing Co., 1991.

[2] Glasserman, P., *Gradient Estimation Via Perturbation Analysis*, Kluwer Publishing Co., 1991.

[3] Cassandras, C.G., *Discrete Event Systems: Modeling and Performance Analysis*, Irwin Publ., 1993.

[4] Ho, Y.C., Sreenivas, R., and Vakili, P., "Ordinal Optimization of Discrete Event Dynamic Systems", *Journal of Discrete Event Dynamic Systems*, Vol. 2, 2, pp. 61-88, 1992.

[5] Law, A.M., and Kelton, W.D., *Simulation Modeling and Analysis*, 2nd Edition, McGraw-Hill, 1991.

[6] Bratley, P.B., Fox, B.L., and Schrage, L.E., *A Guide to Simulation*, 2nd Edition, Springer-Verlag, New York, 1987.

[7] Ziegler, B.(Ed.), Special Issue on Specification Methods for Evaluation of Discrete Event Dynamic Systems, *Journal of Discrete Event Dynamic Systems*, Vol. 3, 2/3, 1993.

[8] Cassandras, C.G., and Strickland, S.G., "On-Line Sensitivity Analysis of Markov Chains," *IEEE Trans. on Automatic Control*, AC-34, 1, pp. 76-86, 1989.

[9] Cassandras, C.G., and Strickland, S.G., "Observable Augmented Systems for Sensitivity Analysis of Markov and Semi-Markov Processes," *IEEE Trans. on Automatic Control*, AC-34, 10, pp. 1026-1037, 1989.

[10] Cassandras, C.G., and Strickland, S.G., "Sample Path Properties of Timed Discrete Event Systems", *Proceedings of the IEEE,* Vol. 77, 1, pp. 59-71, 1989.

[11] Vakili, P., "A Standard Clock Technique for Efficient Simulation", *Operations Research Letters* , 10, pp. 445-452, 1991.

[12] Ho, Y.C., Cassandras, C.G., and Makhlouf, M., "Parallel Simulation of Real-Time Systems via the Standard Clock Approach", *J. of Math. and Comp. in Simulation*, Vol. 35, pp. 33-41, 1993.

[13] Chen, C.H. and Ho, Y.C., "An Approximation Approach to the Standard Clock Method for General Discrete Event Simulation" *IEEE Trans. on Control Technology*, Vol. 3, 3, pp. 309-317, 1995.

[14] Hu, J.Q., "Parallel Simulation of DEDS via Event Synchronization", *J. of Discrete Event Dynamic Systems*, Vol. 5, 2/3, pp. pp. 167-186, 1995.

[15] Strickland, S.G., and Phelan, R.G., "Massively Parallel SIMD Simulation of Markovian DEDS: Event and Time Synchronous Methods", *J. of Discrete Event Dynamic Systems*, Vol. 5, 2/3, pp. pp. 141-166, 1995.

[16] Glasserman, P., and Vakili, P., "Comparing Markov Chains Simulated in Parallel", *Proabability in Engin. and Info. Sciences*, to appear, 1995.

[17] Fujimoto, R.M., "Parallel Discrete Event Simulation," *Comm. of ACM* , 33(10), pp. 31-53, 1990.

[18] Cassandras, C.G. (Ed.), "Parallel Simulation and Optimization of Discrete Event Systems", *J. of Discrete Event Dynamic Systems*, Vol. 5, 2/3, 1995.

[19] Cassandras, C.G., and Pan, J. "Parallel Sample Path Generation for Discrete Event Systems and the Traffic Smoothing Problem", *Journal of Discrete Event Dynamic Systems*, Vol. 5, 2/3, pp. 187-218, 1995.

[20] Jefferson, D., and Sowizral, H., "Fast Concurrent Simulation Using the Time warp Mechanism", *Proceedings of SCS Dist. Sim. Conf.*, pp. 63-69, 1985.

[21] Ho, Y.C., and Larson, M.E., "Ordinal Optimization Approach to Rare Event Probability Problems", *Journal of Discrete Event Dynamic Systems*, *Journal of Discrete Event Dynamic Systems*, Vol. 5, 2/3, pp. 281-302, 1995.

[22] Gong, W.B., Ho, Y.C. and Zhai, W.G., "Stochastic Comparison Algorithm for Discrete Optimization with Estimation", *Proc. 31st IEEE Conf. on Decision and Control*, pp. 795-800, 1992.

[23] Cassandras, C.G., and Pan, J., "Some New Approaches to Discrete Stochastic Optimization Problems", to appear in *Proc. 34th IEEE Conf. Decision and Control*, 1995.

[24] Bao, G., and Cassandras, C.G., "A Stochastic Comparison Algorithm for Continuous Optimization with Estimation", to appear in *J. of Optimization Theory and Applic.*, 1994.

[25] Cassandras, C.G., and Julka, V.,, "Descent Algorithms for Discrete Resource Allocation Problems", *Proc. 33rd IEEE Conf. Decision and Control*, pp. 2639-2644, 1994.

[26] Ingber, L., "Statistical Mechanics of Combat and Extensions", Manuscript, 1993.

[27] Stiller, P.F., "Applications of Differential Geometry and Stochastic Differential Equations to Selected Problems in Combat Analysis", Manuscript, 1992.

[28] Kloeden, E. and Platen, J., *Numerical Solution of Stochastic Differential Equations*, Springer-Verlag, 1992.

[29] Robinson, S., "Scenario Analysis : What it is and how it works", Manuscript, University of Wisconsin at Madison, May 1990.

[30] Jain, A.K. and Dubes, R.C., *Algorithms for Clustering Data*, Prentice Hall, 1988

[31] Luger, G.F. and Stubblefield, W.A., *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*, Benjamin/Cummings Publishing Co. 1993

[32] Jones, L.K., "A Simple Lemma on Greedy Approximation in Hilbert Space and Converngence Rates for Projection Pursuit Regression and Neural Network Training", *Ann. Statistics*, Vol.20, No. 1, pp.608-613, 1992

[33] Barron, A.R., "Universal approximation bounds for superpositions of a sigmoidal function", Technical Report 58, 1991, Dept of Statistics, Univ. Illinois.

[34] Wong, E. "Stochastic Neural Networks," *Algorithmica* , v.6, pp. 466-478, 1991.

[35] Gong, W.B., Nananukul, S., and Yan, A., "Padé Approximations for Stochastic Discrete Event Systems", *IEEE Trans. on Automat. Control* , Vol.40, No. 8, August 1995.

[36] Gong, W.B. and Yang, H. "Global Rational Approximation for Stochastic Discrete Event Systems", To appear, *IEEE Transactions on Computers*, November 1995.

[37] Nananukul, S. and Gong, W.B., "On the MacLauring Series in Light Traffic for a Single Sever Queue", *Journal of Applied Probability*, March 1995.

[38] Gong and H. Yang, "On Global Rational Approximants for Stochastic Discrete Event Systems", *Proceedings of the 33rd IEEE Conference on Control and Decision,* Lake Buena Vista, Florida, Dec., 1994.

[39] Gong, W.B. and Nananukul, S., "Approximation of Performance Measures Using Extrapolation Methods", *Proceedings of the 33rd IEEE Conference on Control and Decision,* Lake Buena Vista, Florida, Dec., 1994.

[40] Yang, H. and Gong, W.B., "A New approach to Calculate Normalization Constants in Queueing Networks", *Proceedings of the 33rd IEEE Conference on Control and Decision,* Lake Buena Vista, Florida, Dec., 1994.

[41] Bao, G., and Cassandras, C.G., "A Rational Approximation Approach to Rare Event Probability Estimation for High-Performance Systems", to appear in *34th IEEE Conf. Decision and Control*, 1995.

[42] Zhai, W., P. Kelly and W.B. Gong, "Genetic Algorithms with Noisy Fit Functions", To appear, *Mathematical and Computer Modeling*, 1995.

[43] Aarts, E., and Korst, J., *Simulated Annealing and Boltzmann Machines*, Wiley, 1989.

[44] Yan, D., and Mukai, H., "Stochastic Discrete Optimization", *SIAM J. on Control and Optimization*, Vol. 30, 3, pp. 549-612, 1992.

[45] Johnson, R.E. et al, "Stochastic Concepts Evaluation Model", Technical Paper CAA-TP-91-6, US Army Concept Analysis Agency, August 1991.

[46] Frantz, F.K., and Ellor, A.J., "Model Abstraction Techniques", Computer Sciences Corp. Technical Report CDRL A003, January 1995.

[47] Zeimer, M.A., Tew, J.D., Sargent, R.G., and Sisti, A., "Metamodel Procedures for Air Engagement Simulation Models", Rome Lab. IRAE Technical Report, 1993.

[48] Vazquez-Abad, F.J., Cassandras, C.G., and Julka, V., "Centralized and Decentralized Asynchronous Optimization of Stochastic Discrete Event Systems", subm. to *IEEE Trans. on Automatic Control*, 1995.

APPENDIX

Computer Code for Selected Concurrent Simulation and Optimization Algorithms

```
/*************************************************************************
*                                                                        *
*       SC algorithm of  multithread sample path generation              *
*       for systems consisting of parallel M/M/1/K queues                *
*                                                                        *
*       Performance of interest: blocking probability                    *
*                                                                        *
*                                                                        *
*       Jie Pan                  Last revised: 9/27/95                   *
*                                                                        *
*       CODES Laboratory                                                 *
*       Department of Electrical and Computer Engineering                *
*       University of Massachusetts                                      *
*       Amherst, MA 01003                                                *
*                                                                        *
*************************************************************************

        ~/post/nwang/simu/sc/para/pan/fig/para_sc.c

        Stopping criterion:     The total number of events, i.e., both
                                arrivals (including rejected ones) and
                                departures, in each sample path of the
                                constructed systems is 'EV_NUM'.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUM_Q   6
#define TOTAL_B 20
#define MAX_SP  10001

#define ARRIVAL 6
#define TRUE    1
#define FALSE   0

struct event{
        double evtime;
        int evtype;
        } *evnext;

int SP_NUM;             /* number of systems to be constructed */
int EV_NUM;             /* number of events for each systems */

int complete, finish[MAX_SP];

long int num_ev[MAX_SP];
long int num_waste;
int num_a[MAX_SP][NUM_Q], num_d[MAX_SP][NUM_Q];
int buf[MAX_SP][NUM_Q], q_l[MAX_SP][NUM_Q];

int SIZEV;

double tnow;
double lambda;
double mu[NUM_Q] = {3.0, 2.0, 2.0, 3.0, 3.0, 2.0};

double RATE;                    /* RATE = SUM(mu[i]) + lambda */
int evtype[25]={0, 0, 0,        /* departure from Q1 */
                1, 1,           /* departure from Q2 */
                2, 2,           /* departure from Q3 */
                3, 3, 3,        /* departure from Q4 */
                4, 4, 4,        /* departure from Q5 */
                5, 5,           /* departure from Q6 */
```

```
                6, 6, 6, 6, 6,  /* external arrival */
                6 ,6, 6, 6, 6}; /* external arrival */

double unif();
double expo();

void init();
void arrv();
void arrv_to_q();
void dept();
void gene_event();
void gene_random_alloc();

int route[10] = {0, 1, 1, 2, 3, 3, 4, 4, 5, 5};

long int ISEED[5] = {1397, 2171, 5171, 7147, 9913};

main()
{
        int i;
        long int tem, sem;

        init();

        while (complete < SP_NUM)
                {
                tnow = evnext->evtime;

                switch(evnext->evtype)
                        {
                        case ARRIVAL:   arrv();
                                        break;
                        default:        dept(evnext->evtype);
                                        break;

                        }

                gene_event();
                }

        tem = 0;
        for (i = 0; i < SP_NUM; i++) tem += num_ev[i];
        sem = (int) (num_waste * 1.0 / SP_NUM);
        printf("\t %ld %ld %d%% \n", tem, sem, (int) (sem * 100.0 / tem));
                                /* record the sum of all real events in
                                the simulation run, the average number
                                of fictitious events per system, and the
                                percentage */

}

void init()
{
        int i, j;

        SIZEV = sizeof(struct event);

        scanf("%d", &SP_NUM);
        scanf("%d", &EV_NUM);

        gene_random_alloc(SP_NUM);

        num_waste = 0;
        for (i = 0; i < SP_NUM; i++)
                {
                finish[i] = FALSE;
                num_ev[i] = 0;
                for (j = 0; j < NUM_Q; j++)
```

```
                        {
                        q_l[i][j] = 0;
                        num_a[i][j] = 0;
                        num_d[i][j] = 0;
                        }
                }

        tnow = 0.0;
        lambda = 10.0;
        RATE = 25.0;    /* RATE = SUM(mu[i]) + lambda */

        complete = 0;

        evnext = (struct event *) malloc(SIZEV);
        gene_event();
}

void gene_event()
{
        double x;
        int key;

        x = expo(RATE, 3);
        evnext->evtime = tnow + x;
        key = (int) (25.0 * unif(0));   /* key is from 0 to 24 */
        evnext->evtype = evtype[key];
}

void arrv()
{
        double x;
        int pick_num;

        pick_num = (int) (10 * unif(1));        /* pick_num is from 0 to 9 */
        arrv_to_q(route[pick_num]);             /* arrive in which queue? */
}

void arrv_to_q(int qid)
{
        int i;

        for (i = 0; i < SP_NUM; i++)
                if (!finish[i])         /* if not finish */
                        {
                        num_ev[i]++;
                        num_a[i][qid]++;
                        if (q_l[i][qid] < buf[i][qid]) q_l[i][qid]++;

                        if (num_ev[i] == EV_NUM)
                                {
                                finish[i] = TRUE;
                                complete++;
                                }
                        }
}

void dept(int qid)
{
        int i;

        for (i = 0; i < SP_NUM; i++)
                if (!finish[i])
                        if (q_l[i][qid] > 0)
                                        /* if not a fictious event */
                                {
                                num_ev[i]++;
```

```
                                num_d[i][qid]++;
                                q_l[i][qid]--;

                                if (num_ev[i] == EV_NUM)
                                        {
                                        finish[i] = TRUE;
                                        complete++;
                                        }
                                }
                        else num_waste++;
                /* NOTE: this is an extra counter compared to ASA */
}

double unif(int i)
{
        long int a, b, c;
        double tem;

        a = ISEED[i] / 16384;
        b = ISEED[i] % 16384;
        c = (13205 * a + 74505 * b) % 16384;
        ISEED[i] = (c * 16384 + 13205 * b) % 268435456;

        tem = ISEED[i] / 268435456.0;
        return(tem);
}

void gene_random_alloc(int num)
{
        int i, j;
        double sum;
        double p[NUM_Q];

        for (i = 0; i < num; i++)
                {
                sum = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        p[j] = unif(2);
                        sum += p[j];
                        }
                buf[i][NUM_Q - 1] = TOTAL_B;
                for (j = 0; j < NUM_Q - 1; j++)
                        {
                        buf[i][j] = (int) (p[j] * TOTAL_B / sum);
                        buf[i][NUM_Q - 1] -= buf[i][j];
                        }
                for(j = 0; j < NUM_Q; j++) buf[i][j]++;
                }
}

double expo(double lambda, int i)
{
        double tem;

        tem = - 1.0 / lambda * log(unif(i));
        return(tem);
}
```

```
/*************************************************************************
*                                                                        *
*       ASA algorithm of multithread sample path generation              *
*       for systems consisting of parallel M/M/1/K queues                *
*                                                                        *
*       Performance of interest: blocking probability                    *
*                                                                        *
*                                                                        *
*       Jie Pan                  Last revised: 9/12/95                   *
*                                                                        *
*       CODES Laboratory                                                 *
*       Department of Electrical and Computer Engineering                *
*       University of Massachusetts                                      *
*       Amherst, MA 01003                                                *
*                                                                        *
*************************************************************************

        ~/post/simu/asa/asa.c

        In this 'max' version of ASA algorithm, the buffer size of each
        queue in the additional 'nominal' system is set to be the number
        of all buffers available to the whole system.

        Stopping criterion:     The total number of events, i.e., both
                                arrivals (including those rejected) and
                                departures, in each sample path of the
                                constructed systems is 'EV_NUM'.

        Buffer allocations for constructed systems may be generated
        randomly.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NUM_Q   6       /* number of queues in one system */
#define TOTAL_B 20      /* number of total buffers in one system */
#define MAX_SP  53131   /* maximum number of multithread systems */

#define ARRV    6
#define TRUE    1
#define FALSE   0

struct event{
        double evtime;
        int evtype;
        struct event *next;     /* one direct link */
        } *evlist;

int SP_NUM;             /* number of systems to be constructed */
int EV_NUM;             /* number of events for each system */

int IRN;

int complete, finish[MAX_SP];

long int num_ano, num_dno;
long int num_ev[MAX_SP];
int num_a[MAX_SP][NUM_Q], num_d[MAX_SP][NUM_Q], num_bk[MAX_SP][NUM_Q];
int bufno[NUM_Q], q_lno[NUM_Q];
int buf[MAX_SP][NUM_Q], q_l[MAX_SP][NUM_Q];

int SIZEV;
```

```
double tnow;
double lambda;
double mu[NUM_Q] = {3.0, 2.0, 2.0, 3.0, 3.0, 2.0};

double unif();
double expo();

double power();

void init();
void arrv();
void arrv_to_q();
void dept();
void gene_new_arrv();
void gene_new_dept();
void det_buf_alloc();
void rdm_buf_alloc();
void instevl();
void output();

int route[10] = {0, 1, 1, 2, 3, 3, 4, 4, 5, 5};

long int ISEED[11] = {1397, 2171, 5171, 7147, 9913, 4353, 3347, 6123,
                      8749, 5371, 2789};

main()
{
        struct event *evptr;
        int i;
        long int tem, sem;

        init();

        while (complete < SP_NUM)
                {
                evptr = evlist;          /* pick first event */
                evlist = evlist->next;   /* remove first event from list */
                tnow = evptr->evtime;    /* update time to next event time */

                switch (evptr->evtype)
                        {
                        case ARRV:       arrv();
                                         break;
                        default:         dept(evptr->evtype);
                                         break;
                        }

                free(evptr);
                }

        output();

/*      tem = 0;
        for (i = 0; i < SP_NUM; i++) tem += num_ev[i];
        sem = (int) ((num_ano + num_dno) * 100.0 / tem);
        printf("\t %ld %ld %ld%% \n", tem, num_ano, sem);
                                record the sum of all real events in the
                                simulation run, the actual number of
                                arrivals in the additional 'nominal'
                                sample path, and the percentage of the
                                actual number of arrivals and departures
                                in the 'nominal' over the first sum */
}

void init()
{
```

```
        int i, j;

        SIZEV = sizeof(struct event);

/*      scanf("%d", &IRN);
        scanf("%d", &SP_NUM);
        scanf("%d", &EV_NUM); */

        IRN = 4;
        SP_NUM = 1000;
        EV_NUM = 100000000;

/*      det_buf_alloc(); */

        rdm_buf_alloc(SP_NUM);

        num_ano = 0;
        num_dno = 0;
                /* NOTE: these two are extra counters compared to SC */
        for (i = 0; i < SP_NUM; i++)
                {
                finish[i] = FALSE;
                num_ev[i] = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        q_l[i][j] = 0;
                        num_a[i][j] = 0;
                        num_d[i][j] = 0;
                        num_bk[i][j] = 0;
                        }
                }

        for (j = 0; j < NUM_Q; j++) q_lno[j] = 0;

        tnow = 0.0;
        lambda = 10.0;
        evlist = NULL;

        complete = 0;

        gene_new_arrv();
}

void gene_new_arrv()
{
        double x;
        struct event *evptr;

        x = expo(lambda, 0);                    /* use seed 0 */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = ARRV;

        instevl(evptr);
}

void arrv()
{
        double x;
        int pick_num;

        pick_num = (int) (10 * unif(1));        /* pick_num is from 0 to 9 */
                                                /* use seed 1 */
        arrv_to_q(route[pick_num]);     /* arrive in which queue? */

        gene_new_arrv();
```

```
}

void arrv_to_q(int qid)
{
        int i;

        num_ano++;                              /* nominal sample path */
                /* NOTE: this is one of extra counters compared to SC */
        if (q_lno[qid] < bufno[qid])
                {
                q_lno[qid]++;
                if (q_lno[qid] == 1) gene_new_dept(qid);
                }

        for (i = 0; i < SP_NUM; i++)    /* perturbed sample paths */
                if (!finish[i])         /* if not finish */
                        {
                        num_ev[i]++;
                        num_a[i][qid]++;
                        if (q_l[i][qid] < buf[i][qid])
                                q_l[i][qid]++;
                        else
                                num_bk[i][qid]++;

                        if (num_ev[i] == EV_NUM)
                                {
                                finish[i] = TRUE;
                                complete++;
                                }
                        }
}

void gene_new_dept(int i)
{
        double x;
        struct event *evptr;

        x = expo(mu[i], 2);     /* use seed 2 */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = i;

        instevl(evptr);
}

void dept(int qid)
{
        int i;

        num_dno++;                                      /* nominal sample path */
                /* NOTE: this is one of extra counters compared to SC */
        q_lno[qid]--;
        if (q_lno[qid] > 0) gene_new_dept(qid);

        for (i = 0; i < SP_NUM; i++)                    /* perturbed sample paths */
                if (!finish[i])                         /* if not finish */
                        if (q_l[i][qid] > 0)
                                {
                                num_ev[i]++;
                                num_d[i][qid]++;
                                q_l[i][qid]--;

                                if (num_ev[i] == EV_NUM)
                                        {
                                        finish[i] = TRUE;
                                        complete++;
```

```
                                        }
                                }
}

void instevl(struct event *p)
{
        struct event *q;

        q = evlist;
        /* q points to the head of the list in which p inserted */

        if (evlist == NULL)     /* list is empty */
                {
                evlist = p;
                p->next = NULL;
                }
        else
                {
                if (p->evtime < evlist->evtime)         /* head of list */
                        {
                        p->next = evlist;
                        evlist = p;
                        }
                else                    /* middle or end of list */
                        {
                while ((q->next != NULL) && (q->next->evtime < p->evtime))
                                q = q->next;
                        p->next = q->next;
                        q->next = p;
                        }
                }
}

void output()
{
        int i, j, optimal;
        double qbkop[NUM_Q], qbk[NUM_Q];
        double sbkop, sbk[MAX_SP], sbk_theo[MAX_SP];
        double tem, sem;

        sbkop = 9.0;

        for (i = 0; i < SP_NUM; i++)
                {
                tem = 0.0;
                sem = 0.0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        if (!num_a[i][j])
                                qbk[j] = 1.0 * num_bk[i][j] / num_a[i][j];
                        else
                                qbk[j] = 0.0;

/*                      printf("buf[%d][%d] = %d \t qbk[%d][%d] = %f \n",
                                i, j, buf[i][j], i, j, qbk[j]); */

                        tem += num_a[i][j];
                        sem += num_bk[i][j];
                        }
                sbk[i] = sem / tem;
/*              printf("sbk[%d] = %lf \n \n", i, sbk[i]); */

                sbk_theo[i] = 0.1 * (2.0 / (power(3.0, buf[i][0] + 1) - 1.0)
                        + 1.0 / (power(2.0, buf[i][2] + 1) - 1.0))
                        + 0.2 * (1.0 / (buf[i][1] + 1.0)
                        + 0.5 / (power(1.5, buf[i][3] + 1) - 1.0)
```

```
                        + 0.5 / (power(1.5, buf[i][4] + 1) - 1.0)
                        + 1.0 / (buf[i][5] + 1));

                if (sbk[i] < sbkop)
                        {
                        optimal = i;
                        sbkop = sbk[i];
                        }
                }

/*      printf("The optimal buffer allocation is the following: \n");
        for (j = 0; j < NUM_Q; j++)
                printf("\t The buffer size of queue %d = %d \n", j,
                        buf[optimal][j]);
        printf("The optimal system blocking probability is %f \n", sbkop);
        printf("\t at the %d-th system \n \n", optimal); */

        for (i = 0; i < SP_NUM; i++)
                printf(" %f \n", sbk_theo[i]);

/*      for (i = 0; i < SP_NUM; i++)
                printf(" %f  %f  %f \t %d \t %d  %d  %d  %d  %d  %d \n",
                sbk_theo[i], unif(4), sbk[i], i, buf[i][0], buf[i][1],
                buf[i][2], buf[i][3], buf[i][4], buf[i][5]); */

        printf("\n %ld \t %ld \t %ld \n", IRN, SP_NUM, EV_NUM);
}

double unif(int i)
{
        long int a, b, c;
        double tem;

        a = ISEED[i] / 16384;
        b = ISEED[i] % 16384;
        c = (13205 * a + 74505 * b) % 16384;
        ISEED[i] = (c * 16384 + 13205 * b) % 268435456;

        tem = ISEED[i] / 268435456.0;
        return(tem);
}

void rdm_buf_alloc(int num)
{
        int i, j;
        double sum;
        double p[NUM_Q];

        for (j = 0; j < NUM_Q; j++) bufno[j] = TOTAL_B + 1;

        for (i = 0; i < num; i++)
                {
                sum = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        p[j] = unif(IRN);
                        sum += p[j];
                        }
                buf[i][NUM_Q - 1] = TOTAL_B;
                for (j = 0; j < NUM_Q - 1; j++)
                        {
                        buf[i][j] = (int) (p[j] * TOTAL_B / sum);
                        buf[i][NUM_Q - 1] -= buf[i][j];
                        }
         /* this is not a truely random allocation for each queue */
                for (j = 0; j < NUM_Q; j++) buf[i][j]++;
```

```
                }
}

void det_buf_alloc()
{
        int i, j, k, l, m, n, num_sys;
        double tem;

        for (i = 0; i < NUM_Q; i++) bufno[i] = TOTAL_B + 1;

        num_sys = 0;
        for (i = 0; i < TOTAL_B + 1; i++)
        for (j = 0; i + j < TOTAL_B + 1; j++)
        for (k = 0; i + j + k < TOTAL_B + 1; k++)
        for (l = 0; i + j + k + l < TOTAL_B + 1; l++)
        for (m = 0; i + j + k + l + m < TOTAL_B + 1; m++)
                {
                n = TOTAL_B - i - j - k - l - m;
                if (n >= 0)
                        {
                        buf[num_sys][0] = i + 1;
                        buf[num_sys][1] = j + 1;
                        buf[num_sys][2] = k + 1;
                        buf[num_sys][3] = l + 1;
                        buf[num_sys][4] = m + 1;
                        buf[num_sys][5] = n + 1;

/*      tem = 0.1 * (2.0 / (power(3.0, buf[num_sys][0] + 1) - 1.0) + 1.0
                / (power(2.0, buf[num_sys][2] + 1) - 1.0)) + 0.2 * (1.0
                / (buf[sys_num][1] + 1.0) + power(2.0, buf[sys_num][3])
                / (power(3.0, buf[sys_num][3] + 1) - power(2.0,
                buf[sys_num][3] + 1)) + power(2.0, buf[sys_num][4]) /
                (power(3.0, buf[sys_num][4] + 1) - power(2.0,
                buf[sys_num][4] + 1)) + 1.0 / (buf[sys_num][5] + 1));
        printf("\t %lf \n", tem); */

                        num_sys++;
                        }
                }

        SP_NUM = num_sys;
}

double expo(double lambda, int i)
{
        double tem;

        tem = - 1.0 / lambda * log(unif(i));
        return(tem);
}

double power(double base, int i)
{
        double tem;

        tem = 1;
        while (i-- > 0)
                tem = tem * base;
        return(tem);
}
```

```
/*************************************************************************
*                                                                        *
*       Generalized Stochastic Comparison Algorithm for Parallel         *
*       Queueing Systems.                                                *
*                                                                        *
*                                                                        *
*                                                                        *
*       Jie Pan                    Last revised: 9/27/95                 *
*                                                                        *
*       CODES Laboratory                                                 *
*       Department of Electrical and Computer Engineering                *
*       University of Massachusetts                                      *
*       Amherst, MA 01003                                                *
*                                                                        *
*************************************************************************


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_Q   6
#define TOTAL_B 20
#define MAX_SP 5000

#define ARRV    6
#define TRUE    1
#define FALSE   0
struct event{
        double evtime;
        int evtype;
        struct event *next;     /* one direct link */
        } *evlist;
int SP_NUM;
int EV_NUM;
int IRN;
int complete, finish[MAX_SP];
long int num_ano, num_dno;
long int num_ev[MAX_SP];
int num_a[MAX_SP][NUM_Q], num_d[MAX_SP][NUM_Q], num_bk[MAX_SP][NUM_Q];
int bufno[NUM_Q], q_lno[NUM_Q];
int buf[MAX_SP][NUM_Q], q_l[MAX_SP][NUM_Q];
int SIZEV;
double tnow;
double lambda = 10.0;
double mu[NUM_Q] = {3.0, 2.0, 2.0, 3.0, 3.0, 2.0};
int MK, OPTIMAL;
double SBKOP;

double unif();
double expo();
double power();
void initstart();
void initbegin();
void init();
void arrv();
void arrv_to_q();
void dept();
void gene_new_arrv();
void gene_new_dept();
void rdm_buf_alloc();
void rdm_buf_alloc_first();
void instevl();
void output();
int route[10] = {0, 1, 1, 2, 3, 3, 4, 4, 5, 5};
long int ISEED[11] = {1397, 2171, 5171, 7147, 9913, 4353, 3347, 6123,
```

```
                        8749, 5371, 2789};

main()
{
        struct event *evptr;
        int i, j, k;
        int ic, jc;
        double sbk[MAX_SP], tem, sem;

        initstart();
        for (k = 0; k < 1201; k++)
                {
                MK = 1 + k / 500;
                initbegin();
                for (j = 0; j < MK; j++)
                        {
                        init();

                        while (complete < SP_NUM)
                                {
                                evptr = evlist;
                                evlist = evlist->next;
                                tnow = evptr->evtime;
                                switch (evptr->evtype)
                                        {
                                        case ARRV:      arrv();
                                                        break;
                                        default:        dept(evptr->evtype);
                                                        break;
                                        }
                                free(evptr);
                                }

                        SBKOP = 9.0;
                        for (ic = 0; ic < SP_NUM; ic++)
                                {
                                tem = 0.0;
                                sem = 0.0;
                                for (jc = 0; jc < NUM_Q; jc++)
                                        {
                                        tem += num_a[ic][jc];
                                        sem += num_bk[ic][jc];
                                        }
                                sbk[ic] = sem / tem;
                                if (sbk[ic] < SBKOP)
                                        {
                                        OPTIMAL = ic;
                                        SBKOP = sbk[ic];
                                        }
                                }

                        if (OPTIMAL == 0) break; /* break loop j */

                        if (j == MK - 1)
                                for (i = 0; i < NUM_Q; i++)
                                        buf[0][i] = buf[OPTIMAL][i];
                        } /* end of loop j */

                output(k);
                } /* end of loop k */
}

void initstart()
{
        SIZEV = sizeof(struct event);
        IRN = 3;
```

```
        SP_NUM = MAX_SP * 2 - 2;
        EV_NUM = 500;
        rdm_buf_alloc_first();
}

void initbegin()
{
        SP_NUM = (int) ((float) (SP_NUM) / 2.0) + 1;
        rdm_buf_alloc(SP_NUM);
}

void init()
{
        int i, j;
        num_ano = 0;
        num_dno = 0;
        for (i = 0; i < SP_NUM; i++)
                {
                finish[i] = FALSE;
                num_ev[i] = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        q_l[i][j] = 0;
                        num_a[i][j] = 0;
                        num_d[i][j] = 0;
                        num_bk[i][j] = 0;
                        }
                }
        for (j = 0; j < NUM_Q; j++) q_lno[j] = 0;
        tnow = 0.0;
        evlist = NULL;
        complete = 0;
        gene_new_arrv();
}

void gene_new_arrv()
{
        double x;
        struct event *evptr;
        x = expo(lambda, 0);                    /* use seed 0 */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = ARRV;
        instevl(evptr);
}

void arrv()
{
        double x;
        int pick_num;
        pick_num = (int) (10 * unif(1));        /* pick_num is from 0 to 9 */
                                                /* use seed 1 */
        arrv_to_q(route[pick_num]);     /* arrive in which queue? */
        gene_new_arrv();
}

void arrv_to_q(int qid)
{
        int i;
        num_ano++;
        if (q_lno[qid] < bufno[qid])
                {
                q_lno[qid]++;
                if (q_lno[qid] == 1) gene_new_dept(qid);
                }
        for (i = 0; i < SP_NUM; i++)    /* perturbed sample paths */
```

```
                if (!finish[i])         /* if not finish */
                        {
                        num_ev[i]++;
                        num_a[i][qid]++;
                        if (q_l[i][qid] < buf[i][qid])
                                q_l[i][qid]++;
                        else
                                num_bk[i][qid]++;
                        if (num_ev[i] == EV_NUM)
                                {
                                finish[i] = TRUE;
                                complete++;
                                }
                        }
}

void gene_new_dept(int i)
{
        double x;
        struct event *evptr;
        x = expo(mu[i], 2);     /* use seed 2 */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = i;
        instevl(evptr);
}

void dept(int qid)
{
        int i;
        num_dno++;
        q_lno[qid]--;
        if (q_lno[qid] > 0) gene_new_dept(qid);
        for (i = 0; i < SP_NUM; i++)            /* perturbed sample paths */
                if (!finish[i])                 /* if not finish */
                        if (q_l[i][qid] > 0)
                                {
                                num_ev[i]++;
                                num_d[i][qid]++;
                                q_l[i][qid]--;
                                if (num_ev[i] == EV_NUM)
                                        {
                                        finish[i] = TRUE;
                                        complete++;
                                        }
                                }
}

void instevl(struct event *p)
{
        struct event *q;
        q = evlist;
        if (evlist == NULL)     /* list is empty */
                {
                evlist = p;
                p->next = NULL;
                }
        else
                {
                if (p->evtime < evlist->evtime)         /* head of list */
                        {
                        p->next = evlist;
                        evlist = p;
                        }
                else                    /* middle or end of list */
                        {
```

```
                while ((q->next != NULL) && (q->next->evtime < p->evtime))
                        q = q->next;
                p->next = q->next;
                q->next = p;
                }
        }
}

void output(index)
{
        double sbkop;

        sbkop = 0.1 * (2.0 / (power(3.0, buf[0][0] + 1) - 1.0) + 1.0 /
                (power(2.0, buf[0][2] + 1) - 1.0)) + 0.2 * (1.0 /
                (buf[0][1] + 1.0) + 0.5 / (power(1.5, buf[0][3] + 1) -
                1.0) + 0.5 / (power(1.5, buf[0][4] + 1) - 1.0) + 1.0 /
                (buf[0][5] + 1));

        printf("%d \t %lf \t %d %d %d %d %d %d \n", index, sbkop,
                buf[0][0], buf[0][1], buf[0][2], buf[0][3], buf[0][4],
                buf[0][5]);
}

void rdm_buf_alloc_first()
{
        int j;
        double sum;
        double p[NUM_Q];

        buf[0][NUM_Q - 1] = TOTAL_B;
        sum = 0;
        for (j = 0; j < NUM_Q; j++)
                {
                p[j] = unif(IRN);
                sum += p[j];
                }
        for (j = 0; j < NUM_Q - 1; j++)
                {
                buf[0][j] = (int) (p[j] * TOTAL_B / sum);
                buf[0][NUM_Q - 1] -= buf[0][j];
                }
        for (j = 0; j < NUM_Q; j++) buf[0][j]++;
}

void rdm_buf_alloc(int num)
{
        int i, j;
        double sum;
        double p[NUM_Q];

        for (j = 0; j < NUM_Q; j++) bufno[j] = TOTAL_B + 1;
        for (i = 1; i < num; i++)
                {
                buf[i][NUM_Q - 1] = TOTAL_B;
                sum = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        p[j] = unif(IRN);
                        sum += p[j];
                        }
                for (j = 0; j < NUM_Q - 1; j++)
                        {
                        buf[i][j] = (int) (p[j] * TOTAL_B / sum);
                        buf[i][NUM_Q - 1] -= buf[i][j];
                        }
                for (j = 0; j < NUM_Q; j++) buf[i][j]++;
```

```
                }
}

double expo(double lambda, int i)
{
        double tem;
        tem = - 1.0 / lambda * log(unif(i));
        return(tem);
}

double unif(int i)
{
        long int a, b, c;
        double tem;
        a = ISEED[i] / 16384;
        b = ISEED[i] % 16384;
        c = (13205 * a + 74505 * b) % 16384;
        ISEED[i] = (c * 16384 + 13205 * b) % 268435456;
        tem = ISEED[i] / 268435456.0;
        return(tem);
}

double power(double base, int i)
{
        double tem;
        tem = 1;
        while (i-- > 0) tem = tem * base;
        return(tem);
}
```

```
/*************************************************************************
*                                                                        *
*       Stochastic Descent Algorithm for Parallel Queueing Systems       *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*       Jie Pan                    Last revised: 9/27/95                 *
*                                                                        *
*       CODES Laboratory                                                 *
*       Department of Electrical and Computer Engineering                *
*       University of Massachusetts                                      *
*       Amherst, MA 01003                                                *
*                                                                        *
*************************************************************************


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_Q   6
#define TOTAL_B 20
#define MAX_SP 13

#define ARRV    6
#define TRUE    1
#define FALSE   0
struct event{
        double evtime;
        int evtype;
        struct event *next;     /* one direct link */
        } *evlist;
int SP_NUM;
int EV_NUM, EV_NUMA;
int IRN, JRN, KRN, LRN;
int complete, finish[MAX_SP];
long int num_ano, num_dno;
long int num_ev[MAX_SP];
int num_a[MAX_SP][NUM_Q], num_d[MAX_SP][NUM_Q], num_bk[MAX_SP][NUM_Q];
int bufno[NUM_Q], q_lno[NUM_Q];
int buf[MAX_SP][NUM_Q], q_l[MAX_SP][NUM_Q];
int SIZEV;
double tnow;
double lambda = 10.0;
double mu[NUM_Q] = {3.0, 2.0, 2.0, 3.0, 3.0, 2.0};
int MK, UOPTIMAL, DOPTIMAL;
double SBKOP;

double unif();
double expo();
double power();
void initstart();
void init();
void arrv();
void arrv_to_q();
void dept();
void gene_new_arrv();
void gene_new_dept();
void buf_alloc_no();
void buf_alloc_all();
void det_buf_alloc_first();
void rdm_buf_alloc_first();
void instevl();
void output();
int route[10] = {0, 1, 1, 2, 3, 3, 4, 4, 5, 5};
```

```
long int ISEED[11] = {1397, 2171, 5171, 7147, 9913, 4353, 3347, 6123,
                      8749, 5371, 2789};

main()
{
        struct event *evptr;
        int i, j, k;
        int ic, jc;
        double sbk[MAX_SP], tem, sem;

        buf_alloc_no();
        initstart();
        for (k = 0; k < 101; k++)
                {
                MK = 1 + k / 500;
                EV_NUMA = EV_NUM * MK;
                buf_alloc_all(SP_NUM);
                init();

                while (complete < SP_NUM)
                        {
                        evptr = evlist;
                        evlist = evlist->next;
                        tnow = evptr->evtime;
                        switch (evptr->evtype)
                                {
                                case ARRV:      arrv();
                                                break;
                                default:        dept(evptr->evtype);
                                                break;
                                }
                        free(evptr);
                        }

                for (ic = 0; ic < SP_NUM; ic++)
                        {
                        tem = 0.0;
                        sem = 0.0;
                        for (jc = 0; jc < NUM_Q; jc++)
                                {
                                tem += num_a[ic][jc];
                                sem += num_bk[ic][jc];
                                }
                        sbk[ic] = sem / tem;
                        }
                SBKOP = 9.0;
                for (ic = 0; ic < SP_NUM / 2; ic++)
                        {
                        if (sbk[2*ic+2] < SBKOP)
                                {
                                SBKOP = sbk[2*ic+2];
                                UOPTIMAL = ic;
                                }
                        }
                SBKOP = 9.0;
                for (ic = 0; ic < SP_NUM / 2; ic++)
                        if (sbk[2*ic+1] < SBKOP)
                                {
                                SBKOP = sbk[2*ic+1];
                                DOPTIMAL = ic;
                                }

                output(k);
                } /* end of loop k */
}
```

```
void initstart()
{
        SIZEV = sizeof(struct event);
        IRN = 4;
        JRN = 2;
        KRN = 3;
        LRN = 1;
        SP_NUM = MAX_SP;
        EV_NUM = 5000;
        det_buf_alloc_first();
/*      rdm_buf_alloc_first(); */
}

void init()
{
        int i, j;
        num_ano = 0;
        num_dno = 0;
        for (i = 0; i < SP_NUM; i++)
                {
                finish[i] = FALSE;
                num_ev[i] = 0;
                for (j = 0; j < NUM_Q; j++)
                        {
                        q_l[i][j] = 0;
                        num_a[i][j] = 0;
                        num_d[i][j] = 0;
                        num_bk[i][j] = 0;
                        }
                }
        for (j = 0; j < NUM_Q; j++) q_lno[j] = 0;
        tnow = 0.0;
        evlist = NULL;
        complete = 0;
        gene_new_arrv();
}

void gene_new_arrv()
{
        double x;
        struct event *evptr;
        x = expo(lambda, KRN);          /* use seed KRN */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = ARRV;
        instevl(evptr);
}

void arrv()
{
        double x;
        int pick_num;
        pick_num = (int) (10 * unif(JRN));      /* pick_num is from 0 to 9 */
                                                /* use seed JRN */
        arrv_to_q(route[pick_num]);     /* arrive in which queue? */
        gene_new_arrv();
}

void arrv_to_q(int qid)
{
        int i;
        num_ano++;
        if (q_lno[qid] < bufno[qid])
                {
                q_lno[qid]++;
                if (q_lno[qid] == 1) gene_new_dept(qid);
```

```
                }
        for (i = 0; i < SP_NUM; i++)     /* perturbed sample paths */
                if (!finish[i])          /* if not finish */
                        {
                        num_ev[i]++;
                        num_a[i][qid]++;
                        if (q_l[i][qid] < buf[i][qid])
                                q_l[i][qid]++;
                        else
                                num_bk[i][qid]++;
                        if (num_ev[i] == EV_NUMA)
                                {
                                finish[i] = TRUE;
                                complete++;
                                }
                        }
}

void gene_new_dept(int i)
{
        double x;
        struct event *evptr;
        x = expo(mu[i], LRN);    /* use seed LRN */
        evptr = (struct event *) malloc(SIZEV);
        evptr->evtime = tnow + x;
        evptr->evtype = i;
        instevl(evptr);
}

void dept(int qid)
{
        int i;
        num_dno++;
        q_lno[qid]--;
        if (q_lno[qid] > 0) gene_new_dept(qid);
        for (i = 0; i < SP_NUM; i++)              /* perturbed sample paths */
                if (!finish[i])                   /* if not finish */
                        if (q_l[i][qid] > 0)
                                {
                                num_ev[i]++;
                                num_d[i][qid]++;
                                q_l[i][qid]--;
                                if (num_ev[i] == EV_NUMA)
                                        {
                                        finish[i] = TRUE;
                                        complete++;
                                        }
                                }
}

void instevl(struct event *p)
{
        struct event *q;
        q = evlist;
        if (evlist == NULL)      /* list is empty */
                {
                evlist = p;
                p->next = NULL;
                }
        else
                {
                if (p->evtime < evlist->evtime)          /* head of list */
                        {
                        p->next = evlist;
                        evlist = p;
                        }
```

```
                else            /* middle or end of list */
                        {
                while ((q->next != NULL) && (q->next->evtime < p->evtime))
                                q = q->next;
                        p->next = q->next;
                        q->next = p;
                        }
                }
}

void output(index)
{
        double sbkop;

        sbkop = 0.1 * (2.0 / (power(3.0, buf[0][0] + 1) - 1.0) + 1.0 /
                (power(2.0, buf[0][2] + 1) - 1.0)) + 0.2 * (1.0 /
                (buf[0][1] + 1.0) + 0.5 / (power(1.5, buf[0][3] + 1) -
                1.0) + 0.5 / (power(1.5, buf[0][4] + 1) - 1.0) + 1.0 /
                (buf[0][5] + 1));

        printf("%d \t %lf \t %d %d \t %d %d %d %d %d %d \n", index, sbkop,
                UOPTIMAL, DOPTIMAL, buf[0][0], buf[0][1], buf[0][2],
                buf[0][3], buf[0][4], buf[0][5]);

        buf[0][UOPTIMAL] = buf[0][UOPTIMAL] + 1;
        buf[0][DOPTIMAL] = buf[0][DOPTIMAL] - 1;
}

void buf_alloc_no()
{
        int j;
        for (j = 0; j < NUM_Q; j++) bufno[j] = TOTAL_B + 1;
}

void det_buf_alloc_first()
{
        int j;

        buf[0][0] = TOTAL_B;
        for (j = 1; j < NUM_Q; j++)
                {
                buf[0][j] = 0;
                }
        for (j = 0; j < NUM_Q; j++) buf[0][j]++;
}

void rdm_buf_alloc_first()
{
        int j;
        double sum;
        double p[NUM_Q];

        buf[0][NUM_Q - 1] = TOTAL_B;
        sum = 0;
        for (j = 0; j < NUM_Q; j++)
                {
                p[j] = unif(IRN);
                sum += p[j];
                }
        for (j = 0; j < NUM_Q - 1; j++)
                {
                buf[0][j] = (int) (p[j] * TOTAL_B / sum);
                buf[0][NUM_Q - 1] -= buf[0][j];
                }
        for (j = 0; j < NUM_Q; j++) buf[0][j]++;
}
```

```
void buf_alloc_all(int num)
{
        int i, j;

        for (i = 1; i < num; i++)
                for (j = 0; j < NUM_Q; j++)
                        buf[i][j] = buf[0][j];

        for (j = 0; j < NUM_Q; j++)
                {
                buf[2*j+1][j] = buf[0][j] - 1;
                buf[2*j+2][j] = buf[0][j] + 1;
                }
}

double expo(double lambda, int i)
{
        double tem;
        tem = - 1.0 / lambda * log(unif(i));
        return(tem);
}

double unif(int i)
{
        long int a, b, c;
        double tem;
        a = ISEED[i] / 16384;
        b = ISEED[i] % 16384;
        c = (13205 * a + 74505 * b) % 16384;
        ISEED[i] = (c * 16384 + 13205 * b) % 268435456;
        tem = ISEED[i] / 268435456.0;
        return(tem);
}

double power(double base, int i)
{
        double tem;
        tem = 1;
        while (i-- > 0) tem = tem * base;
        return(tem);
}
```

*U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-47158

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.