Physical-Level Synthetic Workload Generation for Load-Balancing Experiments

Pankaj Mehra and Benjamin W. Wah
University of Illinois

Abstract

Synthetic workload generation uses artificial programs to mimic the resource-utilization patterns of real workloads. It is important for systematic evaluation of dynamic load-balancing strategies because load-balancing experiments require the measurement of task-completion times under realistic and reproducible workloads. We describe a generator that permits accurate replay of measured system-wide loads, thus providing an ideal setting for conducting load-balancing experiments. The generator is implemented inside the operating-system kernel and, therefore, has complete control over the local resources. It controls the usage levels of four key resources: CPU, memory, disk, and network. In order to reproduce accurately the behavior of the process population generating the measured load, the generator gives up a fraction of its resources in response to the arrival of new jobs, and reclaims these resources when the jobs terminate. Our results show near-perfect reproduction of background load even in the presence of interfering foreground load.

I. Introduction

The performance of a distributed system on a parallel stream of independent tasks can be improved by *dynamic load balancing*, which routes incoming tasks from the heavily-loaded sites to the idle or lightly-loaded sites. Load-balancing strategies may be designed for a number of objectives: minimization of total (or average) completion time, maximization of throughput, and maximization of average speed-up over local execution. Since all these objectives depend on task-completion times, accurate assessment of load-balancing performance requires accurate assessment of task-completion time.

The completion time of a task is a sum of its service time and its waiting time. The former is the time required by a task if it is given immediate and sole control of all the resources; the latter, the time spent by a task waiting for resources occupied by competing tasks. We assume that the service times of tasks are unknown; therefore, absolute completion time can be measured but not predicted.

When the operating system employs the round-robin scheduling policy or its variants, the waiting time of a task may be assumed to grow linearly with its service time. Under this assumption, the ratio of a task's completion time to its service time — also known as *stretch factor* — is independent of its service time. Such relative measures of completion time depend upon the loads on the various resources of contention, including the resources for computation (CPU), primary storage (memory), secondary storage (disk), and communication (network).

Measured completion times of tasks are useful in two ways: i) development of new *load indices*; and, ii) development of better load-balancing policies. In the first case, completion times of tasks are measured under a wide variety of *realistic loading conditions*; statistical techniques, such as nonlinear regression, are then used to model relative completion times as functions of the load time series. The functions so developed can be used by a load-balancing policy to rank the alternative destinations for an incoming task [11]. In the second case, the completion times of a set of tasks are measured under *identical loading conditions* but different load-balancing policies. The overall objective of load balancing is then evaluated for each policy. Since the same set of tasks is used for evaluating each policy, and exactly the same loading conditions are used in each case, the differences in evaluation can be attributed solely to the relative quality of different polcies. Such comparisons are useful for benchmarking different policies as well as for automated tuning of existing policies [14]. Thus, realistic and reproducible loading conditions are important for performing load-balancing experiments.

This paper presents DWG, a tool for synthetic generation of realistic and reproducible load. Section II compares and contrasts various methods for synthetic workload generation. Section III describes DWG's approach and its implementation. Section IV presents our initial results using a kernel-based implementation. Section V contains conclusions and directions for future research.

II. Synthetic workload generation

A. Issues

Load balancing and other resource allocation functions operate at the level of process scheduler of the operating system. Studies in these areas require a workload generator that can consistently replay the same load under different policies. In load balancing, the particular sequence of jobs at each site depends upon the particular policy in use. For consistent evaluation, job sequences for every policy must be executed under the same background workload. This means that the combination of a test job and generated load should have exactly the same resource utilization pattern as the combination of that job and the measured load. The exact impact of a test job on an

Research supported by National Aeronautics and Space Administration Contracts NCC 2-481, and NAG-1-613.

existing workload depends upon the process scheduler. Thus, workload generators for load balancing must interact closely with the process scheduler of the operating system in order to generate consistent loads. From the perspective of implementation, the generator should be implemented either at the same level as the process scheduler or below it.

Besides *close coupling with scheduling policies* of the operating system, synthetic workload generators for load balancing must also address three other issues of workload generation identified by Ferrari [8]: (i) *choice of parameters*, namely, what parameters of the system are monitored during measurement and controlled during generation; (ii) *choice of metrics* for comparing measured (actual) load against generated (synthetic) load; and, (iii) reproducing the *dynamic behavior* of real workloads using the workload generator.

B. *Alternative approaches*

Close coupling with the process scheduler can be achieved in two ways. The first method is for the generator to explicitly consider scheduling events such as context switches in deciding what load to generate. For example, the generator may have a trace of instructions executed by each processes, and may switch between these at context switches. The second is for the generator to be implemented at either the same level as the scheduler or below it. For example, in UNIX and related operating systems, where the scheduler is implemented in the kernel, the generator may also be a part of the kernel.

The next issue — namely, the choice of parameters — determines the *level* at which load is measured and generated. Ferrari et al. [6,7] identify three broad levels, with increasing machine independence and decreasing precision of reproduction. The lowest of these is the *physical level*, wherein the parameters are the amounts and rates of utilization of various resources of contention. At this level, the generator imitates only the aggregate behavior of a process population and/or application modules. The next is the *virtual level*, wherein the parameters may be the frequencies or inter-event times of command invocation. Finally, at the *functional level*, the parameters may be the proportions of various commands in an application. The final two levels attempt to imitate the behaviors of high-level entities, such as commands, modules, processes, or application programs. The high-level loads are eventually translated into physical-level loads; but, there is loss of precision in translation: since it is described independently of operating system scheduling events, the same high-level load may generate different physical-level loads. Because of this loss of precision, high-level loads can be compared based only on aggregate metrics, such as percentages and averages on parameters. In contrast, the quality of synthetic physical-level loads can be assessed using both instantaneous values as well as short-term moving averages of their parameters.

There are two ways to address the issue of choosing metrics for comparing measured against generated loads: one, from a performance-oriented perspective; the other, from a resource-oriented one. From a *performance-oriented perspective*, two loads are considered identical if they result in identical performance w.r.t. some user-supplied performance index. From a *resource-oriented perspective*, two loads are considered identical if they result in identical resource-utilization patterns. The former results in loose but perhaps semantically significant comparisons; the latter, precise but not necessarily meaningful comparisons.

The final issue, reproduction of workload dynamics, can be addressed using either a model-based approach or a model-free approach. In the *model-based* approach, a model of workload dynamics is estimated from measured data, which are then discarded. For example, one may estimate either deterministic dynamics using time-series analysis or, stochastic dynamics, by first clustering parameter values and then estimating probabilities of transitions between clusters. In general, information is lost in the process of going from actual dynamics to modeled dynamics. When the parameter distributions are stationary and precision in reproduction is not critical, this is a viable approach. For low-level generation or in non-stationary environments, a *model-free approach* is more suitable. Complete sets of observations are stored for future replay. Within this class, there are two methods: (i) *trace-based*, in which complete instruction traces are stored; and, (ii) *sampling-based*, in which periodically sampled states of the system are recorded. The first requires hardware support for monitoring and buffering; the rate of data generation severely limits the duration of such traces. This is especially difficult for a multiprocessing system. The second only requires a clock capable of generating interrupts, and the information rate can be controlled by the sampling interval. Model-free approaches buy precision at the cost of storage and machine-dependence.

C. *Previous work*

Most existing testbeds for evaluating load-balancing policies using direct experimentation and measurement merely have a stream of jobs that arrive on either an uncontrolled or idle distributed system [2,4,13,17]. This is tantamount to virtual-level synthetic workload generation because the earlier jobs create the 'background' workload for future jobs. Since job sequences are affected by the policy being evaluated, unless load is generated consistently for a variety of job sequences, it may not be possible to evaluate different policies under identical loading conditions. If the chosen goal of load balancing is based on total completion time of the job stream, then such experiments are rendered inadequate by the facts that performance of a job stream is measured only on an idle or uncontrolled system. This problem can be solved to some extent using separate jobs for background load and foreground load, while subjecting only the foreground load to load balancing decisions. On the other hand, if the goal depends upon the speed-ups of individual jobs, then a fair comparison of two different scheduling policies is not possible because they may have been tested under disparate workloads. This requires the precision and close-coupling with process scheduler possible only with physical-level load generation.

The implemented synthetic workload generators that we are aware of operate at the virtual level [3,5,15,16]. One system, SWG [12], accommodates multiple levels but does not exhibit close coupling with the process scheduler; consequently, some precision is lost. Other systems that support physical level load generation [9,16] do not support reproduction of load dynamics.

Existing model-based generators do not store entire load patterns for later regeneration. Instead, they extract various characteristics such as periodicities [12], transition probabilities [5], and inter-event time distributions [3,12].

The model-free generators we are aware of operate at the virtual level, using recorded job traces [10, 17].

D. *Our approach*

Close coupling with process scheduler and precise reproduction of generated load are important requirements of a testbed for load balancing. In this section, we have shown that virtual and functional level generators are unable to satisfy these requirements. The requirement of precision also precludes model-based generation because computer workloads are often difficult to predict accurately. Thus, what is needed is a physical-level workload generator that should interact closely with the process scheduler of the operating system and exhibit model-free reproduction of load dynamics. Existing systems are however geared more towards machine-independence and usually operate at the virtual level. They therefore do not meet our requirements of a testbed for load balancing. With the aim of satisfying these requirements, we have designed DWG, a physical level model-free load generator that, like the process scheduler, is implemented in the operating system kernel.

III. DWG: A Dynamic Workload Generator

An ideal physical-level workload generator for load balancing should be completely integrated into the operating system. It should collect the complete execution trace of each process and know which process to activate at any given time. However, ideal generation has several limitations: i) without special hardware for monitoring and buffering traces, large volumes and high rates of information can severely disrupt memory and I/O traffic; ii) it generates too much overhead for the scheduling algorithm; and, iii) every context-switch and process-preemption event must be replayed, which requires a high-resolution timer capable of generating interrupts. In a typical workstation, the low-level bits of the clock driving the system are masked off to prevent the overhead of saving and switching contexts everytime there is an interrupt; therefore, an ideal generator requires hardware modifications and, even then, it may be too inefficient to implement.

A realistic generator can measure and generate only certain sampled values of the system's state. While real events may occur at the resolution of few tens of nanoseconds, measured and generated events may occur only at the resolution of few tens of milliseconds. This assumes that i) system states are context-free or Markovian, which is generally true of a running system in which random events such as interrupts and preemption by unrelated processes are frequent occurrences, and ii) the system behavior is stationary so that averages of sampled behavior are meaningful statistics. In this case, the measurement process is a Markov chain embedded in a larger and more complex chain corresponding to the true system. Sampling allows measurement and generation to proceed at rates that neither cause excessive resource overhead nor require any hardware modifications.

An important design issue is the rate of generation. Measurement and generation events should occur at the same rate. Otherwise, signal processing limitations may limit the abilities of either the generator to reproduce certain measured loads or of the measurement routines to measure certain generated loads. Measurement and generation events may be triggered by either periodic interrupts from a real-time clock or context switches in a process scheduler. The first, clock-based, approach produces a periodic sample of observations, which can be replayed by issuing resource-consumption instructions periodically. The second, context-switch-based, approach produces an aperiodic sample whose precise reproduction may require a very high resolution clock for reproduction of scheduling events. In DWG, we adopt the first approach because we do not have access to a timer device capable of producing interrupts at the resolution required by the second approach.

The overall operation of DWG can be described in 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. (There are 60 interrupts per second.) In the second phase, which is performed off-line, provision is made for running test jobs in conjunction with the recorded load. This is done by inserting *traps*, or data items in a special format, at appropriate points in the recorded data. During the generation phase, one recorded item is read at every clock interrupt and, unless it is a trap, load proportional to the measured load is generated by issuing synthetic resource-usage instructions. When a trap is encountered, depending on its contents, the generator updates its internal count of the number of processes constituting the forground load. It then signals the local job manager to carry out the operation (starting or terminating a job) specified by the trap. Using this approach, recorded loads can be replayed and test jobs introduced on top of generated loads at prescribed times. Our goal is to implement DWG in such a fashion that the behavior of test jobs on top of generated loads would closely approximate their behavior on top of real loads. The rest of this section describes the architecture of DWG; the next section describes experimental assessment of DWG's load-generation capabilities.

DWG measures and generates load at exactly the same frequency using interrupts from a real-time clock device. Figure 1 shows the basic architecture of the measurement and generation functionality, including mechanisms for data handling, asynchronous event management using traps, and dynamic determination of generated load levels.

A. *Measurement of workload*

The core of DWG, consisting of its measurement and generation routines (Figure 1), resides in the operating system kernel. This functionality is periodically invoked by a clock interrupt. It samples the system state and records (i) whether or not the CPU is busy; (ii) the number of free memory pages; (iii) the number of disk transfers since last interrupt; and, (iv) the number of packets active on the network since last interrupt. Also recorded with each data item are the number of local and global processes generating the current load. The interrupts occur anywhere between fifty to a hundred times per second. Therefore, this data can grow at a fantastic rate. The information is compressed into a few bytes and periodically removed from the kernel by an external process. Similarly, during generation, information needs to be fed to the kernel at the rate of a few hundred bytes per second. In order to keep the number of data transfers to a minimum, buffer pairs are allocated inside the kernel. Data transfer can proceed using the idle buffer while the kernel is busy reading/writing the other buffer. The size is chosen large enough so that there

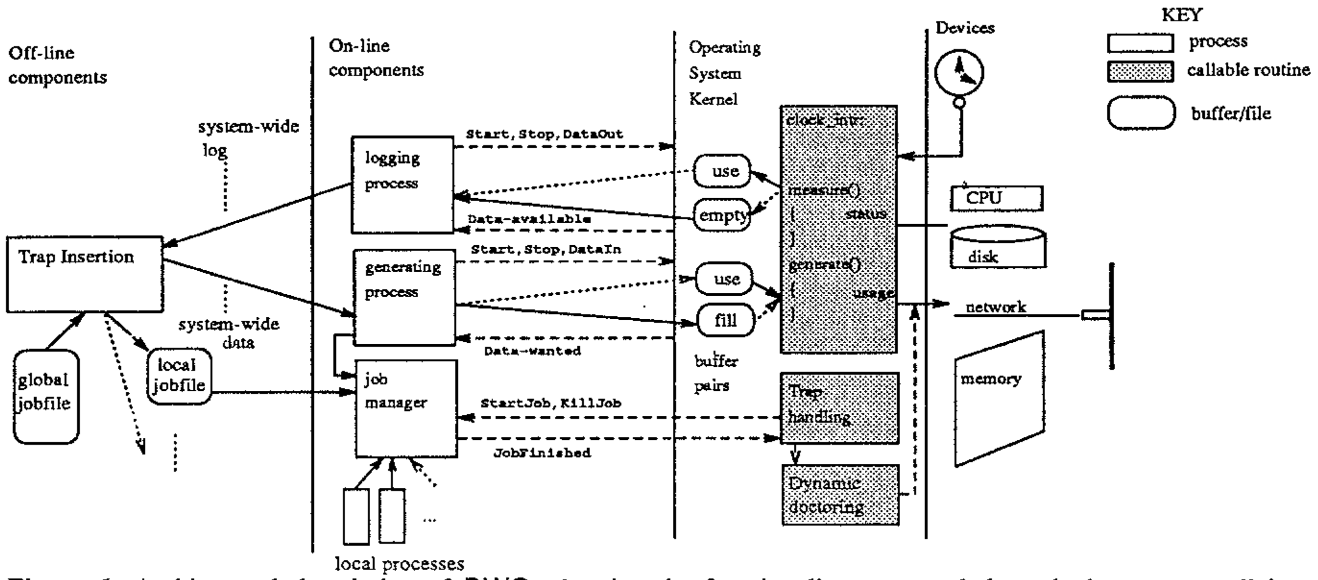


Figure 1. Architectural description of DWG, showing the functionality supported through the system call interface. In addition, there is software for coordinating synchronized distributed start-up of measurement and generation. Trap insertion considers logs obtained from multiple sites in order to insert traps that help the disk and network traffic generators keep track of global process populations.

are at most only a few transfers per minute.

B. Initiation and termination of test processes

Events related to test processes include initiation and termination of local processes, and (only at sites generating load for shared resources) the initiation and termination of global processes. DWG uses traps in recorded data for determining when to trigger such events during generation. The purpose of introducing these traps is to let the generator know about the status of competing processes that exist during generation but did not exist at the time of measurement. As explained below (Section III.C), the presence of such processes warrants a reduction in generated load; the amount of reduction depends on the number of such processes. Since the number of such processes changes only when a new process is introduced or an old process terminates, traps trigger an update of process counts inside the generator precisely when the size of the competing process population changes. Thus, before measured data is given back to the kernel for replay, traps are inserted at the starting and (if known) stopping points of jobs. If the stopping point of a job is unknown, the job manager (Figure 1) notifies the kernel dynamically when such a job stops.

In determining where to insert traps in pre-recorded system-wide logs, which should have been obtained by a synchronized start-up, the trap-insertion routines use global *jobfiles*. These contain information about: (i) the site at which a job will be executed; (ii) starting time of the job; (iii) 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, (iv) the command and arguments needed for starting the job. The trap-insertion routines associate a unique global identifier with each such job, and partition the global jobfiles into local jobfiles, which are passed on, by the generating process, to the individual job managers at each site (Figure 1).

The trap-insertion routines essentially perform an event-driven simulation, using the offset into the logged data as simulated time. The events in jobfiles are sorted primarily by their starting times, and secondarily 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 trap-insertion routines maintain two eventlists headed by, respectively, the next job to start, and the next job to stop (if its stopping time is non-zero). At every 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. If stopping times of some events are not known, then it is the job manager's responsibility to notify the kernel when such an event terminates.

Using traps in data as well as the job-manager-initiated calls to their trap-handling routines (Figure 1), the kernels at various sites generating load for shared resources are able to track the global process population size, and all kernels are able to track the sizes of their local populations. The doctoring routines, described next, use this information to dynamically adjust generated load.

C. "Doctoring" rules

The processes generating the real workload are subject to scheduling. In UNIX and related operating systems [1], the scheduler maintains queues of ready-to-run processes, each queue corresponding to a priorty level. It essentially goes through these processes in round-robin fashion within each queue, and in order of priorty among the queues. Processes having low priorty can be preempted by the ones having higher priory. Priorties are recomputed periodically, thus causing processes to move between queues. In contrast, the generating process is not subject to scheduling; it behaves like a high-priorty real process. If the generator were to always reproduce the measured load exactly (which it can do rather well; Section IV.A), test jobs introduced into the system would encounter greater delays under a generated load than under the corresponding real workload.

Ideally, the generator would need to implement the entire queueing discipline in order to emulate the behavior of the measured load in the presence of an additional process. If all the information about process start and stop times were available, this could possibly be done off-line in a fashion similar to the trap insertion process. However, stopping times of test jobs are usually unknown, and are, in fact, dependent on the load. In this case, an ideal generator would have to implement the queueing discipline of the scheduler *on-line*! That would be prohibitively expensive computationally. As a compromise, DWG makes certain simplifying assumptions that allow it to compute the altered behaviors dynamically without incurring too much computational overhead. This component of our generator, containing rules for altering the generated load in the presence of competing test jobs, constitutes its *doctoring rules*.

We assume, first, that all the processes (measured and new) have the same priorty and that resources are allocated to them in a round-robin fashion; and, second, that the consumption levels of all resources are identically affected by competition from test processes. For example, a 15% reduction in CPU usage warrants a 15% reduction in memory occupancy, and so on. An alternative is to determine the instantaneous relative usage of each resource by separately running the test job on an idle machine. In that case, the log of recorded usage can be used subsequently by the doctoring routines to determine exactly how much reduction to apply to other resources relative to the reduction in (say) CPU usage.

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 space. On the other hand, such buffers do not exist for the CPU and memory resources. Therefore, their usage needs to be explicitly spread out over time by buffering unfulfilled requests in the generator.

Finally, we assume that the load on CPU and memory is affected only by the local process population whereas that on disk and network traffic, by the system-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 as in the case of NFS; in other models, disk may be treated as a private resource.

Under these assumptions, the reduction in generated load can be computed as a ratio of the process-population sizes at the times of measurement and generation. As mentioned above, global population sizes are used for shared resources (disk and network), and, local, for private resources (CPU and memory). The information required for this computation is obtained as follows (Figure 1): (i) local population sizes at the time of measurement are recorded by the measure() routine; (ii) global population sizes are computed by the trap insertion process, off-line, by summing up local process population sizes from different logs; (iii) the trap-handling routines maintain an up-to-date counts of (local and global) test processes at the time of generation; and, (iv) the dynamic doctoring routines of the kernel compute the aforementioned ratios.

However, as illustrated with experiments in Section IV.B, plain reduction in load levels is insufficient for reproducing the behavior of the measured process population. The processes constituting the measured process population, when deprived of the full use of the original resources, would have required more time to finish their computations. Therefore, whenever the generator fails to achieve the measured load levels, either due to competing processes or (as explained below) due to finite resources, it carries the difference of measured and generated loads over as pending load for subsequent cycles. At certain time instants, the pressure of pending work may overwhelm the generator to such an extent that, instead of getting new work from measured data, it will be forced to spend one or more cycles just getting rid of pending work. This happens in two cases: (i) when the pending load on CPU exceeds the portion of the clock interval devoted to CPU load generation; and, (ii) when the pending load on memory exceeds the number of currently available free pages.

When the load to be generated hits the ceiling on either of the two constrained resources described above, the generator determines the more constraining of these resources. (In the event of a tie, CPU is considered more constraining than memory.) It then computes, w.r.t. the most constraining resource, the fraction of pending work that can be accommodated in the current generation interval. The same fraction of pending load on the remaining resources is then added to the current loads of those resources. On the combined load, reductions specified by the doctoring routine are applied. Those reductions are then subtracted from the current load and added to the pending load for each resource. The 'doctored' load thus computed is then generated. This cycle of doctoring and generation is performed once every clock interrupt.

D. Basic mechanisms of the generator

The simplest generators are for disk and network traffic. The former writes out as many blocks to the disk as the number of transfers computed from the measured disk traffic by the doctoring routines. The latter broadcasts as many packets as the doctored number of packets. The other two generators are for constrained resources, and can only accept bounded amounts of work.

The statistics for CPU usage, unlike the other three, are *sampled* quantitities. The code for sampling the CPU state is contained in the measure() routine of the kernel. While there may be millions of different CPU states per second, only a few tens of clock interrupts can occur every second for reasons of efficiency. (This problem may be overcome using hardware instrumentation.) Knowing that the sampled state was busy, it is not at all clear whether the generator should attempt to occupy the CPU for an entire interval between successive clock interrupts. The code for CPU generation is contained in the generate() routine of the kernel. Occupying the CPU for a fraction of that interval requires a high-resolution timer, which can either interrupt faster than the kernel's clock or be examined repeatedly for determining the fraction of clock interval consumed so far. In our implementation on Sun Microsystems Inc.'s SUN 3 workstations, we installed a microsecond-resolution timer for this purpose. Using this timer, we can consume, say, sixty or eighty percent of the interval between successive clock interrupts whenever the sampled CPU state is busy. The exact interval does not matter because during replay in load balancing experiments, we attempt to remove the bias due to ad hoc parameters by repeating each experiment under many different parameter settings. Certain other workstations already have a microsecond timer; on these, installation of hardware timers may be unnecessary.

An alternative way to generate computational load, also inside the kernel, without using a separate high-resolution timer, is based on *calibration*. Off-line, in a user process, a program fragment comprising a loop enclosing a small computation is repeatedly executed. The number of iterations of the loop is plotted against the time consumed by the program fragment. The resulting data structure is a calibration table, using which the appropriate number of iterations for consuming a given fraction of the clock interval can be determined by look-up. The trap-insertion routines can then use these tables off-line to convert measured CPU state into the associated number of loop iterations. During doctoring, the number of iterations may be reduced if there are competing processes at the time of generation. Where possible, the timer-based approach is simpler and does not require storage of calibration tables; otherwise, the calibration-based approach offers a viable alternative.

The generator for memory usage maintains its own pool of free pages while attempting to keep the count of free memory pages as close as possible to the doctored count. This is not always exactly achievable because processes may sometimes claim those memory pages that should have been free. When it needs to free up some memory, the generator frees at most as many pages as it is currently holding. When it needs to consume some, it only takes up as many pages as are currently available on the system. The unfulfilled needs of the generator add to its pending needs. When extra free memory is available, the generator may hold onto some extra pages for a few cycles in order to satisfy its pending needs.

The memory-occupying component of the generate() routine (Figure 1), maintains counts of the pages being held by it as well as a count of those held only to satisfy pending needs. Pending memory needs are stored in units of work (utilization×time). Therefore, one page occupied for one clock interval reduces the value of pending needs by one. It is important to note that in our current method, we do not issue memory references during generation in order to simulate memory usage. Where necessary, as for evaluation of memory-prefetching strategies or cache-coherence protocols, some more sophisticated generation technique may be used.

The software and hardware modifications made to our workstations in order to implement DWG are discussed in the appendix. Our approach is easily extensible to shared memory systems as well as to systems having point-to-point links or multiple disks. The important principle is to have one generator per resource and to keep track of the number of processes loading that resource during both measurement and generation.

IV. Initial Results & Discussion

Initial results from DWG are available for the generation of computational and memory loads. The part on the generation of disk and network traffic is not fully operational. The means of evaluation employed are: (i) visual comparison of measured and generated loads; and, (ii) statistical analysis of delays and errors resulting from imperfect generation. Two experiments are described here: the first compares the measured and reproduced loads when a single job (comprising a single compilation command) is introduced into an idle system; and, the second, when a job (containing a single compilation command) is introduced while a background job (comprising two compilation commands) is running.

A. *Replay without competing jobs*

Through the first experiment, we illustrate the accuracy with which the generation mechanisms can replay measured loads when no additional jobs are introduced during generation. This experiment proceeds in three simple steps:

(i) The load on the idle system is measured. Call this log0.

(ii) A job j is introduced at time t_j into the replay of log0 using DWG. The resulting combined load is measured; call this log0.j.

(iii) The load log0.j is replayed using DWG. No additional jobs are introduced. The generated load is measured; call it log0+j.

Figures 2(a) and 3(a) show, respectively, the CPU and memory components of log0.j; Figures 2(b) and 3(b), the corresponding components for log0+j. Due to the density of the data, the utilization levels are plotted with impulses. For CPU, high activity shows up as dark bands; for memory, as troughs in the curve. The measured loads indicate the starting and stopping points of the job as 200/60 sec. and 580/60 sec., respectively, from the start of the experiment. The generated loads track this activity well and their start and stop times match those measured.

Because measurement of generated load is initiated by a user process, small delays can creep in if the process is late at getting control of the CPU. (The fixed lag between measured and generated loads can easily be removed by triggering both measurement and generation using the same system-call command. This mechanism will be implemented in the near future.) In this experiment, these delays seem to be the only difference between the measured and generated loads. With this hypothesis, we computed both the average signed error between the two patterns, as well as the root mean square error (RMSE), at different lags. It was found that for the curves of Figures 2 and 3, a lag of 9 produced minimum error. This agrees well with the visual determination of shift between the curves. At this lag, for the CPU load, the average signed error was found to be -0.01, and the RMSE was found to be 0.1. For the memory load, also at a lag of 9, both these errors were zero!

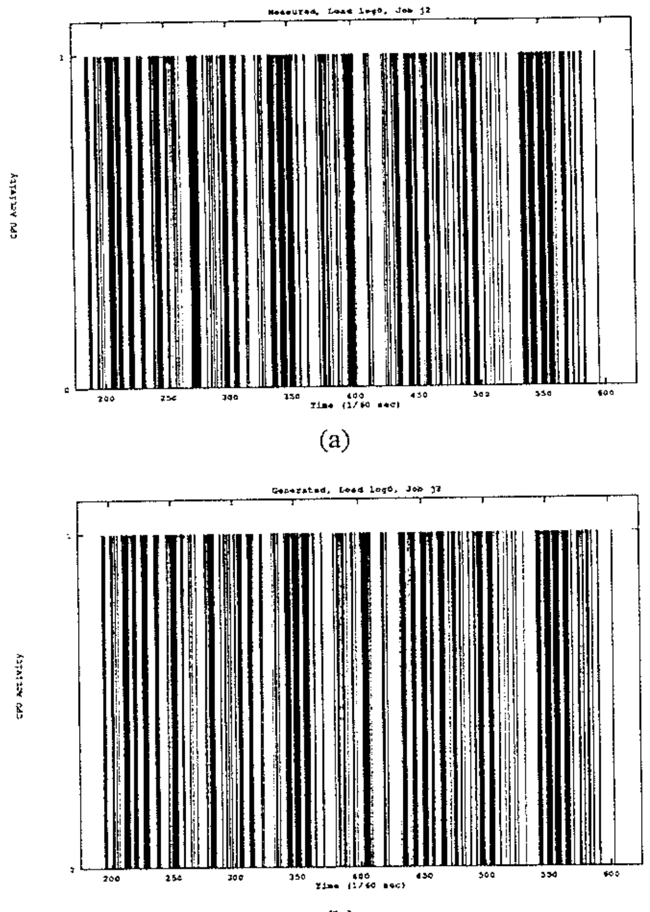


Figure 2. The measured and reproduced CPU utilization patterns due to a single C compilation command 200/60 seconds into the experiment; the command terminated normally at about 580/60 seconds.

B. Replay with competing jobs

The second experiment demonstrates the behavior of the generator in the presence of competing tasks. The job j2 is run in the presence of, respectively, the real and generated loads due to the compilation commands contained in job j1. This experiment proceeds in four steps; the first of these is the same as in the first experiment. The remaining steps are:

(ii) A job j1, consisting of two compilation commands whose (overlapping) executions start at, respectively, 100/60 and 350/60 seconds into the experiment. The background load is log0 replayed by the generator. The resulting combined load (called log0.j1) is measured.

(iii) A job j2, consisting of a single compilation command whose execution starts at 200/60 seconds into the experiment, is introduced while the generator replays log0.j1. The resulting combined load (called log0.j1.j2) is measured.

(iv) If the generator is indeed performing as desired, then the results of the previous step should mimic the measurements resulting from the introduction of the combined compilation commands contained in j1 and j2, where each command is introduced with the same delay as before, while the generator replays log0. The combined load is measured.

In the first experiment, for which the memory component of measured and generated log0.j1.j2 are plotted in Figures 4(a) and 4(b), respectively, the generator only reduces its load level in response to test processes but does not reclaim the lost work later. It thus chooses to do less work (load × time) overall than the measured process population. Due to the reduced competition, the three jobs finish about three seconds sooner under the generated load than under the actual load. Ideally, not only should the generated load be shrunk but also the discounted work should be added to the pending load. This will result in a *smeared* (reduced and spread out) load. Figure 6 illustrates the result of such smearing of generated load.

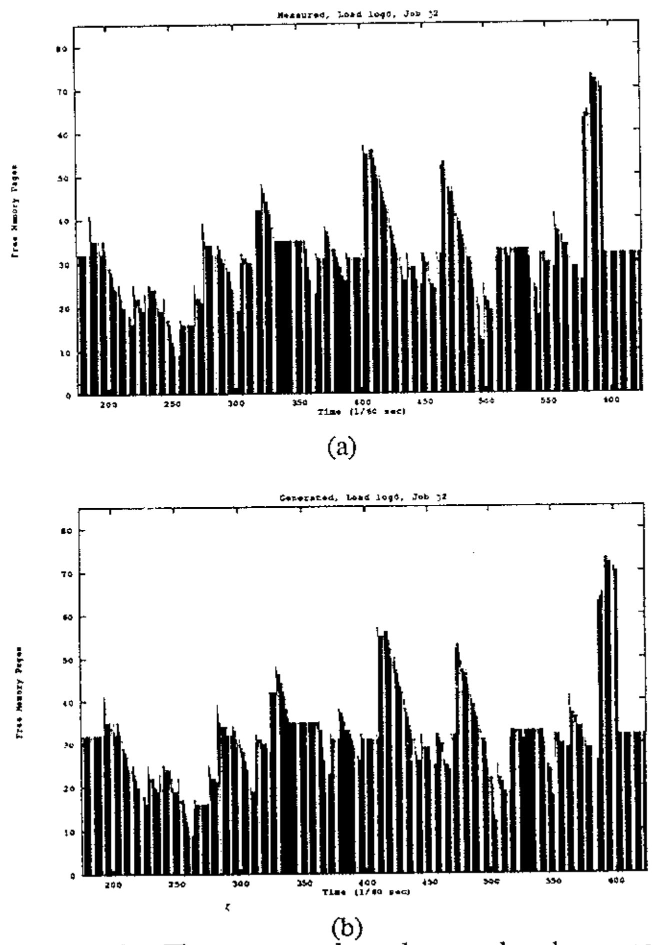
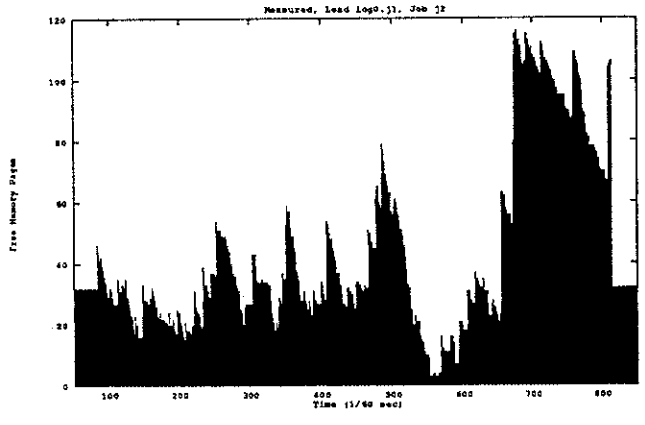
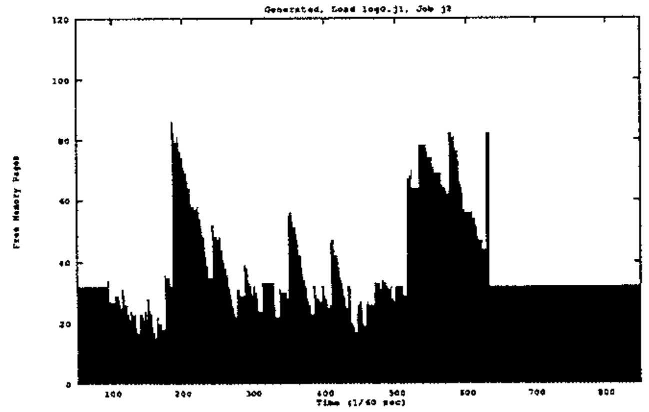


Figure 3. The measured and reproduced memory utilization patterns due to a single C compilation command 200/60 seconds into the experiment; the command terminated normally at about 580/60 seconds.

Because the generated load in Figure 4(b) does not appear to be a simple shifted copy of the measured load, it is not feasible to compute RMSE at various (fixed) lag values in order to match the entire curve. However, certain features on the measured curve appear to match certain others on the generated curve. I.e., parts of the generated curve appear to lead or lag corresponding parts of the measured curve by variable amounts. Figure 5 shows where the optimal lags (corresponding to the least RMSE) are located for various windows of 50 successive observations, each window starting 25 observations ahead of its predecessor. (Negative lags imply that the generated curve leads the measured curve; i.e., either the measurement was started too soon or the generator did not do enough work.) The plummeting lags suggest that the generator is not producing enough load, especially after the the introduction of the first job 100/60 seconds into the experiment.



(a)



(b)

Figure 4. The measured and reproduced memory utilization patterns due to three C compilation commands introduced at, resp., 100/60, 200/60, and 350/60 seconds into the experiment. In (a) log0 was replayed by the generator and measurements obtained as the three jobs were introduced. In (b), the load log0.j1 was replayed which included the regeneration of the first and third compilation commands; the measurements were obtained as the job j2 comprising only the second compilation command was introduced.

Figure 6 shows the results of performing the same experiment when the generator attempts to preserve the total amount of work done by reclaiming pending work at later times. The figure clearly shows great improvement both in the level as well as the spread of utilization, especially when multiple jobs are active, i.e., in the 200-500 time units range. Figure 7(a) shows the location of optimal lags for matching portions of the new generated curve for the memory resource with those on the measured curve. In contrast to Figure 5, Figure 7(a) clearly shows increase in lags as new jobs are introduced into the system at, respectively, 100/60, 200/60, and 350/60 seconds. There is also a reduction in lags as jobs finish. Figure 7(b) shows the corresponding curve for the CPU load.

The lagged crosscorrelation (assuming fixed lags) between the entire curves of Figures 6(a) and 6(b) was found to reach a peak at a lag of 107 time units. This is very close to the average value (assuming variable lags) of the optimal lags shown in Figure 7(a) and 7(b), for both CPU and memory. The good match between a global statistic (crosscorrelation of the entire pattern) and the average of local statistics (optimal lags for subpatterns) suggests stationarity in the generation process. This observation is supported by the plots in Figure 7 showing that lags tend to stabilize (growing when a new process is added to the system and shrinking when that process is terminated). From this, it is evident that the generator is slow at tracking high loads and is forced to compenstate for its slowness by increasing its duration of activity. Future improvements in the generator should focus on eliminating this problem. Ideally, the lag plot should be as flat as possible. We conclude that although events in the generator are delayed by a few seconds relative to the true events, their occurrences remain discernible and the overall characteristics of measured behavior are preserved during generation.

One possibility we are considering for ensuring the flatness of the lag curves is based on inserting

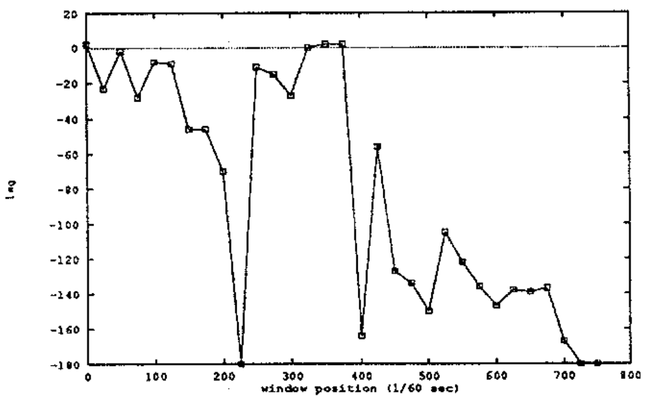
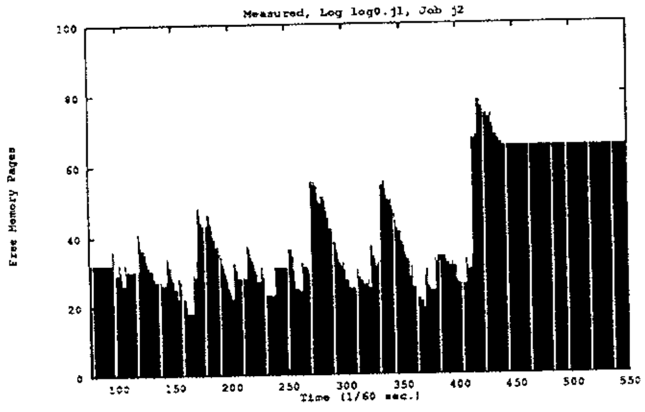


Figure 5. Optimal lags (the lags for which the RMSE between the measured and generated curves is minimum) when windows of size 50 from the measured curve are matched against (lagged) windows of size 50 from the generated curve. The abscissa shows the starting positions of these (overlapping) windows.

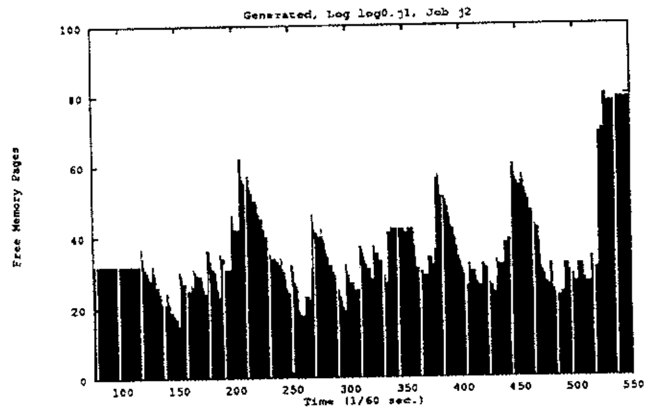
synchronization points in test jobs, and then recording the measured utilization of the test job when it is run on an idle machine. When running this job on top of a generated load, we can prevent the generator from lagging or leading the job by increasing the volume of generated load when a synchronization point occurs too soon, and reducing the generated load when it occurs too late. Certain other parameters of the doctoring algorithm can be tuned to improve the quality of generated load. These include (i) the computation of reduction levels applied to the current load in response to an increase in the number of competing processes; (ii) determination of the amount of pending work carried over to subsequent steps; and, (iii) thresholds used for determining whether pending work is sufficient to delay current load entirely. Improved generation can be expected when these parameters are tuned.

V. Conclusions and Future Work

Generation of physical-level system-wide workload is essential for performing load-balancing experiments. We have described DWG, a workload-generation tool that resides in the operating-system kernel. Empirical data suggest that the tool is able to accurately replay measured workloads in the absence of competing tasks. However, in the presence of such tasks, it needs to incorporate a 'delayed work' effect. Precise computation of this effect is precluded by lack of models and ignorance of future load. Our generator is able to perform reasonably even though it maintains only a few scalar items of information about

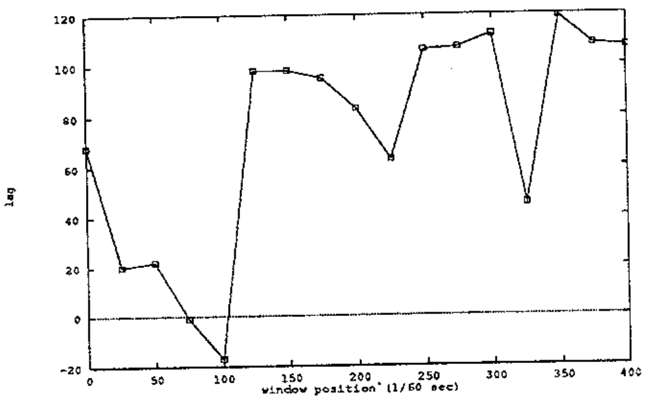


(a)

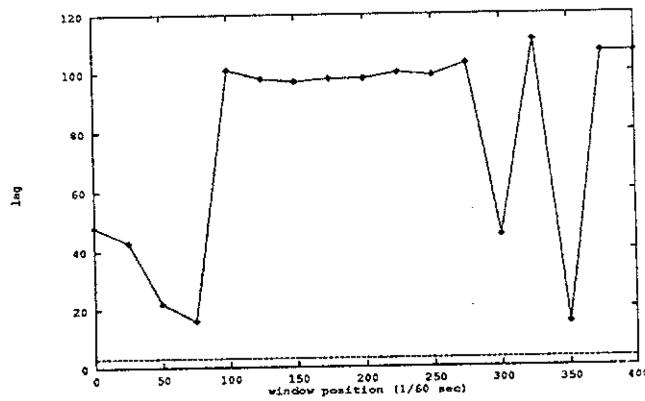


(b)

Figure 6. The measured and reproduced memory utilization patterns due to three C compilation commands introduced at, resp., 100/60, 200/60, and 350/60 seconds into the experiment. In (a) log0 was replayed by the generator and measurements obtained as the three jobs were introduced. In (b), the load log0.j1 was replayed which included the regeneration of the first and third compilation commands; the measurements were obtained as the job j2 comprising only the second compilation command was introduced. The generated loads employ load smearing to delay background work not finished due to competing processes.



(a)



(b)

Figure 7. Optimal lags (the lags for which the RMSE between the measured and generated curves is minimum) when windows of size 50 from the measured curve are matched against (lagged) windows of size 50 from the generated curve. The abscissa shows the starting positions of these (overlapping) windows. These curves show the lags for (a) the memory load, and, (b) the CPU load, generated with the doctoring algorithm that applies load smearing to background load.

pending work.

Recent versions of DWG provide functionality for synchronized start-up of generation and measurement, thereby eliminating the initial lag between the measured and the generated loads. Moreover, by using the UNIX `rdate` utility, we have successfully eliminated the lags caused by clock skew between sites. Completion-time measurements obtained using DWG are being used for learning new load indices [11] as well as new load-balancing policies [14].

Appendix

The details of software and hardware modifications made to the SUN 3-50 and 3-60 workstations on which the experiments reported here were performed are as follows.

1. A microsecond-resolution timer chip from Zytek, Inc. was installed in the (unused) DES socket of the CPU board. The device driver and necessary modifications to the system call interface were also provided by the vendor.
2. The mechanism for generating memory traffic directly modifies the count of free memory pages maintained by the virtual memory management software. In addition, the generator keeps track of total memory pages held by it, including those being held to satisfy pending needs.
3. The interface from the kernel to measurement and generation processes as well as to the job manager is signal-based. The data transfer buffers each occupy one-half of a memory page (or 8Kb, on our system). Each instant's measurement/generation codes occupy 4 bytes, resulting in a capacity of 1000 items per buffer. At approximately 60 clock interrupts per second, we need fewer than four data transfers per minute.
4. One new system call was added to the kernel, which supports all the functionality of the process-kernel interface.
5. The mechanisms for generating disk and network activity simulate the effects of repeated calls to, respectively, the write() and sendto(), system calls. The file and socket descriptors necessary for this are set up externally in the generating process and passed into the kernel prior to the start of generation.
6. Load reductions due to doctoring, for the memory resource, are applied to 'memory in use', computed as the difference of maximum available memory and the current count of free memory pages.
7. If there are delays in generation, as evidenced by Experiment 2 in Section IV, the start and stop times of jobs may become inaccurate because generation delays cannot be estimated at the time of trap insertion.
8. During generation, pending needs for CPU and memory are given precedence over the current needs of the simulated process population on these resources.
9. Synchronized system-wide start-up of measurement and generation capabilities is performed by setting a prescribed time for starting the experiment. Therefore, arbitrary lags may creep in due to clock skews. This is largely unavoidable even with the most sophisticated distributed synchronization algorithms. Other delay effects are kept to a minimum by allowing the measurement and generation processes to settle before issuing the starting system calls.

REFERENCES

[1] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[2] K. Baumgartner and B. W. Wah, "GAMMON: A Load Balancing Strategy for a Local Computer System with a Multiaccess Network," *Trans. on Computers*, vol. 38, no. 8, pp. 1098-1109, IEEE, Aug. 1989.

[3] R. B. Bodnarchuk and R. B. Bunt, "A Synthetic Workload Model for a Distributed File Server," *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 50-59, ACM, 1991.

[4] P. Dikshit, S. K. Tripathi, and P. Jalote, "SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies," *Software — Practice and Experience*, vol. 19, no. 5, pp. 411-435, John Wiley and Sons, Ltd., May 1989.

[5] R. T. Dimpsey and R. K. Iyer, "Performance Prediction and Tuning on a Multiprocessor," *Proc. Int'l. Symp. Computer Architecture*, pp. 190-199, ACM/IEEE, 1991.

[6] D. Ferrari, "Characterizing a Workload for the Comparison of Interactive Services," *Proc. National Computer Conf.*, pp. 789-796, AFIPS Press, 1979.

[7] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[8] D. Ferrari, "On the foundations of artificial workload design," *Proc. ACM SIGMETRICS conf. on Measurement and Modeling of Computer Systems*, pp. 8-14, 1984.

[9] J. Geers, "A New Generation of Benchmarking," *MIPS Magazine*, pp. 92-98, Feb. 1989.

[10] K. K. Goswami and R. K. Iyer, *Dynamic Load-Sharing Using Predicted Process Resource Requirements*, Tech. Rep. UILU-ENG-90-2224, Coordinated Sci. Lab., Univ. of Illinois, Urbana, 1990.

[11] T. Karnik, B. W. Wah, and P. Mehra, *Learning to Predict Relative Completion Times using Comparator Neural Networks*1992 (submitted for review).

[12] D. L. Kiskis, M. H. Woodbury, and K. G. Shin, *Design, Implementation and Application of Synthetic Workload Generators for Real-time Systems*, Tech. Rep. CSE-TR-25-89, Computer Science and Engineering Division, Univ. of Michigan, Ann Arbor, MI, 1989.

[13] T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Trans. Software Engineering*, vol. 17, no. 7, July 1991.

[14] P. Mehra and B. W. Wah, "Adaptive Load-Balancing Strategies for Distributed Systems," *Proc. 2nd Int'l Conf. on Systems Integration*, IEEE, Morristown, NJ, June 1992.

[15] A. K. Nanda, H. Shing, T-H. Tzen, and L. M. Ni, "A Replicated Workload Framework to Study Performance Degradation in Shared-Memory Multiprocessors," *Proc. Int'l. Conf. Parallel Processing*, vol. I, pp. 161-168, IEEE, 1990.

[16] K. Sreenivasan and A. J. Kleinman, "On the Construction of a Representative Synthetic Workload," *Communications of the ACM*, vol. 17, no. 3, ACM, 1974.

[17] S. Zhou, *Performance Studies of Dynamic Load Balancing in Distributed Systems*, Tech. Rep. UCB/CSD 87/376 (Ph.D. Dissertation), Computer Science Division, Univ. of California, Berkeley, CA, 1987.