Synthetic Workload Generation for Load-Balancing Experiments

Pankaj Mehra
Indian Institute of Technology
Benjamin Wah
University of Illinois at Urbana-Champaign

The Dynamic Workload Generator accurately records and replays network-wide resource-utilization patterns, even when competing foreground tasks are present during replay. Its pattern-doctoring rules allow it to assimilate vital process-level sheduling properties into a low-overhead, physical-level synthetic workload generator.

Dynamic Workload Generator is a facility for generating realistic and reproducible synthetic workloads for use in load-balancing experiments. For such experiments, the generated workload must not only mimic the highly dynamic resource-utilization patterns found on today's distributed systems but also behave as a real workload does when test jobs run concurrently with it. The latter requirement is important in testing alternative load-balancing strategies, a process that requires running the same job multiple times, each time at a different site but under an identical network-wide workload.

Parts of DWG are implemented inside the operating-system kernel and have complete control over the utilization levels of four key resources: CPU, memory, disk, and network. Besides accurately replaying network-wide load patterns recorded earlier, DWG gives up a fraction of its resources each time a new job arrives and reclaims these resources upon job completion. Pattern-doctoring rules implemented in DWG control the latter operation. This article presents DWG's architecture, its doctoring rules, systematic methods for adjusting and evaluating doctoring rules, and experimental results on a network of Sun workstations.

Design of load-balancing strategies

Load balancing seeks to improve the use of distributed resources through equitable distribution of workload, principally by moving work from overloaded resources to underused ones. The decision-making

1063-6552/95/$4.00 © 1995 IEEE

policies, task-migration protocols, workload parameters, and performance criteria used in making balancing moves comprise a load-balancing strategy. The *classical approach* to designing load-balancing strategies assumes that

- inter-arrival times and service requirements of tasks follow well-known distributions, such as Poisson and exponential; and
- resources meet these requirements using certain simple, well-understood service disciplines, such as first-come, first-served and round-robin.

This approach is exemplified by strategies based on queuing models. Because it is difficult to exactly model a large population of interacting user processes on a network of computers, load-balancing strategies developed this way are at best heuristic in nature.

The *systems approach* to developing load-balancing strategies uses experimental design of certain load-index functions (for selecting the site likely to complete an incoming task in minimum time) as well as experimental setting of other policy and protocol parameters. Just as the validity of the classical approach hinges on the accuracy of its models, so does the validity of the systems approach hinge on using a realistic experimental setup and accurately measuring workload and performance parameters.

The large number of possible strategies notwithstanding, the number of distinct experimental outcomes per incoming task for a given network-wide workload cannot exceed the number of sites.

We consider the load-balancing problem in the context of multi-user, multiprogrammed distributed computing environments, typified by workstations connected using local area networks. Our focus is on distributed execution of independent tasks spawned interactively and independently by the different users. In such an environment, it makes sense to restrict our attention to dynamic strategies, which schedule—that is, either execute locally or send to a remote site—each task as it arrives, independently and in a distributed fashion.

The classical approach is inapplicable here not only because of difficulties in accurately modeling multi-user, multiresource, distributed workloads but also due to the impossibility of precisely characterizing the service rates and service disciplines of CPU, memory, disk, and network resources. The systems approach is often the only option available to designers of load-balancing strategies in this environment.

Load-balancing experiments

Systematic exploration of possible strategies requires that we compare alternative strategies. Fair comparison requires that the performance parameters (such as the completion time of a task) of different strategies be measured under identical experimental conditions. Because we are only concerned with dynamic strategies, which schedule one task at a time (called the foreground or incoming task), we can treat the already scheduled tasks as well as processes running outside the control of the load-balancing system as *background workload*. We can then interpret the stipulation of identical experimental conditions as scheduling the same task under the same background workload conditions (but perhaps to a different site). Notice, however, that each incoming task has only a finite number of possible destinations. So, even though the number of strategies one might explore could be very large—and the number of strategy pairs to be compared even larger—the number of possible performance outcomes for each foreground-background pair is relatively small and equals the number of sites in the distributed system.

Experimental design of load-balancing strategies for dynamically scheduling independent tasks on a network of workstations must exploit both of these observations: that one can consider only one task at a time in the foreground; and that the number of distinct experimental outcomes is a product of the number of sites and the number of possible foreground-background pairs, irrespective of the number of strategies explored.

We first need to create a representative mix of foreground tasks and background loads. Then, for each foreground-background pair, we measure the performance of the foreground task by running it repeatedly under the same network-wide background workload, each time at a different site in the distributed system. Armed with these prerecorded measurements, we can efficiently explore the large space of strategy parameters without performing further experiments. Compared to the

approach in which experimentation proceeds online with the search for good strategies, this approach does not waste time in repeated measurement of identical foreground-background-site triples.

A word of caution: One can consider various decision points of a load-balancing protocol in isolation only when the background load fluctuates much faster than the load changes caused by the protocol itself. Otherwise, as in the case of single-user dedicated workstation clusters now in vogue with the parallel processing community, one must consider *sequences* of load-balancing decisions and the corresponding protocol-driven sequential transformations of network-wide workload. Fortunately for us, the load patterns in our environment exhibit a rate of variation (10–50 Hz) several orders of magnitude faster than the rate of arrival of tasks (0.01–0.1 Hz). Because load-balancing decisions are made only at the time of arrival of incoming tasks, the rate of decision-making is approximately the same as the task arrival rate. The high rate of background-load fluctuation virtually eliminates all correlation between the network-wide load patterns at successive decision points, so that one can equally well pick an uncorrelated (independently sampled) network-wide load pattern for experiments involving the next decision point. This observation also allows us to work with a statically chosen sample of background workload patterns and to ignore any relationship between background workloads and decision points.

Efficient exploration of the space of strategy parameters in dynamic load balancing, therefore, demands from the underlying experimentation environment functionality for accurate measurement and accurate and repeatable replay of multiresource, highly dynamic, network-wide workloads. Recording and replaying of system workloads is called the *workload-generation problem.*[1]

Workload-generation problem

Foreground tasks and background workloads affect each other by competing for resources. For real workloads, the operating system's resource scheduler resolves such competition by apportioning the available resource capacity among competing processes. While a background load pattern is being recorded, the process population generating that load has complete control over a site's resources. A foreground task introduced on top of such a workload would take some of the resources away from the background process population, thereby altering its resource-utilization pattern. Therefore, the impact of foreground tasks introduced on top of generated workloads needs careful consideration. Such *foreground-background interaction* is an important characteristic of our workload-generation problem.

It is impractical to maintain a process-level view of the entire background-process population while generating workload at the physical level.

Since the performance of a foreground task under a given background load depends solely on the latter's resource-utilization pattern, it is natural to represent workloads by their utilization patterns. This is called the *physical-level view* of a workload.[1] Modeling the interaction between a foreground task and the generated workload, however, favors adopting a *process-level view*. On the other hand, asynchronous events such as keyboard interrupts and context switches are difficult to model at the process level because of

- ignorance regarding the separate resource-access patterns of each process;
- the complexity of modeling the interaction between processes and interrupt-handling routines of the operating system; and
- inadequate clock resolution to either precisely time the occurrences of interrupts and context switches, or to replay them at the same fine grain.

Overcoming these difficulties will necessitate the use of costly hardware instrumentation for gathering of information and high-resolution timers for driving the generator.

It follows that it is impractical to maintain a process-level view of the entire process population while generating workload at the physical level. Consequently, we consider the representation and regeneration of workloads at physical level only: the aggregate effect of interacting processes is measured in terms of their total resource utilization; the workload generator uses artificial programs (or *synthetic workload*) to regenerate the recorded resource-utilization patterns.

As we will demonstrate, regenerating resource-utilization patterns in the absence of foreground tasks is quite easy. However, additional control must be exercised if, at the physical level, one wants the interaction between a foreground task and a generated background

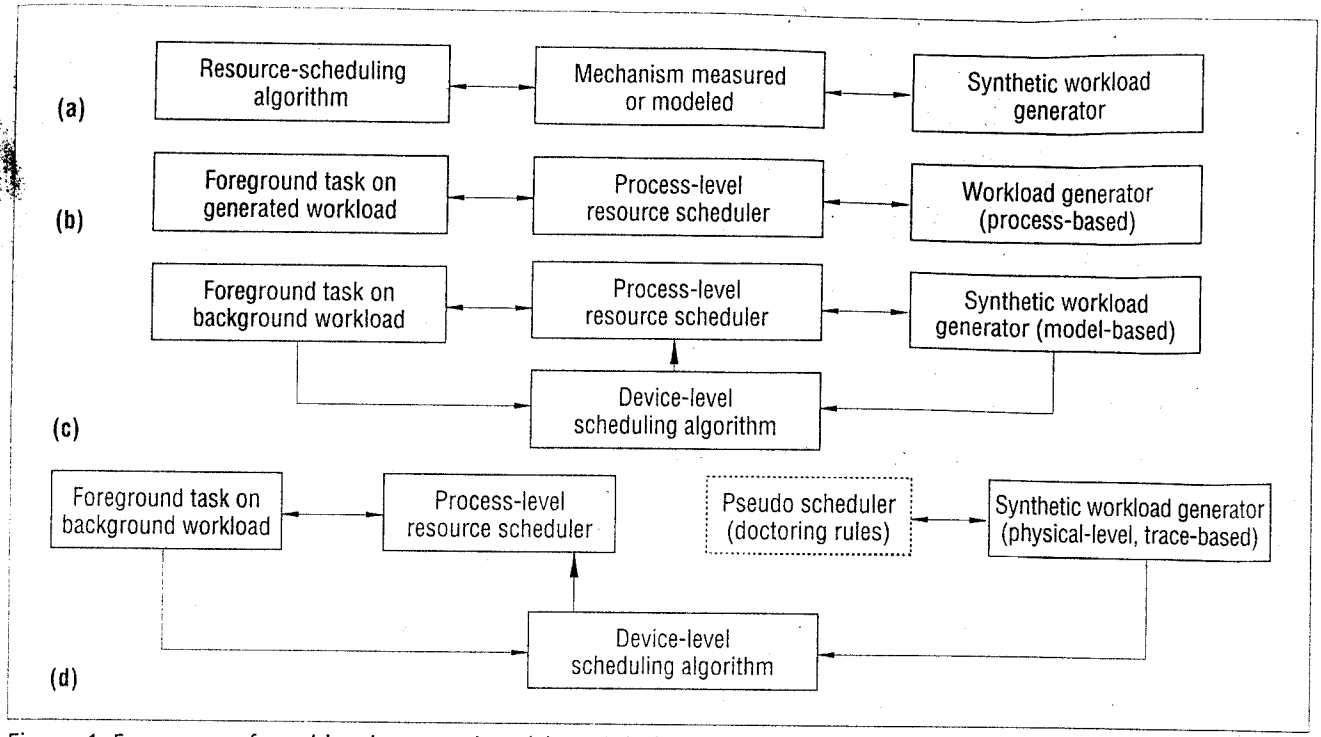


Figure 1. Four ways of workload generation: (a) straightforward record-and-replay scheme,(b) workload generation using real workloads, (c) workload generation using synthetic processes, and (d) workload generation with feedback using dynamic doctoring.

workload to resemble the interaction between that task and the real process population that generated the recorded background load pattern. Physical-level synthetic workload generators for load balancing must embody mechanisms for dynamically adjusting the amount of generated work whenever foreground tasks are run concurrently with the synthetic workload.

Approaches to workload generation

Workload generators that faithfully reproduce recorded loads (Figure 1a) do not offer any mechanism for dynamic adjustment of load: the generated workload does not depend upon the mechanism being measured or modeled. Most existing synthetic workload generators follow this paradigm.[2] They are quite adequate for certain performance evaluation problems in computer systems, such as in evaluating alternative implementations of file systems, where it can be assumed that the file-access patterns of a user are implementation-independent.[3,4] Another example is in the evaluation and refinement of virtual-memory algorithms using traces of memory references. Once again, memory-access patterns of programs can be assumed to be independent of (say) the particular cache-coherence protocol being evaluated. Such generators are, however, unsuitable for load-balancing experiments because it is impossible to model foreground-background interactions when new tasks are introduced on top of generated loads.

Experimenters in load balancing have tended to use streams of real user jobs to create workload: the earlier jobs generate background workload for later jobs.[5–7] Since this approach generates and represents workloads at the process level, it is incapable of modeling asynchronous events, such as handling of keyboard and mouse interrupts, on account of its large granularity. Because the resource scheduler (Figure 1b) handles foreground-background interactions, no adjustment in generated load is necessary. The key problem with this approach is that real workloads in a multiprogrammed environment cannot be precisely represented using only user-level programs and commands.

One way to combine the expressiveness of physical-level workloads with the correct scheduling behavior of process-level workloads is to model the physical-level behavior of all the processes, including system processes. The workload generator then generates synthetic workload using the model. This approach amounts to recreating an entire process population (Figure 1c). As already discussed, the volume and rate of data necessary for measuring, much less generating, such a workload precludes feasible implementation in software alone. Even if hardware support is available, it is nontrivial to model a large process population with complex interactions.

Our approach (shown in Figure 1d) is to represent and regenerate workloads only at the physical level. To simulate foreground-background interactions, we consider a simplified model of the resource scheduler, and collect information during measurement and generation to drive that model. The simplified model requires that process

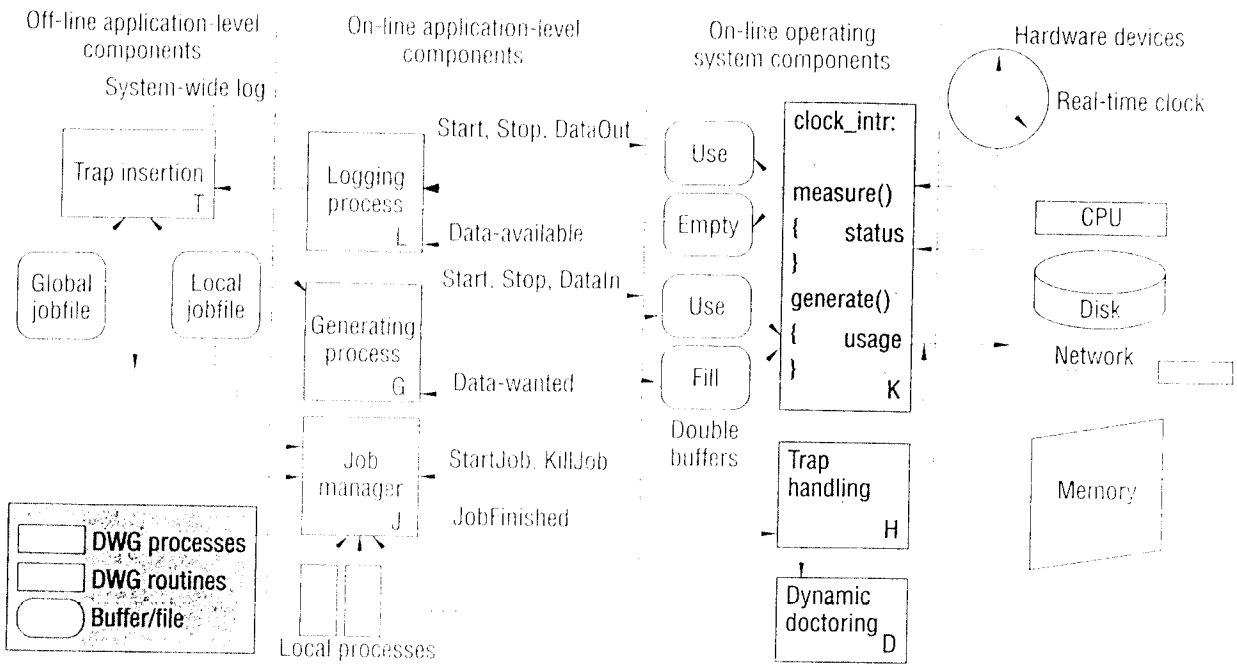


Figure 2. DWG architecture.

counts be recorded along with resource-usage patterns, and that the generator keep track of the number of tasks introduced on top of the replayed load. Using these counts, the model provides a way to dynamically alter generated loads in response to initiation or termination of test jobs; formulae for computing load reductions are encoded in *doctoring rules*, which make the workload generator behave as though it were under the control of the resource scheduler even when it is not.

DWG architecture

DWG (Figure 2) is a software architecture that implements our workload-generation approach (Figure 1d) and illustrates the principles of design using our approach. In Figure 2, DWG's components are either processes (shown as unshaded rectangles), callable kernel routines (shaded rectangles), or buffers and files (boxes with rounded corners); they include mechanisms for

- measurement and generation (box labeled K),
- data transfer in and out of the kernel (labeled L and G),
- handling of asynchronous events using traps (labeled H and J), and
- dynamic doctoring of generated load levels (labeled D).

These mechanisms are organized into layers of software. The lowest of these layers (shown second from right in the figure) comprises functions implemented inside the operating-system kernel. The next higher layer comprises user-level processes that control kernel-level functions. The topmost layer (shown leftmost in the figure) comprises off-line mechanisms for controlling the initiation and termination of foreground jobs.

Functions implemented in the operating-system kernel

The core of DWG consists of its measurement and generation routines (boxes labeled K in Figure 2); these measure and control the utilization levels of locally accessible resources (CPU, memory, disk, and network). At every real-time clock interrupt, the measurement routines estimate the current utilization as follows. If the CPU was busy when the clock interrupted, it is assumed to have been busy for c percent of the time since the last interrupt, where c is a tunable parameter of DWG. For the memory, the routines record the number of free pages. For the disk, they record the number of blocks transferred since the previous interrupt and, for the network, the number of packets received or transmitted since the previous interrupt. Also recorded is the number of running processes contributing to the measured load at any given time.

At every real-time clock interrupt, the generation routines determine the amount of work to generate for each resource. For the CPU, this amount is expressed as a fraction of the interval between successive interrupts; for memory, it is the number of pages to be occupied until the next interrupt; for disk and network resources, it is the number of disk transfers and the number of network packets.

When there are foreground processes, the computation of generated load employs *dynamic doctoring rules* (box labeled D in Figure 2) to compensate for foreground-background interactions. The amount of reduction in

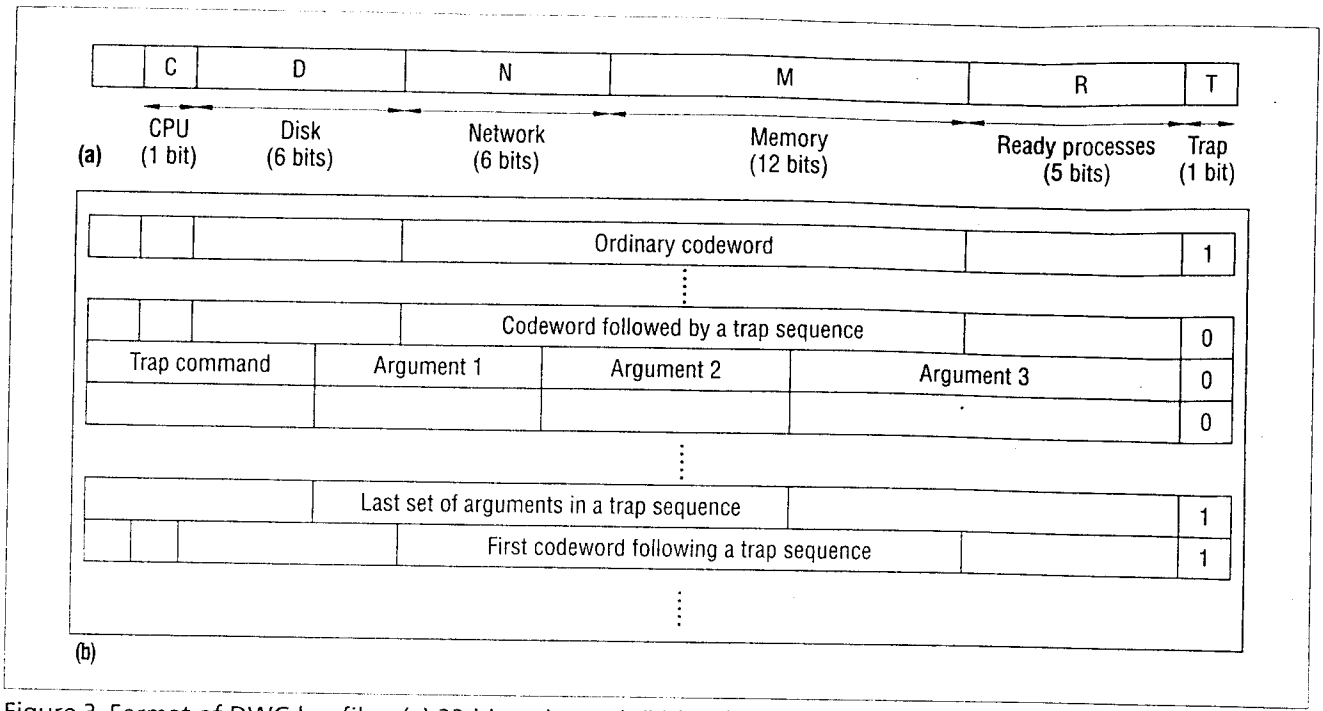


Figure 3. Format of DWG log files: (a) 32-bit codeword; (b) log file arranged as a sequence of codewords and arguments of trap calls.

generated load caused by these rules depends upon the relative sizes of foreground and background process populations. The trap-handling routines of DWG keep track of the foreground population size. (The size of the foreground population equals the number of test jobs introduced on top of the generated load.) The size of the background process population is already recorded in the log being replayed.

While CPU and memory loads are generated inside the kernel, requests for generation of disk and network traffic pass on to external processes. CPU load is generated by repeatedly executing a segment of pure computational code. Memory load is generated by taking pages out of the kernel's pool of free virtual-memory pages, thereby making them unavailable to user processes. The external process responsible for generating disk and network traffic does so by, respectively, performing unbuffered output to a file and broadcasting synthetic packets over the local-area network. CPU and memory loads are generated at each site; disk traffic, only at diskful sites; and network traffic, only at one selected site on each subnet.

The measurement and generation routines switch buffers upon reaching the end of the current buffer. (See Figure 2.) When switching buffers, they signal their respective external processes (the logger process, labeled L in the figure, and the generator process, labeled G), to initiate data transfer. While the kernel is busy with the other buffers, the external process either loads or unloads the idle buffers. Buffer sizes are large enough that data transfers happen only a few times a minute, and yet small enough that the memory overhead of synthetic workload generation is at worst two or three pages. (This is small overhead compared to the hundreds of pages occupied by typical operating-system kernels.)

On-line application-level components

DWG requires three processes at each site:

- *the logging process* (labeled L in Figure 2), which transfers measurements out of the kernel into the log file;
- *the generating process* (labeled G), which transfers data from recorded load patterns into the kernel; and
- *the job manager* (labeled J), which initiates and terminates test jobs upon receiving signals from the trap-handling routines (labeled H) of the kernel, as well as measures the completion time of test jobs.

The interface between these processes and the kernel-based functions is via a system call.

In addition to these functions, the kernel signals the generating process when there is some disk or network traffic to be generated. The process determines how much traffic to generate and does the necessary input/output. The logging and generating processes are also responsible for starting and stopping measurement and generation, respectively, inside the kernel.

In DWG, it is possible to synchronize measurement and generation so that measurement begins and ends exactly when generation does. This capability allows us to compare actual and generated loads, as explained later, and allows us to translate the time of occurrence of an experimental event into the offsets of the corresponding codeword in a log file.

Figure 3b shows the typical format of a DWG load

pattern, including the formats for encoding resource-utilization information in codewords (Figure 3a), and for managing asynchronous events using traps. (*Traps* are data bytes in a special format.) Upon hitting a trap, the trap-handling functions of DWG (labeled H in Figure 2) queue up the trapped requests, and signal the local job-manager process (labeled J in Figure 2) to interpret and service the requests. If a trap command affects the size of the foreground process population, these routines also update the process-population counts appropriately.

Off-line application-level components

These components come into play after a system-wide log has been measured but before it can be replayed. An initial log contains no traps. The trap-insertion routines (labeled T in Figure 2) insert traps at suitable offsets into each log file; each log is instrumented so that test processes can start or stop at precise moments relative to the start of the experiment.

Traps inserted into a recorded load pattern allow the pattern-doctoring rules to keep a count of foreground processes.

Traps allow the dynamic doctoring routines (labeled D in Figure 2) to maintain a count of processes that exist during generation but did not exist at the time of measurement. The presence of such processes warrants a reduction in generated load; the amount of reduction depends upon the number of foreground processes. Because that number changes only when either a new process starts or an old process finishes, traps can trigger an update of process counts inside the generator precisely when the size of the competing process population changes. Thus, before measured data go back to the kernel for replay, the trap-insertion routines insert traps at the starting and (if known) stopping points of jobs.

In determining where to insert traps in prerecorded network-wide logs, the routines use global *jobfiles*. These contain information about

- the site at which a job will be executed;
- starting time of the job;
- stopping time of the job, if known (otherwise, upon noticing job termination, the job manager makes a system call that has exactly the same effect as a job-stop trap); and
- the command and arguments needed for starting the job.

The trap-insertion routines associate a unique global identifier with each job, and partition the global jobfile into local jobfiles, which the generating process passes on to the individual job managers at each site.

DWG operation

The overall operation of DWG involves three phases: measurement, trap insertion, and generation. In the first phase, utilization levels of four key resources—CPU, memory, disk, and network—are recorded at each clock interrupt. (In our implementation on a network of Sun 3 workstations, there are 50 interrupts per second.) In the second phase, which is performed off-line, DWG makes provisions for running test jobs on top of the recorded load. It does so by inserting traps at appropriate points in the recorded data. During the generation phase, at each clock interrupt, DWG dynamically determines the amount of load to generate for each resource. It does so either by reading the instrumented log or by assessing the work left pending from previous interrupts. It then generates the requisite load by issuing synthetic resource-usage instructions. If DWG encounters a trap while reading the instrumented log, it decodes the trap and carries out the corresponding functions, such as updating process-population counts and signaling the local job manager to perform appropriate job-control functions. When a test job started by the job manager finishes, the job manager records the completion time. Thus, DWG replays background loads, introduces test jobs at precise instants, and measures their completion time under controlled loading conditions. A detailed description of each phase of DWG's operation follows.

Workload measurement

The `measure()` routine of the kernel is periodically invoked by a real-time clock interrupt. It samples the system's state and records

- whether or not the CPU is busy,
- the number of free memory pages,
- the number of disk transfers since the previous interrupt, and
- the number of packets active on the network since the previous interrupt.

Also recorded with each data item is the number of local and global processes generating the current load. Because clock interrupts occur several tens of times per second, the measured data grow at a phenomenal rate. We keep such growth in check through efficient coding of information and periodic removal of data from the kernel by an external process. Similarly, during generation, information needs to be transferred into the kernel at the rate of a few hundred bytes per second. To minimize the number of data transfers, buffer pairs are allocated inside the kernel. Data transfer can proceed using the idle buffer while the kernel is busy reading and writing the other buffer. Buffer sizes are large enough that there are at most only a few transfers per minute.

Trap insertion

The trap-insertion routines perform essentially an event-driven simulation, using the offset into the logged data as simulated time. Recall that in load-balancing experiments, experiments with the same foreground-background pair need to be repeated several times, each time using a different site for executing the foreground job. A trap at a specified time, therefore, indicates the time when a test job will be initiated during generation.

To insert traps, the trap-insertion routines sort the events in jobfiles, first, by their starting times, and, second, by their stopping times (if known). The intuition is that stop events (if any) for the same job must necessarily follow the corresponding start events. The routines maintain two event-lists, one headed by the next job to start, and the other by the next job to stop (if known). At every simulated instant, lists of *fired* events are computed. Every event results in at least one trap at the associated job's site of execution, and possibly others at sites generating disk and network traffic. This phase ends with the creation of *instrumented* logs, one per site; the traps inserted into these logs contain instructions for the `generated()` kernel, which executes those instructions upon hitting these traps during the generation phase.

Generation

The processes generating the real workload are subject to scheduling. In Unix and related operating systems,[8] the scheduler maintains queues of ready-to-run processes, each queue corresponding to one priority level. It allocates resources to these processes in a round-robin fashion within each queue, and in order of priority among the queues. Higher priority resources can preempt low-priority ones. Priorities are recomputed periodically, thus causing processes to move between queues.

The generation routines are implemented inside the kernel; unlike user-level processes, they are not subject to a Unix scheduling algorithm and essentially behave like a high-priority real process. If the generator were to always replay the recorded load exactly, test jobs introduced into the system would encounter greater delays under a generated load than under the corresponding real workload. Therefore, recorded workload pattern must be dynamically adjusted, taking into account the number of active foreground tasks.

When foreground processes are present, DWG's doctoring rules smear its load pattern in a principled way, effecting a net reduction in generated load while conserving the total amount of background work.

Ideally, the generator would need to implement the entire queuing discipline to emulate the true behavior of the recorded load in the presence of an additional process. If the starting and stopping times of all jobs in the replayed workload were known ahead of time, such emulation could possibly be done off-line in a fashion similar to the trap-insertion process. However, stopping times of jobs in the replayed workload are usually unknown, and are, in fact, load-dependent. Therefore, an ideal generator would need to implement the process scheduler's queuing discipline *on-line*! That would be prohibitively expensive computationally. As a compromise, DWG makes certain simplifying assumptions about the scheduler's queuing discipline; these assumptions allow it to compute the altered behaviors dynamically without incurring too much computational overhead. This component of our generator contains *doctoring rules* for altering the generated load in the presence of competing test jobs. These rules, and the assumptions on which they are based, are as follows.

Rule 1: Reduce generated load in the presence of foreground processes.

We assume, first, that all the processes (background as well as foreground) have the same priority and that resources are allocated to processes in round-robin fash-

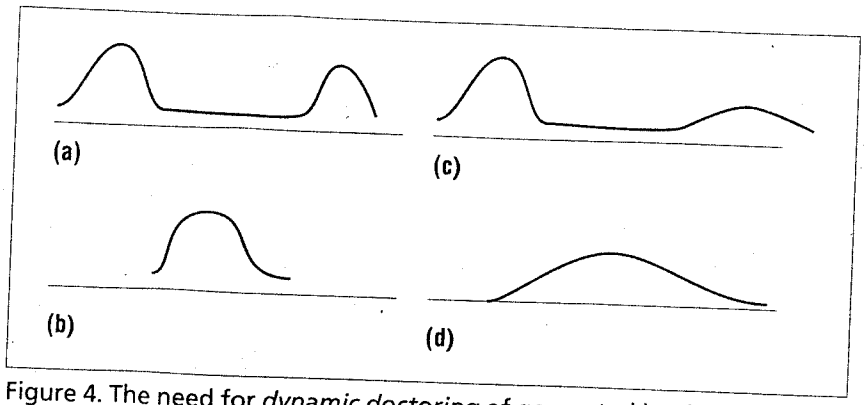


Figure 4. The need for *dynamic doctoring* of generated load: (a) generated load without test process, (b) test process under no load, (c) generated load with test process, and (d) test process under the generated load. The load patterns in (c) and (d) are smeared by foreground-background interaction, with both the background and the foreground loads taking longer to complete.

ion; and, second, that the background processes' utilization of different resources is reduced by the same proportion in the face of competition from foreground processes. For example, a 15% reduction in CPU use warrants a 15% reduction in memory occupancy, and so on. Also, we model CPU and memory as *constrained resources* whose usage levels are bounded by, respectively, 100% busy and 100% occupied. In practice, all resources have physical limits on utilization levels. The logical limits on disk and network appear to be infinite because requests for their use can be buffered in memory space; such buffers do not exist for the CPU and memory resources. Therefore, CPU and memory usage need to be explicitly spread out (see Figure 4) over time by buffering unfulfilled requests as *pending work* in the generator.

We assume that the load levels on private resources (CPU and memory) are affected only by the local process population, whereas those on shared resources (disk and network), by the network-wide process population. (For shared-memory systems, memory would also be treated like a shared resource.) The treatment of disk as a shared resource is specific to the client-server model of distributed file systems; in other models, disk may be treated as a private resource. Under these assumptions, the reduction in generated load computes as a ratio between the process-population sizes at the times of measurement and generation. Let b be the number of background processes, as recorded in the log being replayed; and f, the number of foreground processes, as maintained by the trap-handling routines of DWG. Then, the percentage of resource needs satisfiable at each clock interrupt is at most

$$p_i = (100 \cdot b)/(b + f). \quad (1)$$

Further, the visible capacities of constrained resources (CPU and memory) are reduced to p_i percent of their maximum values.

Rule 2: Conserve total work generated.

Plain reduction in load levels is insufficient for reproducing the true behavior of a background-process population. The processes constituting that population, when deprived of the full use of resources, would have taken longer to finish. Therefore, whenever the generator fails to achieve the recorded load levels, either due to competing processes or due to reduced resource capacities, it should carry over the difference between recorded and generated loads as pending work for subsequent cycles.

When a foreground process remains active for a few consecutive cycles of generation, the pressure of pending work may overwhelm the generator to such an extent that, instead of getting new work from the log, it will be forced to spend one or more cycles just to finish the pending work. This happens when the pending loads on constrained resources (CPU and memory) exceed the corresponding resource capacities. DWG is said to be *on hold* when in this mode. Holding allows us to slow down the replay of a recorded log; the first rule governs the rate of this slowdown. When on hold, the generator determines the maximally constrained resource. It then computes, with respect to that resource, the fraction of pending work that the current generation interval can accommodate. The same fraction of pending loads on other resources is then targeted for generation in the current cycle.

During generation, pending work receives priority. For constrained resources, the rest of the (possibly reduced) capacity is allocated to (possibly reduced) background workload from the current interval. The combination of new and pending workloads for a resource may exceed its visible capacity; when that happens, the overflow is simply added to the pending work for future cycles.

So long as only one job is introduced on top of a generated workload, the doctoring rules and the trap-insertion algorithms do not interfere with each other. This is indeed the case in our experiments. If more than one job were to be introduced, doctoring rules triggered after the initiation of the first job could delay the start of the next job if the generator goes on hold, as explained earlier. Such a possibility clearly does not arise in the case of only one foreground task.

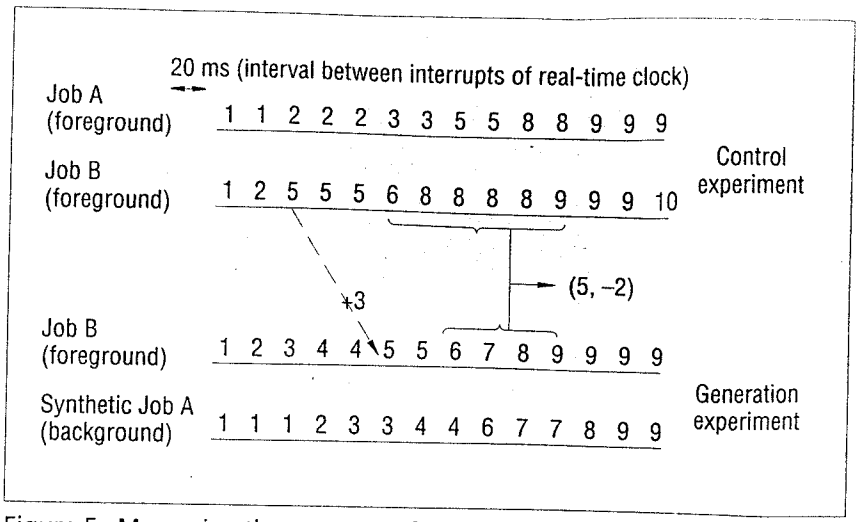


Figure 5 . Measuring the accuracy of generated loads.

Evaluation, parameterization, and tuning of generation mechanisms

The generation mechanisms described just now allow us to record and replay workloads generated by a population of test jobs, and replace the background process populations used in traditional experiments with synthetic workloads. However, we still need to assess how well the generated patterns approximate those from real workloads. To achieve high-quality generation, we need to first parameterize the generation mechanisms, then experiment with many different parameter sets on a prototype system, and finally select the best one for our load-balancing experiments.[9] We performed our experiments on a configurationally heterogeneous system consisting of a diskless Sun 3/50 with 4 Mbytes of RAM, a diskful Sun 3/50 with 4 Mbytes, a diskful Sun 3/260 with 8 Mbytes, and a diskless Sun 3/60 with 24 Mbytes. Connecting the four workstations was a single 10 Mbps Ethernet.

Evaluation of generated loads

To evaluate DWG, we designed an experiment to compare generated workloads against measured workloads of real processes. In this experiment, we used certain test jobs that had been instrumented to produce checkpoints when they reached certain preset points in their program codes. We expanded the measurement routines and the system-call interface of DWG to include mechanisms for recording the most recent checkpoint of every active job, be it in the foreground or the background.

Figure 5 shows the design of our experiment. Each experiment involves a pair, (A,B), of jobs and proceeds as follows. First, both A and B execute in the foreground on top of an idle background load; at each clock interrupt, the most recent checkpoint of each job is recorded in the resulting log. This is our *control experiment*. Next, only job A executes on top of an idle load, and the resulting load pattern (including checkpoint timings) is recorded by DWG. In the final step, the recorded log of job A is replayed as a synthetic load using DWG, while job B runs in the foreground. Once again, DWG's measurement routines record the resulting log (now including the checkpoint times of both the foreground and the background jobs). This is our *generation experiment*. As illustrated in Figure 5, we can assess the quality of generation using the errors between offsets of corresponding checkpoints in the logs of control and generation experiments.

In Figure 5, suppose that we perform a control experiment in which two jobs, A and B, run in the foreground, and that the checkpoints of the two jobs occur as shown on the top two time lines. Each of the uniformly spaced ticks on a time line shows the *most recent checkpoint* of the corresponding job. Next, suppose that we perform another experiment with the load pattern for job A generated synthetically in the background and with job B actually run in the foreground, and that the checkpoints of the two jobs occur as shown on the bottom two time lines. We can compute errors for individual checkpoints: for example, for checkpoint 5 of job B, the signed error is +3 because it occurs 3 ticks too late with respect to the control experiment. Likewise, errors can be computed for segments of each job: for example, for job B, the segment that begins at checkpoint 6 and ends at checkpoint 9 takes 5 clock intervals to complete in the control experiment, but only 3 clock intervals in the generation experiment.

Suppose that checkpoints k_i and k_j of a job occur at times t_i and t_j in the control experiment. Further suppose that the same checkpoints occur at times T_i and T_j in the generation experiment. Then, the error e_i of the ith checkpoint is

$$e_i = T_i - t_i , \qquad (2)$$

and the error for the job segment contained between the ith and jth checkpoints is

$$e_{ij} = (T_j - T_i) - (t_j - t_i). \qquad (3)$$

Both signed and absolute values of errors were considered.

In our experiments, we used seven different benchmark jobs as described in Table 1. They include three

Table 1. Benchmark programs used in evaluation and tuning of DWG.

Name	Description
Sort1	Sorting a small file by multiple fields with unlimited memory
Sort2	Sorting a large file by a single field with unlimited memory
Sort3	Sorting a small file by a single field with limited memory
UC1	Uncompressing a compressed file (#1)
UC2	Uncompressing a compressed file (#2)
W.TF	The Perfect Club benchmark FLO52Q—solving Euler equations
W.TI	The Perfect Club benchmark TRFD—two-electron integral transformation

Table 2. Parameter sets for doctoring rules.

	Set number								
	1	2	3	4	5	6	7	8	9
x	100	95	110	100	95	110	110	100	95
c	80	80	90	90	90	80	70	70	70

jobs of Unix sort utility with different file sizes and memory requirements, two jobs of the Unix uncompress program, and two of the Perfect Club benchmarks.[10] More detailed evaluation using other benchmarks (including other Perfect Club benchmarks) are available.[9] Governing the choice of benchmarks was our desire to achieve a job mix with sufficient variety so that the load index would need to take into account resources other than the CPU. For instance, sorting with large amount of memory and compressing large files achieve, respectively, pure memory-intensive and disk-intensive (network-intensive on clients) behavior. The Perfect Club benchmarks exhibit compute-intensive behavior. Because each experiment involves two jobs, we made sure that the selected combinations represented a variety of foreground-background combinations.

We allotted 15 minutes for each experiment, and chose the data for the benchmark programs such that idle completion times would be close to five minutes on the slowest site. We instrumented benchmarks to produce checkpoints at certain fixed points in their execution. Each program produced approximately 200 checkpoints at regular intervals. We placed checkpoints at the beginning of each major loop. Depending on loop size, every kth iteration of the loop caused a checkpoint, where k was chosen so that k iterations of the loop would take approximately 1.5 seconds on the slowest site.

We added additional functionality for timing of checkpoints to the kernel-based components of DWG; at each checkpoint, the benchmark programs incremented a checkpoint identifier and notified DWG of the checkpoint's occurrence by making a system call. We also assigned each benchmark program a unique job identifier so that its checkpoints could be distinguished from those of another (concurrently active) program.

Parameterization of generation mechanisms

Two components significantly affected DWG's behavior: the formula for computing p_1 in the first doctoring rule (Eq. 1); and c, the percentage of clock interval consumed by the CPU generator when the recorded log shows that the CPU was busy in that interval. We can parameterize the doctoring rule as:

$$p_1 = (100 \cdot b)/(b + f \cdot x), \tag{4}$$

where parameter x controls the relative importance of processes generating the foreground load with respect to the processes that generated the recorded load which is now being replayed in the background.

Table 2 shows nine different parameter sets for DWG. In this table, both x and c are expressed as percentages. For example, $x = 100$ means that foreground and background processes are given equal weight; and $x = 110$, that foreground processes are given 10% more weight than background processes. The latter may be appropriate when the count of background processes includes some inactive (operating system) processes. Continuing the example, $c = 70$ means that CPU-load generation can use up to 70% of the clock interval.

Other parameter sets are possible. However, because we do not have automated mechanisms for parameter selection at the current time, we have chosen to limit our attention to the parameter sets described in Table 2. Our goal is to first evaluate each of these parameter sets using the experiments described in the previous section, and then select the best one among them for subsequent load-balancing experiments.[9]

Results of comparing different parameter sets

Signed and unsigned errors were calculated for each of the parameter sets. We performed 17 experiments per parameter set, each using a different combination of foreground and background jobs. For each experiment, and for each job, we computed the following statistics (also see Eq. 2):

$$E_s = \sum_{i=1}^{k} (e_i - e_{i-1})/k, \qquad E_u = \sum_{i=1}^{k} |e_i - e_{i-1}|/k \tag{5}$$

where k is the total number of checkpoints of the foreground job. E_s is the average signed delay incurred by a checkpoint in the generation experiment, relative to the control experiment. Positive values of E_s indicate that, on the average, checkpoints in generation experiment

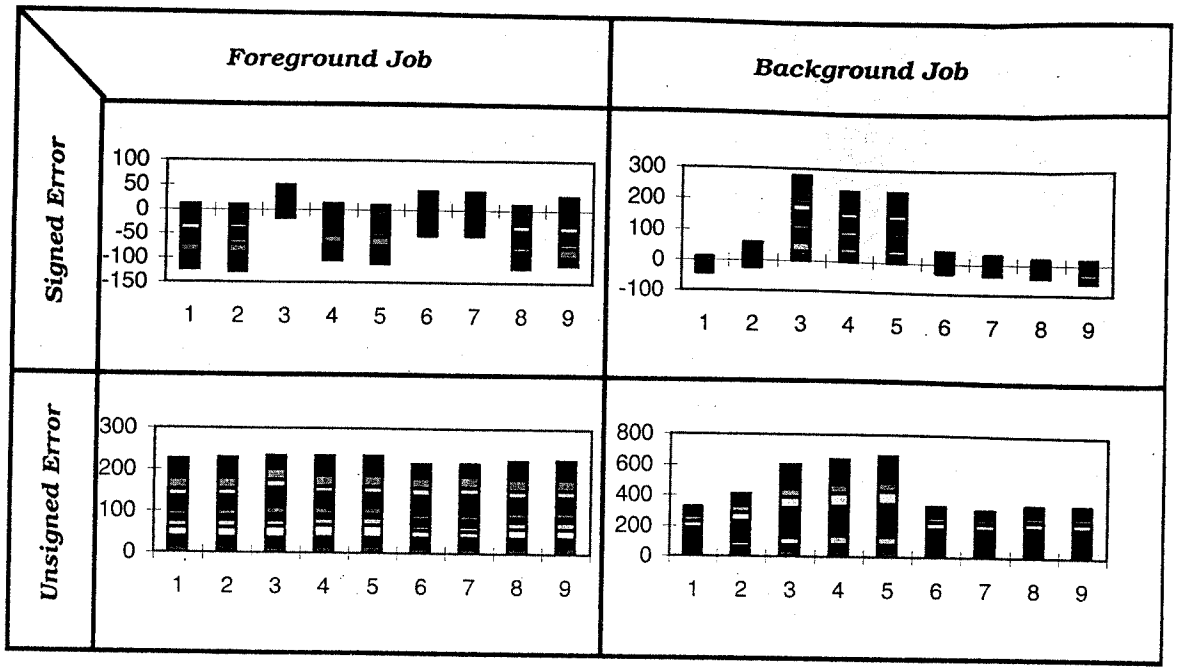


Figure 6. Errors due to generation for the nine parameter sets.

are delayed with respect to their true timings from the control experiment, and negative values, that checkpoints occur too soon in the generation experiment. Because positive and negative errors can sometimes cancel out, it is necessary to consider the absolute delays as well; E_u is the average (unsigned) error between the times at which the same checkpoint occurs in the generation and control experiments. We refer to E_s and E_u as the signed and the unsigned errors of generation.

We computed statistics E_s and E_u for both jobs of each evaluation experiment. Figure 6 shows the total signed and unsigned errors for 17 evaluation experiments using the benchmarks mentioned above. In each of the graphs each column presents the errors for one parameter set from Table 2. Two values obtained from each experiment: the error for the background job, and the corresponding error for the foreground job. Errors are in units of clock intervals of 1/50 seconds each. Parameter sets 3, 4, and 5 have unacceptable performance especially on errors for the background job (right half of Figure 6). Parameter sets 6 and 7 appear to have no unacceptably large errors, but there is no clear winner.

Due to the way that benchmarks are instrumented, each pair of checkpoints describes a program segment. With preemptive load-balancing policies, we can schedule complete jobs as well as program segments. The size of the scheduled segments depends upon the preemption interval of the policy: the longer the preemption interval, the longer is the maximum size of a scheduled program segment. Because the generator should incur larger errors for longer jobs, we wish to study the relationship between errors and preemption intervals. To this end, let us consider *all pairs of checkpoints* in the foreground job of each experiment, and for the program segment between the ith and jth checkpoints, let us create a data-pair $<l_{i,j}, e_{i,j}>$, where $l_{i,j} = (t_j - t_i)$ represents the true running time of that segment, and $e_{i,j}$ represents the net delay in the execution of that segment due to generation (Eq. 3).

Considering that each job produces close to 200 checkpoints during its execution, the information about all pairs of checkpoints is too voluminous to be meaningfully visualized; therefore, we group different data pairs into *frequency classes*. Let us consider a resolution of 12 seconds along the l-axis, and 4 seconds along the e-axis. For notational convenience, let us refer to each frequency class as $<E, L>$, where both E and L denote ranges: E represents a 4-second range along the e-axis, and L, a 12-second range along the l-axis. Let $f_E{}^L$ denote the number of data points whose l and e values lie within these ranges—that is,

$$f_E{}^L = | \{<l,e> \textit{ such that } l \in L \textit{ and } e \in E\} |. \quad (6)$$

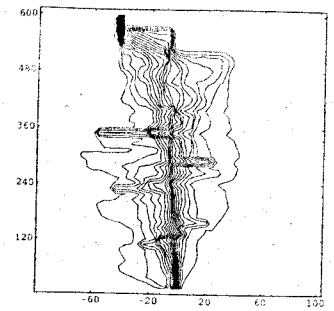
Further, let $R(E)$ denote a super-range formed by including all the ranges to the left of (and including) E on the e-axis. Thus, we can compute the cumulative probability of error:

$$P_E{}^L = f_R{}^L(E)/\Sigma_E f_{R(E)} \quad (7)$$

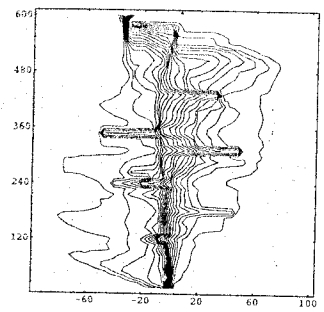
$P_E{}^L$ represents the probability that jobs whose length lies inside the range L will incur an error that lies either inside or to the left of the range E.

Figure 7 shows contour plots of $P_E{}^L$ in the e-l space.

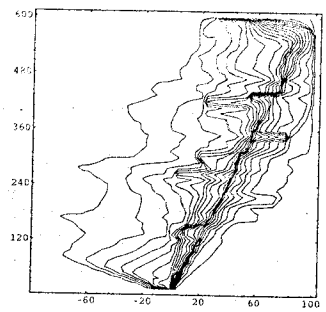
Figure 7. Contour plots of cumulative probability of error with nine different parameter sets. Each contour connects points with equal cumulative probability of error. The y-axis represents possible lengths (in seconds) of the preemption interval; and the x-axis, errors due to generation (also in seconds). There are 20 regions of equal probability: the leftmost region corresponds to cumulative probabilities of error between 0 and 0.05; the one to its right, between 0.05 and 0.1; and the rightmost one, between 0.95 and 1.



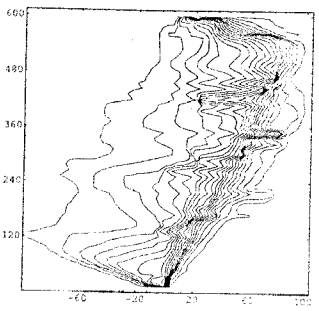
Parameter Set 1



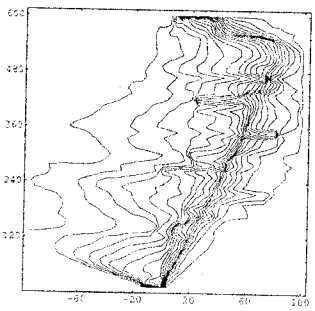
Parameter Set 2



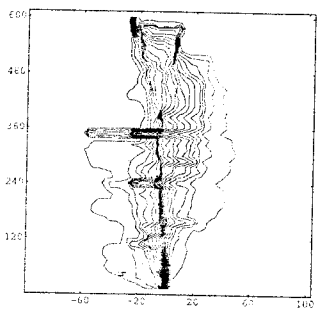
Parameter Set 3



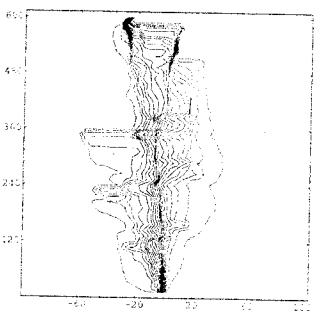
Parameter Set 4



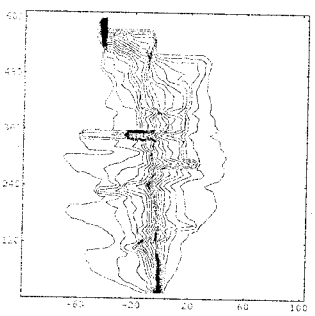
Parameter Set 5



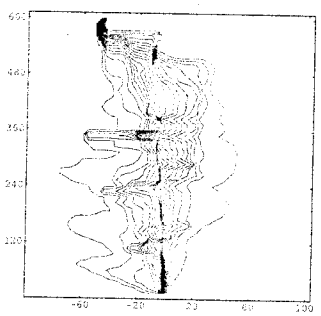
Parameter Set 6



Parameter Set 7



Parameter Set 8



Parameter Set 9

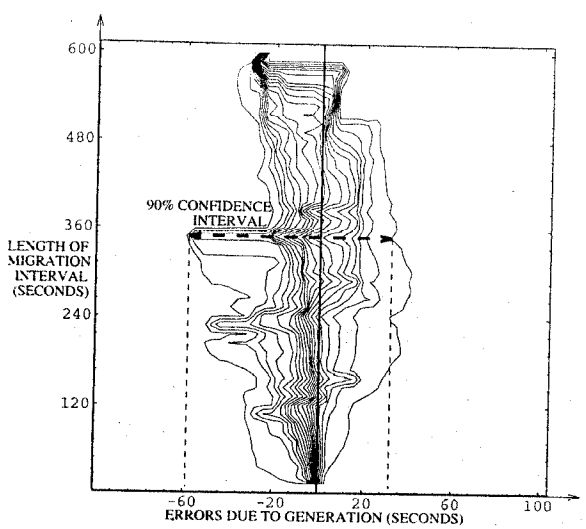


Figure 8. Contour plot of cumulative probability of error for the selected parameter set 7.

Each contour connects points with equal cumulative probability of error. Because the line of zero error is a vertical line in the middle of the plot, we can see why parameter sets 3, 4, and 5 are so poor: almost all the errors are large for any value of L. This figure also shows that most of the contours for parameter set 7 lie close to the line of zero error; this parameter set is used in our doctoring rules.

Figure 8 shows a detailed view of the contour plot for parameter set 7. The y-axis represents the length of preemption interval in seconds; the x-axis represents delays due to generation (also in seconds). Parameter set 7 was selected from the nine sets shown in Figure 7 because its contours are clustered tightly around the line of zero error. The 19 contours divide the space into 20 regions of equal probability; thus, if we omit the leftmost and rightmost

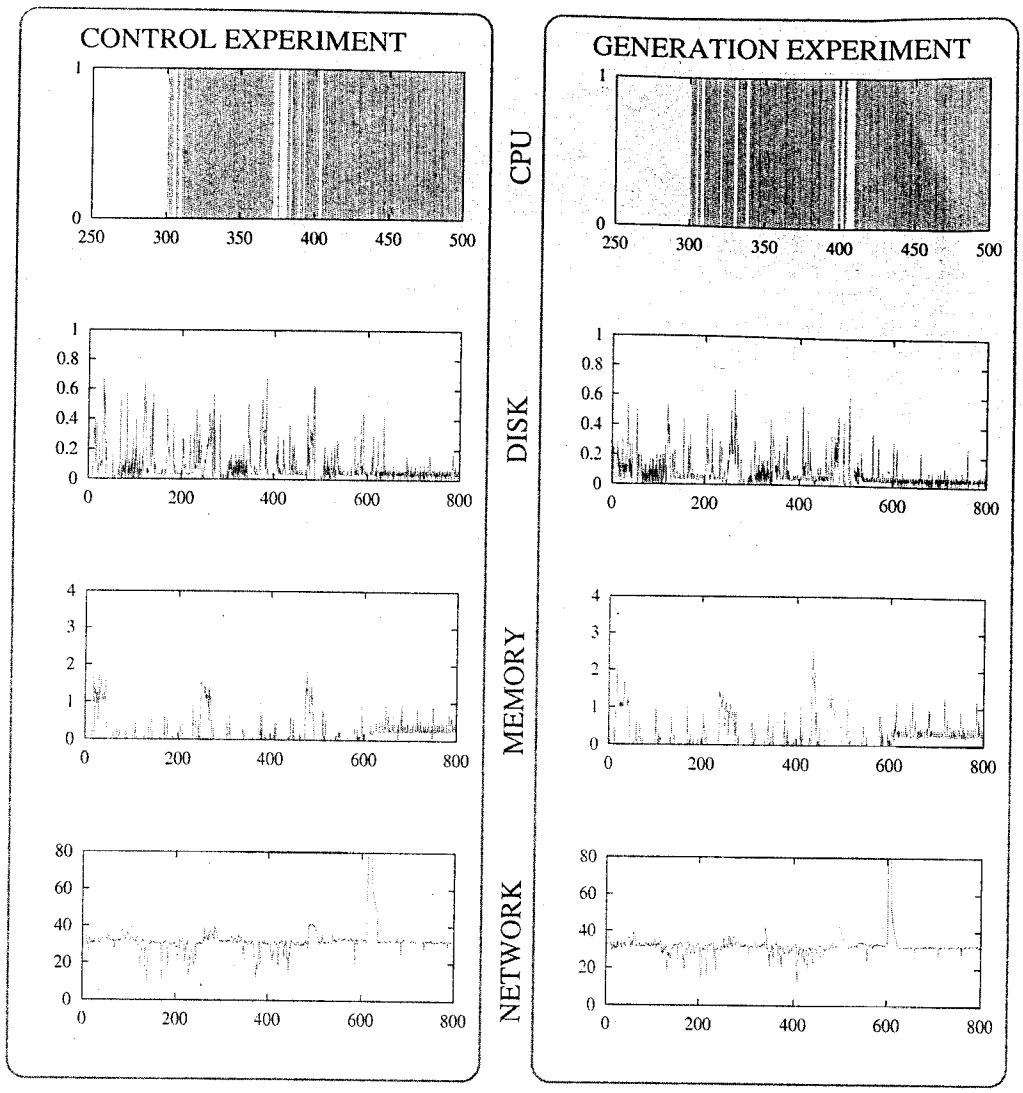


Figure 9. Comparison of true and generated resource-utilization patterns.

regions, then we can say (with 90% confidence) that all the errors for a preemption interval of 6 minutes (360 seconds) will lie in the range [–60,+32] seconds. Further, we believe that the somewhat larger errors along the middle of the *Y*-axis are due to repeated inclusion of a few small intervals having large errors. Even so, Figure 8 shows that there is room for improvement in both the parameterization and parameter-selection of DWG. Perhaps, better modeling of the process-level scheduling algorithm and better parameterization of the generation process will bring DWG even closer to perfect generation.

To evaluate DWG's behavior in the absence of foreground jobs, we performed additional experiments in which no foreground jobs were introduced. In all such experiments, DWG was able to reproduce the checkpoint timings exactly, with zero error.

Comparison of recorded and reproduced loads

For the selected parameter set (7), we compared the load patterns of individual resources from the control experiment against the corresponding load patterns from the generation experiment.

Figures 9 and 10 show the comparison for two of the 17 experiments performed.[9] Visually, we can confirm that the generator reproduces the utilization patterns rather well when the doctoring rules use the selected parameter set.

Figures 9 and 10 show the true utilization patterns of various resources to the left; and the generated ones, to the right. For CPU, the *y*-value is either 0 or 1, indicating whether CPU was busy; the x-axis represents time (1 unit = 20 ms). For the remaining resources, sampled plots are shown, with 100 samples per minute. The x-axes in these cases show time (1 unit = 1/100 min.). The

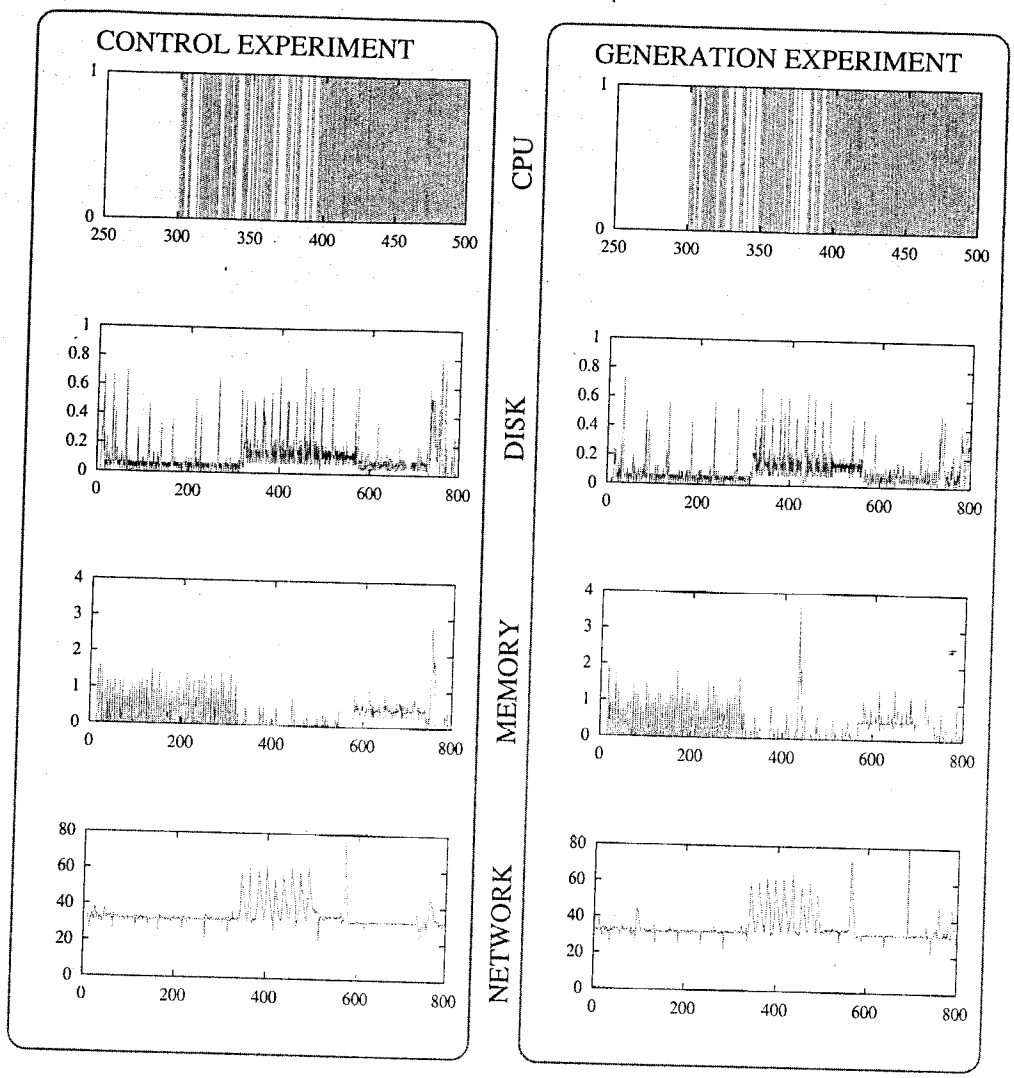


Figure 10. Comparison of true and generated resource-utilization patterns.

y-axes show for disk, transfers per tick; for network, packets per tick; and for memory, the number of free pages.

Physical-level synthetic workload generation is vital for load-balancing experiments. The DWG workload-generation tool accurately replays measured workloads in the presence of competing foreground tasks. To do so, it needs to compensate for foreground-background interactions using dynamic-doctoring rules. The impracticality of exactly modeling process-level scheduling policies precludes perfect doctoring of generated loads. DWG performs reasonably even though it maintains only a few scalar items of information about pending work. Completion-time measurements obtained using DWG are used for learning new load indices as well as new load-balancing policies.[11]

We have applied the workload generator described in this article in designing better load-balancing strategies for our prototype distributed computer system. Using DWG, we generated learning patterns for training a neural network[12] to predict the relative speedups of different sites for an incoming task, using only the resource-utilization patterns observed prior to the task's arrival. Outputs of these comparator networks behave like load indices; they were broadcast periodically over the distributed system, and the resource schedulers at each site used them to determine the best site for executing each incoming task.

To overcome both the uncertainty regarding the actual workload experienced by a job at its site of execution and the delays involved in propagating tasks and

load information across sites, we parameterized our load-balancing policy and tuned it to system-specific values using a genetic learning algorithm.[13] Our other results[11] show that the load-balancing policies learned by our system, when used along with the load indices discovered by our comparator neural networks, are effective in exploiting idle resources of a distributed computer system.

Acknowledgments

NASA contracts NCC 2-481 and NAG-1-613 and NSF grant MIP 92-18715 supported this research.

References

1. D. Ferrari, "On the Foundations of Artificial Workload Design," *Proc. ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM Press, New York, 1984, pp. 8–14.
2. A.K. Nanda et al., "A Replicated Workload Framework to Study Performance Degradation in Shared-Memory Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, Vol. I, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 161–168.
3. W-L. Kao and R.K. Iyer, "A User-Oriented Synthetic Workload Generator," *Proc. 12th Int'l Conf. Distributed Computing Systems*, CS Press, 1992, pp. 270–277.
4. R.B. Bodnarchuk and R.B. Bunt, "A Synthetic Workload Model for a Distributed File Server," *Proc. Sigmetrics Conf. Measurement and Modeling of Computer Systems*, ACM, 1991, pp.50–59.
5. P. Dikshit, S.K. Tripathi, and P. Jalote, "Sahayog: A Testbed for Evaluating Dynamic Load-Sharing Policies," *Software: Practice and Experience*, Vol. 19, No. 5, May 1989, pp. 411–435.
6. S. Zhou, *Performance Studies of Dynamic Load Balancing in Distributed Systems*, Tech. Report UCB/CSD 87/376, PhD dissertation, Computer Science Division, Univ. of California, Berkeley, 1987.
7. K.M. Baumgartner, *Resource Allocation on Distributed Computer Systems*, PhD thesis, School of Electrical Engineering, Purdue Univ., West Lafayette, Ind., 1988.
8. M.J. Bach, *The Design of Unix Operating System*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
9. P. Mehra, *Automated Learning of Load-Balancing Strategies for a Distributed Computer System*, PhD thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, Ill., 1992.
10. M. Berry et al., "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l J. Supercomputing Applications*, Vol. 3, No. 3, 1989, pp. 5–40.
11. P. Mehra and B.W. Wah, *Load Balancing: An Automated Learning Approach*, World Scientific Publishers, Singapore, 1995.
12. P. Mehra and B.W. Wah, "Automated Learning of Workload Measures for Load Balancing on a Distributed Computer System," *Proc. Int'l Conf. Parallel Processing*, CRC Press, 1993, Ann Arbor, Mich., pp. III-263-270.
13. P. Mehra and B.W. Wah, "Population-Based Learning of Load-Balancing Policies for a Distributed Computing System," *Proc. Ninth Computing in Aerospace Conf.*, American Inst. Aeronautics and Astronautics, San Diego, 1993, pp. 1120–1130.
14. P. Mehra and B.W. Wah, "Physical-Level Synthetic Workload Generation for Load-Balancing Experiments," *Proc. First Symp. High-Performance Distributed Computing*, CS Press, 1992, pp. 208–217.

Pankaj Mehra is a member of visiting faculty in the Department of Computer Science and Engineering at the Indian Institute of Technology, Delhi. His current areas of research include performance of parallel and distributed programs, computational science, and machine learning. With Benjamin Wah, Mehra has coauthored *Load Balancing: An Automated Learning Approach* and co-edited *Artificial Neural Networks: Concepts and Theory*. He obtained his PhD in computer science from the University of Illinois at Urbana-Champaign in 1993 and his B.Tech. in Computer Science and Engineering from the Indian Institute of Technology, Delhi, in 1986. He is a member of IEEE, the ACM, and Tau Beta Pi. Readers can contact him at the Dept. of Computer Science and Engineering, Indian Inst. of Technology, Hauz Khas, New Delhi 110016, India; mehra@cse.iitd.ernet.in.

Benjamin W. Wah is a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. His current research interests are in the areas of parallel and distributed processing, knowledge engineering, and optimization. He received his PhD in computer science from the University of California, Berkeley, in 1979. He is Editor-in-Chief of the *IEEE Transactions on Knowledge and Data Engineering*, and serves on the editorial boards of the *Journal of Parallel and Distributed Computing*, *Information Sciences*, *International Journal on Artificial Intelligence Tools*, and *Journal of VLSI Signal Processing*. He will chair the 1996 International Conference on Neural Networks. He is a Fellow of the IEEE and currently serves on the IEEE Computer Society Publications Board, Press Activities Board, and Fellows Committee. He can be reached at the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801; b-wah@uiuc.edu.

Results and algorithms described in this article are different from an alternative design and our early results published before as "Physical-Level Synthetic Workload Generation for Load-Balancing Experiments," *Proc. First Symp. High-Performance Distributed Computing*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 208–217.