Vol. 42 No. 1 IPSJ Journal Jan. 2001

Regular Paper

A Framework for Performance Evaluation Based on Event Tracing

Takashi Horikawa[†,☆]

A framework is described for establishing a formulated method of performance evaluation based on event tracing. It consists of abstractions representing a system, its behavior, and the events to be measured. It also includes formulations of performance measures and of algorithms for obtaining them from an event trace. Measures include not only such usual ones as throughput, response time, and resource utilization, but also those describing such process interactions as process waiting time and semi-busy-waiting. The framework has been applied to the performance evaluation of an Apache HTTP server running on Linux 2.2.14 and 2.3.41 and executing the SPECweb96 benchmark program. This case study has clarified the differences between the two versions of Linux, including differences in CPU usage, global-kernel-lock usage, processing time, and process-switch frequency. The case study also indicates that Linux 2.2.14 had a performance bottleneck other than physical resources, thus the measurement of CPU usage alone is not enough; we also need to consider object interactions including interprocess communications, at least so far as Apache HTTP servers are concerned.

1. Introduction

Performance evaluation is necessary throughout the entire software development lifecycle: design, coding, testing, and operation. Early detection of performance problems is particularly important because the effort needed to solve problems increases the longer detection is delayed.

Performance estimation is especially useful in the design and early coding phases to ascertain whether the target system will effectively process users' jobs or not. The queueing network models and simulation models usually used to estimate performance require the input of parameters, often obtained by performance measurements in actual systems[12]. As Smith[14] has shown, for performance estimation, event tracers (she calls them "event recorders") are on the whole significantly more effective than sampling-based tools.

Bottleneck detection is useful in the later coding and testing phases. Event-driven measurement tools have also been used with good success in bottleneck detection that is designed to tune performance. Typical case studies in this regard include the performance tuning of an interrupt-handling routine[10], of a disk-intensive program[5], of multithreaded programs[8], and of a squid proxy server[17].

Some event-driven measurement tools are commercially available (such as IBM's General Tracing Facility and TNFtracing on Solaris[16]), while others are used exclusively for the research being conducted by their developers (such as Haigh's tracer for UNIX[6] and the tools used in the above case studies). Because these tools have been designed to be used for specific purposes on target machine architectures, operating systems, etc., they measure different sets of software events and are not standardized.

My motivation in developing a event-trace framework was to provide a guiding principle for unifying the disparate approaches that have so far characterized event-based performance evaluation. The proposed framework consists of abstractions representing a system, its behavior, and the events to be measured. It also includes formulations of performance measures and of algorithms for obtaining them from an event trace.

This paper is organized as follows. Section 2 describes an event abstraction and its preparations. Section 3 describes a formulation of trace analysis. Section 4 applies my framework to a case study in which the performance of an Apache HTTP server running on Linux is evaluated. Section 5 briefly reviews related work, discusses how this study has advanced the field, and introduces some possible applications. Section 6 concludes the paper.

† C&C Media Research Laboratories, NEC
☆ Presently with Development Laboratories, NEC Networks

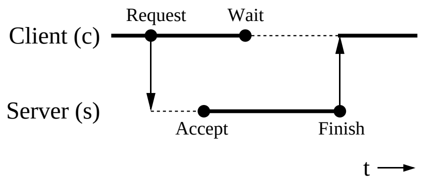


Fig. 1 Interaction between client and server process.

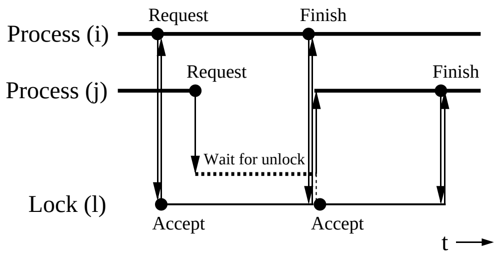


Fig. 2 Mutual exclusion between processes by means of a lock.

2. Event Abstraction

This section describes how we define events to be measured. Those events are specified at an abstract level so that we can define events independently of the OS of the target system.

2.1 System Model

The target system is abstracted by introducing the concept of the object, which is any entity required by the target system for job execution. Target system behavior can be abstracted as the execution of jobs by objects and the interactions between objects.

Objects include physical resources (CPUs, disks, etc.), logical entities (processes, functions in the programs, etc.), logical resources (locks, etc.), and external entities (users, client machines, server machines, etc.).

Interactions between objects include communications and conflicts. Inter-process communication between client and server[18], shown in **Fig. 1**, is a representative example of the communications. The mutual exclusion between processes, achieved by means of a lock (see **Fig. 2**), is a representative example of the resolution of a conflict.

2.2 Abstract Events

In my proposed framework, events are abstracted from actual events, an "actual event" being any kind of change in a system state[4]. There are two categories of abstraction: abstraction of interactions between non-CPU objects and abstraction of CPU-related events.

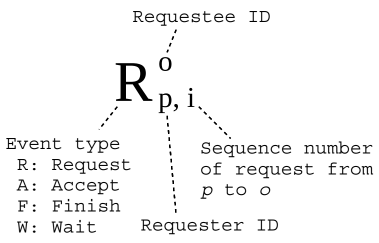


Fig. 3 Notation for interaction events.

2.2.1 Abstraction of Interactions

Interactions include the following abstract events: a request, an acceptance, a finish, and, possibly, a wait event (see Figs. 1 and 2). Hereafter, I will use the notation shown in **Fig. 3** to describe these four events.

The following is a brief description of abstract events. In it, "active object" represents the physical resources and the logical entities and "passive objects" represents the logical resources.

Request An active object, referred to as the requester, either gives a request for a job execution to another active object or requests approval from a passive object for the use of a critical resource managed by that passive object. The object receiving the request is referred to as the requestee. Requests can only be made when the requester, usually a client process, is using a CPU resource for its operation.

Accept For an active requestee object, "accept" means it begins to processes the requested job. For a passive requestee object, it means it approves the request and changes its internal state to disapprove further requests from any other requester.

Finish For an active requestee object, this means it finishes processing a job and notifies the requester of the completion. For a passive requestee object, it means that it is notified by the requester of the finish of a job for which a critical resource has been used; it then changes its internal state so as to approve the next incoming request for that resource.

Wait When a requester needs to know the outcome of a job performed by an active object requestee, it pauses in its operations and waits to be notified. When such notification is unnecessary, there will be no wait.

With these abstract event definitions, we can describe the interactions between objects. **Ta-**

Table 1 Interactions between objects.

Requestee	Logical Entities			Physical Resources		Logical Resources	
Initiate of Accept and Finish	Requestee			Requestee		Requester	
Cue for Accept	Timer	On request		On request		On request	
Requester Behavior (Request - Accept)	–			–		Busy wait	Pause
Requester Behavior (Wait - Finish)	(No wait)		Pause	(No wait)	Pause	(No wait)	
Example	File system flush daemon	Detached thread	Client/Server Fork/Join Func. call	I/O (Async.)	I/O (Sync.)	Spin lock	Resource contention

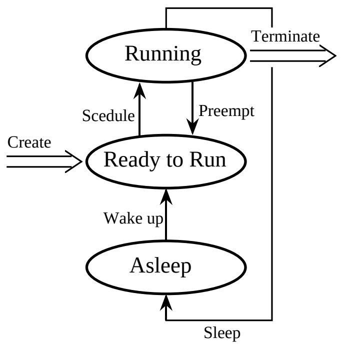


Fig. 4 Abstract events related to CPU.

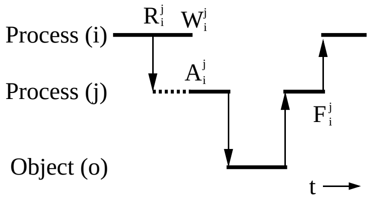


Fig. 5 Time chart showing object utilization: request R_i^j is pending from R_i^j to A_i^j, it is processed by process j from A_i^j to F_i^j, during which time process j uses object o.

ble 1 shows examples of interactions between objects located in a target system.

2.2.2 CPU-related Events

CPU-related events are mainly interactions between the CPU and processes in the target system. They are abstracted on the basis of the process state-transition diagram shown in **Fig. 4**; each abstract event corresponds to an arrow in the figure. Some CPU-related events are abstract events describing the interactions between the CPU and the process object; the wake up, resume, and sleep events correspond, respectively, to the request, accept, and finish events.

The diagram in Fig. 4 is a simplified and modified version of the process state-transition diagram for a UNIX process$^{2)}$, but it can be applied to most operating systems.

3. Formulation of Trace Analysis

With this event-trace framework, it is possible to generate performance measures on a per-transaction basis, which means that such performance measures as response time and resource usage can be obtained for each transaction. Such per-transaction-basis results are quite useful to system engineers.

In this section, I define performance measures by using abstract events, and I describe the algorithms used to obtain them in event-trace analysis.

3.1 Performance Visualization

A time chart can be used to display system behavior; **Fig. 5** shows an example of such a chart. The horizontal lines show object utilization, making it possible to visualize interactions between objects. It basically shows how the CPU is being used by processes; it can also show the utilization of objects other than the CPU.

Time charts like that of Fig. 5 are useful for comprehending target-system behavior intuitively.

3.2 Transaction

In my framework, I define the term "transaction" as a request coming from outside the target system, usually from a user or client machine. More than one process can be involved in the transaction processing; an example is shown in **Fig. 6**.

Each transaction can be divided into subtransactions whose boundaries can be defined arbitrarily; inter-process communications and function calls, which correspond to the abstract

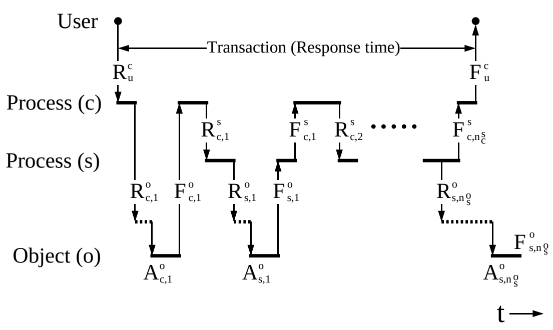


Fig. 6 An example of transaction processing. Transaction processed by process c begins at request event R^c_u and ends at finish event F^c_u.

event of request and finish, are usually used to mark those boundaries.

3.3 Performance Measures

In this section, I formulate various performance measures and the algorithms used to generate them, using the notation shown in Fig. 3 to express the abstract events, and function $t(e)$ to express the time when event e occurred.

3.3.1 Object Utilization

To determine object utilization☆ (i.e., the amount of time an object is used during transaction processing), we must take into account inter-process communications. The algorithm I have developed for doing this analyzes the event trace. (It also can be used for analyzing object-usage time for each sub-transaction.) The algorithm is as follows.

(1) Prepare a time accumulator (U^o_u) whose initial value is zero, a process set (W), a request set (E), and a temporary variable (τ^o), where o is the target-object ID.

(2) Read one record from the event trace.

(3) If the event is A^{Pc}_u, the beginning of a transaction by requestee process p_c, put p_c into W.

(4) If the event is F^{Pc}_u, the finish of transaction processing by p_c, remove p_c from W.

(5) If the event is R^{Ps}_{Pc}, a request from process p_c in W to another process (p_s), put p_s into W.

(6) If the event is F^{Ps}_{Pc}, the finish of transaction processing by process p_s, remove p_s from W.

(7) If the event is $R^o_{p,i}$, a request for target resource o by process (p) in W, put $R^o_{p,i}$ into E.

(8) If the event is $A^o_{p,i}$, the beginning of target-resource utilization corresponding to $R^o_{p,i}$ in E, set τ^o to $t(A^o_{p,i})$, the occurrence time of this event.

(9) If the event is $F^o_{p,i}$, the finish of the target-resource utilization corresponding to $R^o_{p,i}$ in E, add $(t(F^o_{p,i}) - \tau^o)$ to U^o_u, where $t(F^o_{p,i})$ is the occurrence time of this event.

(10) If R and E are both empty, output U^o_u as the object o utilization time for transaction u; else go to step 2.

E is used for the case in which acceptance $A^o_{p,i}$ occurs after the finish of transaction F^{Pc}_u; at that time, process set W must be empty. This case can occur if request $R^o_{p,i}$ was processed asynchronously.

Object utilization for each process, without considering the inter-process communication, is obtained using

$$U^o_p = \sum_{i=1}^{n_p} (t(F^o_{p,i}) - t(A^o_{p,i})), \tag{1}$$

where o is the object ID, p is the process ID, and n_p is the number of times that process p used object o.

3.3.2 Response and Throughput

The response time of a transaction is obtained by subtracting $t(R_u)$ from $t(F_u)$, where R_u is the beginning of the transaction and F_u is the finish. The average response time is $\sum_{i=1}^{n_u}(t(F_{u,i})-t(R_{u,i}))/n_u$, where $R_{u,i}$ and $F_{u,i}$ are the time of the request and of the finish of the ith user request, and n_u is the number of transactions processed during the measurement period.

Throughput is n_u/T, where T is the length of the measurement period.

3.3.3 Recent Measures

Ji, et al.[8)] have recently described two new performance measures, process waiting time and semi-busy-waiting, and used them to evaluate the performance and tuning of multithreaded programs.

My framework can be used to derive these measures, in addition to deriving the traditional performance measures described above.

Process waiting time is defined as the time during which a process is waiting for a CPU resource to become available. Formulation using

☆ Since objects in this framework include computational resources, such as CPUs and disks, object utilization may be thought of as an extension of resource utilization.

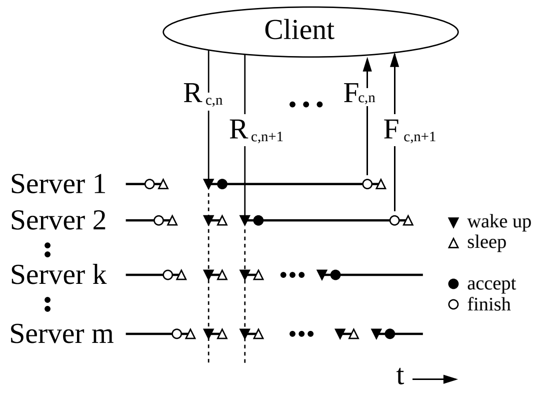


Fig. 7 An example of semi-busy-waiting. At $R_{c,n}$, Servers 1 to m were woken up, and Server 1 processed request $R_{c,n}$. Servers 2 to m immediately went back to sleep.

abstract events can expand this definition to the time during which process p is waiting for any object o; expanded process waiting time Q_p^o is given by

$$Q_p^o = \sum_{i=1}^{n_p}(t(A_{p,i}^o) - t(R_{p,i}^o)), \qquad (2)$$

where the meaning of each symbol is the same as that in the Eq. (1).

Semi-busy-waiting is defined as the number of times that a process was unnecessarily woken up. An example is shown in **Fig. 7**. Servers 2 to m were unnecessarily woken up at $R_{c,n}$; they immediately went back to sleep. Server 1 was woken up and processed request $R_{c,n}$. Semi-busy-waiting for Server k is the number of wake-ups between $F_{c,i}^k$ and $A_{c,i+1}^k$ minus one (i.e., minus the number of necessary wake-ups).

Unnecessary wake-ups often occur in systems in which the server processes are organized on the basis of a team model[19]. In the Linux community, they are referred to as the "thundering herd problem"[3] or the "stampede effect"[11].

4. Application

In this section I describe a case study in which my framework was used to evaluate the performance of a web server that uses the Apache[1] HTTP server on a Linux[9] operating system. In it, I investigated the change in behavior of the SPECweb96 benchmark program[15] with the change of the execution environment including Linux version (2.2.14 and 2.3.41)☆, acceptance policy used in the HTTP servers, and the number of the HTTP-server and client processes.

I mainly focus on the effect of replacing the global kernel lock, considered to be a weak point in Linux 2.2.x[13], with finer granularity locks[11], and of the "wake one" policy with which Linux wakes up only one process among those processes sleeping at the same socket[11], both introduced in the 2.3.x kernel.

4.1 Outline

4.1.1 Target System and Measurement Tool

The configuration of the target system and the measurement tool is shown in **Fig. 8**. To prevent the client machine from becoming a bottleneck, I used the client machine with a higher-performance CPU than that on the server machine. I expected that any bottlenecks would occur in the CPUs on the server machine.

Measurements, that is event tracing in this case study, were conducted using a hybrid monitor[7] which consisted of software probes and an event tracer. The CPU overhead created by this tool was about 2%, estimated by comparing the SPECweb96 throughput of a probed system with that of a non-probed system. It is small enough to ensure measurement accuracy☆☆.

The probes were inserted into the Linux kernel and the Apache server by modifying their source code. The kernel probes are designed to detect CPU-related events (Section 2.2.2) and object interactions, including disk accesses, network accesses, and lock activities. The probes for the Apache server will be described in Section 4.2.1.

4.1.2 Apache Configurations

The following options were applied to the Apache HTTP server used in the experiments.

Acceptance policy The default configuration of the Apache server uses serialized acceptance, in which the accept() system calls made by the HTTPd processes are serialized by means of a flock() system call. The Linux OS, however, allows more than one process to use accept() system calls at the

☆ These are the latest versions, as of the beginning of February, 2000, of the stable kernel and of the development kernel, respectively.

☆☆ With regard to the relationship between measurement accuracy and tool overhead, McKerrow has noted that, if the software tool uses less than 5 % of system resources, measurement accuracy is generally adequate[10].

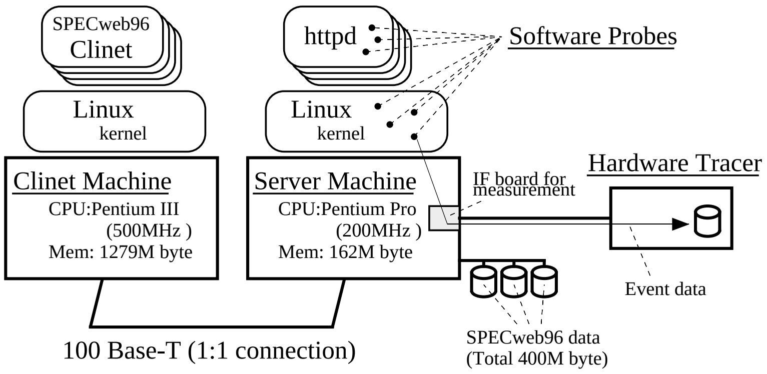


Fig. 8 Event traces were captured from the server machine with hybrid tracer consisted of the software probes and the hardware tracer.

Table 2 Parameters and results of experiments.

Case number	1	2	3	4	5	6	7	8
Linux version	2.2.14				2.3.41			
Acceptance policy	Serialized		Parallel		Serialized		Parallel	
HTTPd processes	5	50	5	50	5	50	5	50
Client processes	8	50	8	50	8	50	8	50
Throughput [ops/s]	212	229	72	95	273	268	280	274
Response time [ms]	37.6	218.1	111.6	527.2	29.2	182.9	28.5	179.1

same time (parallel acceptance)[☆]. I thus tested two cases: serialized acceptance and parallel acceptance.

Number of HTTPd processes In the default configurations of the Apache server and SPECweb96, the number of initial HTTPd process and client process were, respectively, 5 and 8. Both defaults were used in the experiments. In addition, I also tested the case in which the number of HTTPd processes was 50 and that of the client processes was 50.

4.1.3 Benchmark Results

The parameters and results of the testing are summarized in **Table 2**.

Both the throughput and response time with version 2.3.41 of Linux were much better than those with version 2.2.14. In particular, the performance of 2.3.41 with parallel acceptance was much better.

From these benchmark results, we can infer that version 2.3.41 consumes less CPU power per SPECweb96 HTTP operation than does 2.2.14. The wake-one policy and the reduced global-kernel-lock usage, introduced in 2.3.41, apparently have a beneficial effect.

☆ Serialization is necessary because some UNIX operating systems do not guarantee correct operation if more than one process use accept() system calls at the same time.

4.2 Analysis Plan

I planed the performance evaluation conducted with the following steps: 1) capture an event trace from the Apache-HTTP server that includes application probes to divide its executions into five phases, 2) analyze the event trace and produce such performance metrics as usage of CPUs, Disks, and global lock for each phase, and 3) compare the result with those obtained from other cases. The measurement and the trace analysis were designed according to this plan.

4.2.1 Application Probes

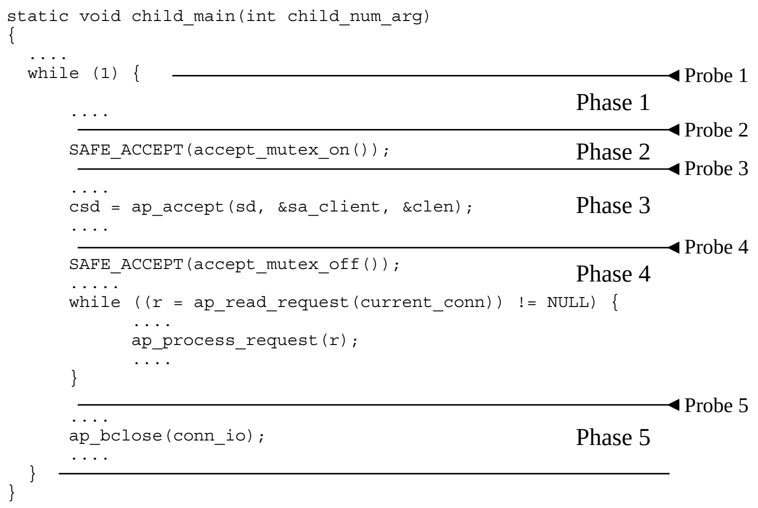
The application probe used in the measurement was inserted at the beginning of each phase (see **Fig. 9**) so that the boundary information could be used during trace analysis.

4.2.2 Abstract Model

The performance analysis was conducted with using a hierarchical model that describes the Apache behavior on the basis of my framework (see **Fig. 10**).

The interactions between server process and execution phases could be treated as a kind of procedure calls; the request and the accept event of a phase and the finish event of the previous phase occurred at the same time and they were marked by the application probe inserted at the beginning of the phase.

The performance metrics for each execution



```
static void child_main(int child_num_arg)
{
  ....
  while (1) {
      ....
      SAFE_ACCEPT(accept_mutex_on());
      ....
      csd = ap_accept(sd, &sa_client, &clen);
      ....
      SAFE_ACCEPT(accept_mutex_off());
      .....
      while ((r = ap_read_request(current_conn)) != NULL) {
            ....
            ap_process_request(r);
            ....
      }
      ....
      ap_bclose(conn_io);
      ....
  }
}
```

Fig. 9 Main loop in Apache HTTP server; the execution of the processes was divided into five phases: 1) preparing the HTTP operation, 2) waiting for the lock for accept() usable, 3) executing the accept() system call, 4) processing the HTTP request made by the client, and 5) closing the TCP/IP connection corresponds to this HTTP operation.

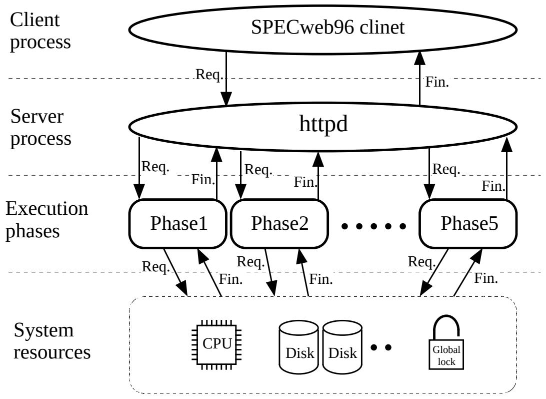


Fig. 10 Apache execution was modeled with objects, forming a hierarchical structure, and interactions between objects in neighboring layers in the hierarchy.

phase were obtained by using the algorithm described in Section 3.3.

4.3 Analysis Results

Performance analysis was conducted using 20-sec event traces obtained from the target machine for each case shown in Table 2. Each event trace started 10 minutes after the benchmark program started.

The results in this section are shown on a per-HTTP-operation basis. That is, the obtained measures were divided by the corresponding number of HTTP operations (**Table 3**).

4.3.1 Object Usage

The amount of object usage-time in processing one HTTP operation☆ is shown in **Table 4** (the two CPUs, three disks, and kernel lock were here considered objects).

The CPU usage for 2.2.14, in rough outline, was almost the same as that for 2.3.41; the average for Cases 1 to 4 was 7.03 ms and that for Cases 5 to 8 was 7.22 ms. Thus, the SPECweb96 throughput results cannot be explained simply on the basis of the CPU usage results. It suggests that Linux 2.2.14 had a performance bottleneck other than physical resources (e.g., CPUs and disks).

The difference due to the change in the number of servers and of clients was smaller for version 2.3.41; the CPU usage for Cases 1 and 3 was smaller than that for Cases 5 and 7 (8 client processes), and that for Cases 2 and 4 was greater than that for Cases 6 and 8 (50 client processes). Linux 2.3.41 should thus perform more effectively in larger systems, which usually have a larger number of HTTPd processes and serve more clients.

The degree of kernel-lock usage in 2.3.41 was about one third of that in 2.2.14, that resulted in the difference in the busy-wait time due to spin locks; the results of the busy-wait time in 2.3.41 were one third to ninth of that in 2.2.14.

☆ The object usage by processes other than HTTPd and kflushd processes was minimal; I thus take it that all objects were dedicated to processing HTTP requests. (A kflushd process wrote mainly the HTTP-access log into the log file.)

Table 3 Number of HTTP operations by case.

Case number	1	2	3	4	5	6	7	8
No. of ops	4442	4411	1290	2173	5559	5406	5619	5563

Table 4 Object usage-time in ms.

Case number	1	2	3	4	5	6	7	8
Process user	1.25	1.70	1.40	1.68	1.83	1.82	1.77	1.77
Process kernel	1.81	3.20	1.57	2.57	2.75	2.9	2.73	2.80
Disk Interrupt	0.02	0.03	0.03	0.03	0.03	0.05	0.04	0.04
Network Interrupt	1.76	1.99	1.78	2.03	2.34	2.37	2.37	2.35
Clock interrupt	0.04	0.05	0.06	0.05	0.04	0.04	0.03	0.04
Busy Wait	0.51	1.81	1.09	1.66	0.19	0.21	0.17	0.19
Total CPU	5.40	8.78	5.92	8.03	7.18	7.39	7.11	7.18
Disk 1	0.34	0.07	0.10	0.08	0.75	0.46	0.51	0.40
Disk 2	0.22	0.38	0.62	0.54	0.55	0.68	0.57	0.31
Disk 3	0.67	0.50	0.52	0.58	0.42	0.77	0.43	0.73
Kernel lock	2.00	2.82	2.19	2.60	0.82	0.78	0.68	0.68

Table 5 Phase details for Cases 1 and 5 in ms.

Case number	1					5				
Phase number	1	2	3	4	5	1	2	3	4	5
Elapsed time	0.20	34.61	4.03	4.27	1.93	0.95	34.00	0.67	21.96	54.51
Process	0.08	0.20	0.17	2.33	0.29	0.14	0.11	0.14	3.64	0.52
Busy-wait for kernel lock	0.02	0.06	0.05	0.27	0.10	0.01	0.01	0.00	0.06	0.06
Busy-wait for other lock	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.02
Total CPU	0.10	0.25	0.21	2.60	0.39	0.15	0.12	0.15	3.71	0.60
Disks 1-3	0.00	0.00	0.00	0.86	0.00	0.00	0.00	0.00	0.99	0.00
Wait for CPU	0.10	4.66	0.21	0.53	1.28	0.8	30.62	0.28	17.07	53.86
Wait for Disk	0.00	0.00	0.00	0.23	0.00	0.00	0.00	0.00	0.17	0.00
Wait for Network	0.00	0.00	3.6	0.04	0.26	0.00	0.00	0.24	0.01	0.05
Conflict with other process	0.00	29.69	0.00	0.00	0.00	0.00	3.26	0.00	0.00	0.00

The influence of the busy-wait time was serious in 2.2.14 because it accounted for 9% to 21% of the CPU power. On the other hand, the busy-wait time in 2.3.41 accounted for about 3% of the CPU power, the effect of the busy-wait would be thus small as long as we use Linux 2.3.41 on a system that have two CPUs.

The CPU usage for network-interrupt handling was about 30% of the total CPU power. Performance tuning of the protocol-stack processing might perhaps improve this.

4.3.2 Phase Detail

The elapsed times by phase (see Fig. 9) are shown in **Table 5** for Cases 1 and 5.

The elapsed times for Phases 4 and 5, Case 1 (version 2.2.14) were far shorter than those of Case 5 (2.3.41), mainly because of the difference in the "Wait for CPU" time. This contradicts my expectation that the the average response times shown in Table 2 would correlate closely with the elapsed times of Phase 4, in which the HTTP requests from clients are processed. The elapsed times of Phases 1 and 2 are not so important because these phases have no connection with the client machine.

The difference in the elapsed times of Phases 3 resulted from the difference in the "Wait for Network" time. It would be a clue to the performance bottleneck in Linux 2.2.14 (Section 4.3.1).

4.3.3 Semi-busy-waiting

A thundering herd problem occurred in Phase 2 for the HTTP servers with serialized acceptance and in Phase 3 for those with parallel acceptance. The number of wake ups in these phases was related to the severity of the thundering herd problem. **Table 6** shows the number of wake ups in Phase 2 for Case 1, 2, 5, and 6 or in Phase 3 for Case 3, 4, 7, and 8.

The number of wake ups in version 2.3.41 was less than that in 2.2.14. This is as might be expected from the fact that Linux 2.3.41 incorporates a wake-one policy.

This smaller number of wake ups occurred not only for the HTTP servers using parallel acceptance but also for those using serialized acceptance, where the wake-one policy for accept() is not effective. Linux 2.3.41 possibly has a process-scheduling policy that differs from that of 2.2.14.

Table 6 Wake up in phase 2 or 3.

Case number	1	2	3	4	5	6	7	8
Wake ups	7.56	10.96	4.80	10.95	0.82	0.10	0.77	0.28

Table 7 Wake ups for HTTPd processes and total number of process switches (including those of processes other than HTTPds).

Case number	1	2	3	4	5	6	7	8
Wake ups: Disk	0.06	0.08	0.06	0.08	0.06	0.08	0.07	0.08
Wake ups: Network	4.85	0.82	5.73	10.06	0.61	0.10	1.21	0.41
Wake ups: Other processes	7.56	10.96	0.83	1.18	0.82	0.11	0.03	0.01
Process switches	12.76	12.77	7.08	12.18	1.95	0.98	1.95	1.21

4.3.4 Process Switches

I have also compared the number of wake ups for HTTPd processes with the number of process switches. As shown in **Table 7**, the number of process switches with version 2.3.41 was much smaller than that with 2.2.14.

4.4 Characteristics of Linux 2.3.41

By comparing the performance of SPECweb96 benchmark program on Linux 2.2.14 with that on Linux 2.3.41, I have identified the following characteristics of Apache servers running on Linux 2.3.41.

(1) Their degree of the global-kernel-lock usage is smaller.
(2) The change in the CPU usage due to the change in the number of servers and of clients is smaller.
(3) The CPU usage is larger for Cases 5 and 7 (where the number of HTTPd processes is 5).
(4) The elapsed times of Phases 4 and 5 are larger.
(5) The number of process switches is smaller.

Observations 1 and 2 are consistent with prior expectations, but not Observations 3 and 4. The origin of the improvement in SPECweb96 throughput on version 2.3.41 is not a reduction in CPU usage; Linux 2.2.14 had a performance bottleneck other than the CPU resource, and it was solved in 2.3.41. It is possible that the bottleneck was connected with the change in the process-scheduling policy (Observation 5).

The cause of the reverse relationship between the SPECweb96 response time and the elapsed times of Phases 4 and 5 is not clear. Experiments that trace the correspondence between received packets and server processes, as well as that between server processes and sent packets, are needed in order to determine the cause. The proposed framework will be useful in designing those behavior measurements by simply extend the communication events to those connected with communications between client and server machines.

5. Discussion

5.1 Relation to Previous Work

5.1.1 Previous Formulation

The formulation of event-trace described by McKerrow[10] defines performance measurement as ascertaining the extent of an object or of the set of modules. It uses the following definitions:

Object The particular entity whose performance is to be measured.

Object hierarchy An object at a specific hierarchical level can be decomposed into modules that are objects lower in the hierarchy. The hierarchy extends from the micro-code level, via the function level, to the process and system levels.

Object execution An ordered sequence of module executions.

Event The termination of one module in the execution sequence and the start of the next module.

This formulation is useful for measurements that focus on the internal behavior of processes, i.e., the execution of program instructions that is not affected by the execution of other processes.

Computer systems now commonly use distributed objects or parallel objects. Therefore measurement of their performance has to take into account the interaction between processes in addition to process-internal behavior. An example of this is the performance evaluation and tuning of multithreaded programs[8] in which process waiting time and semi-busy-waiting were introduced.

5.1.2 Advance by this Work

The framework I have described has been designed to include interactions between processes as well as process-internal behavior. For this purpose, I have created abstractions of the system, its behavior, and events within it.

This framework abstracts target systems by introducing objects that can be any kind of entity with a connection to job execution in the target system. This definition is an expansion of that by McKerrow, in which logical entities alone are considered to be objects.

This framework abstracts events as interactions between objects. These events include interactions across the object hierarchy, the same as a McKerrow event, and interactions between objects with no hierarchical relationship. This definition of event is thus broader than that by McKerrow.

Because events are abstracted, there is no distinction between them based on the properties of the objects involved in the interaction; Table 1 shows some actual object interactions that can be treated by this framework. In the case study described in Section 4, the same algorithm based on this framework could be used for obtaining the results for the whole HTTPd execution and those for its each execution phases.

Since this framework formulates performance measures by using abstract events, we can define and obtain performance measures for any object. We can formulate not only traditional performance measures such as throughput, response time, and resource utilization, but also such recent performance measures as process waiting time and semi-busy-waiting. Process waiting time was originally defined as the time during which a process is waiting for the CPU resource to become available. In this framework, that definition has been expanded to be the time during which a process is waiting for any resource to become available. The expanded process-waiting times were presented in Section 4.3.2.

5.2 Application to Performance Estimation

Performance estimation with such performance models as queueing-network and simulation models requires input data that describe the characteristics of the target system's behavior.

Smith tidied up the input data in her SPE (system performance engineering) method[14], in which she proposed that software execution needs to be divided into sub-executions and resource-usage data needs to be obtained for each component. These sub-executions correspond to the subtransaction and execution phases in this paper.

As described in Section 4.3.2, by using this framework, we can obtain performance measures for each execution phase. It is thus suitable for use in performance estimation methods, including the SPE method.

6. Conclusions

Event-driven measurement tools have traditionally measured different sets of software events and have not been standardized, though event tracing is useful for performance estimation and bottleneck detection. My motivation in developing a event-trace framework was to provide a guiding principle for unifying the disparate approaches that have so far characterized event-based performance evaluation.

The framework consists of 1) abstractions representing a system, its behavior, and the events to be measured and 2) formulations of performance measures and of algorithms for obtaining them from an event trace. It enables the interactions between processes to be treated together with process-internal behavior.

I have applied the framework to the performance evaluation of an Apache HTTP server running on Linux 2.2.14 or 2.3.41 and executing the SPECweb96 benchmark program. The CPU overhead created by the hybrid monitor used in the measurement was about 2% so that I could capture software events concerned with such detail target system behavior as global-kernel-lock usage with little disturbance on the target system behavior. A low-overhead measurement tool is a key to apply the framework to actual systems.

By comparing the performance of the benchmark program on Linux 2.2.14 with that on Linux 2.3.41, I was able to clarify the differences between the two versions of Linux. Some observations, including those of global-kernel-lock usage and robustness for the increase in the number of clients, are consistent with prior expectations, but others are not; in particular, the difference in CPU usage did not explain the difference in SPECweb96 performance, and thus it suggested that Linux 2.2.14 had a performance bottleneck other than physical resources (e.g., CPUs and disks).

This case study indicates that the measurement of CPU usage alone is not enough. We also need to consider process interactions, at least so far as Apache HTTP servers are concerned. This framework includes the treatment of the process interactions, easily extendable to communications between client and server machines, it is thus useful for performance evaluation of systems whose performance is influenced by process interactions as well as by process-internal behavior.

Acknowledgments The author would like to thank Akihiro Kaieda for useful information about Linux operating system. The author would also like to thank the other members of the performance evaluation group for valuable discussion.

References

1) Apache Project: http://www.apache.org/httpd.html.
2) Bach, M.J.: *The Design of The UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).
3) Ezolt, P.: Overscheduling DOES happen with high web server load (1999). http://www.linuxhq.com/lnxlists/linux-kernel/lk_9905_01/msg00655.html, and its follow-up articles.
4) Ferrari, D., Serazzi, G. and Zeinger, A.: *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ (1983).
5) Ganger, G.R. and Patt, Y.N.: The Process-Flow Model: Examining I/O Performance from the System's Point of View, *SIGMETRICS '93 Proceedings*, pp.86–97, ACM (1993).
6) Haigh, P.: An Event Tracing Method for UNIX Performance Measurement, *CMG '90 Proceedings*, pp.603–609, CMG (1990).
7) Horikawa, T.: TinyTOPAZ: A Hybrid Event Tracer for Unix Servers, *SPECTS '99 Proceedings*, Chicago, IL, pp.203–210, SCS (1999).
8) Ji, M., Felten, E.W. and Li, K.: Performance Measurements for Multithreaded Programs, *SIGMETRICS '98 Proceedings*, pp.161–170, ACM (1998).
9) Linux.com site: http://www.linux.com.
10) McKerrow, P.: *Performance Measurement of Computer Systems*, Addison Wesley, Sydney (1987).
11) Pranevich, J.: Wonderful World of Linux 2.4 (1999). http://linuxtoday.com/stories/10698.html.
12) Rose, C.A.: A Measurement Procedure for Queueing Network Models of Computer Sytems, *Computing Surveys*, Vol.10, No.3, pp.263–280 (1978).
13) Russinovich, M.: Linux and the Enterprise, *Windows NT Magazine*, Vol.5, No.4, pp.93–99 (1999).
14) Smith, C.U.: *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA (1990).
15) Standard Performance Evaluation Corporation: SPECweb96. http://www. specbench.org/osg/web96.
16) Sun Microsystems: Tracing Program Execution With the TNF Utilities, Programming Utilities Guide, Chapter 1 (1998).
17) Tamches, A. and Miller, B.P.: Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, *Symposium on Operating Systems '99 Proceedings*, USENIX, pp.117–130 (1999).
18) Tanenbaum, A.S.: *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1992).
19) Tanenbaum, A.S.: *Distributed Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1995).

(Received March 3, 2000)
(Accepted November 2, 2000)

Takashi Horikawa received his BE from Doshisha University, Kyoto, Japan, in 1981, and his ME from Kyoto University, Kyoto, Japan, in 1983. He is currently a Manager, Performance Engineering, at Development Laboratories, NEC Networks. His research interests include performance evaluation of computer & communication systems, distributed systems, and operating systems.