OS/Application Concurrency: A Model

James W. Wendorf

April 1987

CMU-CS-87-153

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A model of the processing performed on a computer system is presented. The model divides processing into two types: OS processing and application processing. It then defines what it means to have OS/application concurrency, and enumerates the different forms such concurrency can take. Examples are presented to illustrate the model's analytic and predictive capabilities. The model provides a common framework for describing the concurrency in different systems, and it aids in identifying the areas where increased concurrency may be possible. The potential performance improvements resulting from increased OS/application concurrency can also be predicted from the model.

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274, and the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of RADC, NOSC, or the U.S. Government.

Table of Contents

List of Figures

1. Introduction

A common approach to providing hardware support for operating system functions is to add specialized processors to a computer system. This allows the supported functions to execute concurrently with other system activities, especially with application processing. As part of our effort to analyze and compare such OS support techniques [7], we have developed a model of OS/application concurrency. This model provides a uniform and precise terminology for describing the processing performed on a computer system, and in particular the relationship between OS processing and application processing. The model defines OS/application concurrency, and enumerates the different forms such concurrency can take.

This paper presents the OS/application concurrency model, and discusses its interpretation in terms of OS and application processing on typical computer systems. Specific examples illustrate the model's analytic and predictive capabilities when applied to actual system environments. The model provides a common framework for describing the different examples, and it indicates the areas where OS/application concurrency can be increased through appropriate system modifications. The model is also used to predict the potential performance improvements resulting from the increased concurrency.

2. Model and Interpretation

In this section we define the OS/application concurrency model, and provide precise definitions for the different forms of concurrency. A general interpretation is provided along with the definitions, indicating how the model applies to typical computer systems. Additional notes clarify certain aspects of the definitions and interpretation, particularly as they relate to modeling real systems.

2.1. Elements of the Model

The OS/application concurrency model has four elements: load, processes, statements, and "works-for" relation. The load on a computer system is a set of processes, where each process is a sequence of statements. Some processes work-for other processes. Without loss of generality, the model assumes a single global clock, and each statement executes during a specified interval in the global time frame. Once a computer system has been modeled in terms of these four elements, the concurrency within the system can be described precisely.

2.1.1. Definitions

Load: A load is a set of processes $\{P_1, \ldots, P_i, \ldots\}$, with a Boolean-valued function "Works-For" defined on all pairs of processes in the set.

Process: Every process P_i has a type $ProcType(P_i) \in \{\text{APP}, \text{OSP}, \text{GMP}\}$, and a statement sequence $\{S_1(P_i), \ldots, S_k(P_i), \ldots\}$.

"Works-For" ($\rightarrow$): For every process P_i with $ProcType(P_i) = \text{OSP}$, there is a single process P_j such that $ProcType(P_j) = \text{APP}$ and $P_i \rightarrow P_j = \text{TRUE}$ (denoted simply $P_i \rightarrow P_j$). In all other cases $P_i \rightarrow P_j = \text{FALSE}$ (denoted $P_i \not\rightarrow P_j$).

Statement: Every statement $S_k(P_i)$ has a type $StmtType(S_k(P_i)) \in \{\text{AP}, \text{OS}\}$, and an integer $StartTime(S_k(P_i))$, and an integer $EndTime(S_k(P_i))$, where:

$$StmtType(S_k(P_i)) = \text{OS} \text{ if } ProcType(P_i) = \text{OSP or GMP, and}$$
$$StartTime(S_k(P_i)) < EndTime(S_k(P_i)) \leq StartTime(S_{k+1}(P_i)).$$

2.1.2. Interpretation and Notes

The load on a computer system is modeled as a set of processes. That set can either be finite or unbounded in size. Each process is a sequence of serially executed statements, where the sequence can either be finite or unbounded in length. Each statement is a period of uninterrupted execution on a processor. Statements are classified as either OS or application processing (OS or AP). There are three different types of processes:

1. An application process (APP) is the normal type of process that a user provides as load to the system. An APP process can include both application processing (AP statements) and OS processing (OS statements).
2. An OS process (OSP) does only OS processing, and the work it performs is on behalf of a particular APP process.
3. A global management process (GMP) also does only OS processing, but the work it performs is for the global benefit of the system, rather than on behalf of a particular APP process.

Notes:

1. Usually the load being modeled will be a finite set of processes, which execute on the system during a particular period of time. However, the load can be unbounded if a continuously running system is to be modeled. The sequence of statements in a continuously running process would then be unbounded as well.
2. Statements may be very fine grain (such as individual machine instructions) or higher level operations (such as entire subroutines), depending on what is most convenient for the system being modeled. Time gaps between successive statements in a process indicate periods of inactivity, when the process is waiting for external events or for an available processor.
3. Transitions between AP and OS statements within an APP process reflect the synchronous

invocation of OS processing. Actual mechanisms for OS invocations can vary from kernel traps, to message transmissions, to object invocations. But each of these mechanisms would be modeled as an AP statement, followed by some number of OS statements. An additional OS statement may also be needed to model the return from an OS invocation back to application processing.

4. OS processing for a particular APP process can be modeled either as OS statements within the APP process, or as separate OSP processes that work-for it. Normally, synchronous OS processing is modeled as OS statements within the APP process, while OSP processes represent asynchronous (and potentially concurrent) OS processing.

5. In many real systems, a single OS server "process" can do work on behalf of multiple client application processes. However, this would be modeled using a separate OSP process for each APP process.

6. When modeling a given system, there are no hard and fast rules for determining what is OS processing and what is application processing. However, we can provide some general guidelines based on what is commonly considered to be "OS processing" by practitioners in the field. In general, OS processing is a set of services, available to all application processes in a controlled manner. These services include communication, synchronization, virtual memory management, file system, and I/O facilities. The global system management functions, which are modeled with GMP processes, include resource scheduling, file system maintenance, and accounting procedures.

2.2. Concurrency

In this model, concurrency is defined in terms of a Boolean-valued function. For any pair of statements in the load, the function will indicate whether or not they are concurrent. Different forms of concurrency can then be distinguished, based on the *StmtTypes* of concurrent statements.

2.2.1. Definitions

"Concurrent-With" ($\|$): $S_k(P_i) \| S_l(P_j) = \text{TRUE}$ if and only if
$StartTime(S_l(P_j)) \le StartTime(S_k(P_i)) < EndTime(S_l(P_j))$, or
$StartTime(S_k(P_i)) \le StartTime(S_l(P_j)) < EndTime(S_k(P_i))$.

"AP/AP-Concurrent-With" (${}^{AP}\|^{AP}$): $S_k(P_i)\, {}^{AP}\|^{AP}\, S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i) \| S_l(P_j)$ and $StmtType(S_k(P_i)) = StmtType(S_l(P_j)) = \text{AP}$.

"OS/OS-Concurrent-With" (${}^{OS}\|^{OS}$): $S_k(P_i)\, {}^{OS}\|^{OS}\, S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i) \| S_l(P_j)$ and $StmtType(S_k(P_i)) = StmtType(S_l(P_j)) = \text{OS}$.

"OS/AP-Concurrent-With" (${}^{OS}\|^{AP}$): $S_k(P_i)\, {}^{OS}\|^{AP}\, S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i) \| S_l(P_j)$ and $StmtType(S_k(P_i)) = \text{OS}$ and $StmtType(S_l(P_j)) = \text{AP}$.

2.2.2. Interpretation and Notes

A statement in one process is concurrent with a statement in another process if any part of their execution intervals overlap. Application/application (AP/AP) concurrency is when two AP statements are concurrent. OS/OS concurrency is when two OS statements are concurrent. OS/application (OS/AP) concurrency is when an OS statement is concurrent with an AP statement.

Notes:

1. Two statements within a single process cannot be concurrent, since by definition the statements of a process are executed serially.
2. The model locates areas of concurrency by specifying which statements are concurrent. Using the *StartTime* and *EndTime* of statements, it is also possible to measure the amount of concurrency (see Section 3 below).
3. "OS/AP-Concurrent-With" is an asymmetric function, but that does not restrict its ability to express all of the OS/application concurrency in a system.

2.3. OS/Application Concurrency

OS/application concurrency can be further classified, based on the types of processes involved. Three distinct classes are defined, called "strong", "weak", and "global management" concurrency.

2.3.1. Definitions

"Strong-Concurrent-With" (${}^{OS}\|_S^{AP}$): $S_k(P_i)\,{}^{OS}\|_S^{AP}\,S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i)\,{}^{OS}\|^{AP}\,S_l(P_j)$ and $P_i \rightarrow P_j$.

"Weak-Concurrent-With" (${}^{OS}\|_W^{AP}$): $S_k(P_i)\,{}^{OS}\|_W^{AP}\,S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i)\,{}^{OS}\|^{AP}\,S_l(P_j)$ and $P_i \not\rightarrow P_j$ and $ProcType(P_i) \neq \text{GMP}$.

"Global-Management-Concurrent-With" (${}^{OS}\|_G^{AP}$): $S_k(P_i)\,{}^{OS}\|_G^{AP}\,S_l(P_j) = \text{TRUE}$ if and only if
$S_k(P_i)\,{}^{OS}\|^{AP}\,S_l(P_j)$ and $ProcType(P_i) = \text{GMP}$.

2.3.2. Interpretation and Notes

Strong concurrency involves concurrent processing on behalf of a single APP process. An OS statement in an OSP process is concurrent with an AP statement in the associated APP process. Weak concurrency involves concurrent processing on behalf of different APP processes. An AP statement is concurrent with an OS statement in another APP process or non-associated OSP process. Global management concurrency is when an OS statement in a GMP process is concurrent with an AP statement.

Notes:

1. In most existing computer systems, strong concurrency is the least common form of OS/application concurrency. However, it can provide greater performance benefits than the other forms, especially in light load situations. It reduces the running time of an individual application process by overlapping its OS and application processing.

2. Strong concurrency can manifest itself either as ''post-computation'' or ''pre-computation''. Post-computation is when an APP process requests asynchronous OS processing, and then continues with application processing while its OSP process handles the request. Pre-computation is when an OSP process performs some work and buffers the results in anticipation of a future request from its APP process.

3. Weak concurrency depends on the existence of multiple application processes. It arises naturally in almost any multiprocessor system, assuming there are enough processes at a given time so that one can be doing application processing while OS processing is being done on behalf of another.

4. Global management concurrency also arises naturally in almost any multiprocessor system, assuming there are enough processes so that AP statements can be executing at the same time as the statements of an OSP process.

3. Graphical Representation and Concurrency Analysis

When modeling a system and analyzing potential performance improvements from increased concurrency, it is helpful to represent processes and their component statements graphically, as illustrated in Figure 3-1. A process is represented by a vertical time-line, with time increasing from top to bottom. Statements are intervals along the line, with heavy, solid segments representing AP statements and dotted segments representing OS statements. If processes are drawn side by side, aligned in time, the concurrency becomes evident.

Figure 3-1(a) shows strong concurrency: $S_j(Q) \, {}^{OS}\|_S^{AP} \, S_{i+2}(P)$ and $S_{j+1}(Q) \, {}^{OS}\|_S^{AP} \, S_{i+2}(P)$. The arrows between the two processes indicate time dependencies. For example, $S_j(Q)$ may represent post-computation that can only occur after the request is made in $S_{i+1}(P)$. $S_{j+1}(Q)$ may represent pre-computation that must be completed before the results can be retrieved in $S_{i+3}(P)$. Figure 3-1(b) shows weak concurrency: $S_{j+1}(Q) \, {}^{OS}\|_W^{AP} \, S_i(P)$ and $S_{i+1}(P) \, {}^{OS}\|_W^{AP} \, S_{j+2}(Q)$. Figure 3-1(c) shows global management concurrency: $S_j(Q) \, {}^{OS}\|_G^{AP} \, S_i(P)$ and $S_{j+1}(Q) \, {}^{OS}\|_G^{AP} \, S_{i+2}(P)$.

The amount of concurrency in a system, and the resulting performance improvements, can be determined from the model. As an example, Figure 3-2 illustrates the effect of adding an I/O processor to a hypothetical uniprocessor system. Assume there is an application process P that repeatedly computes

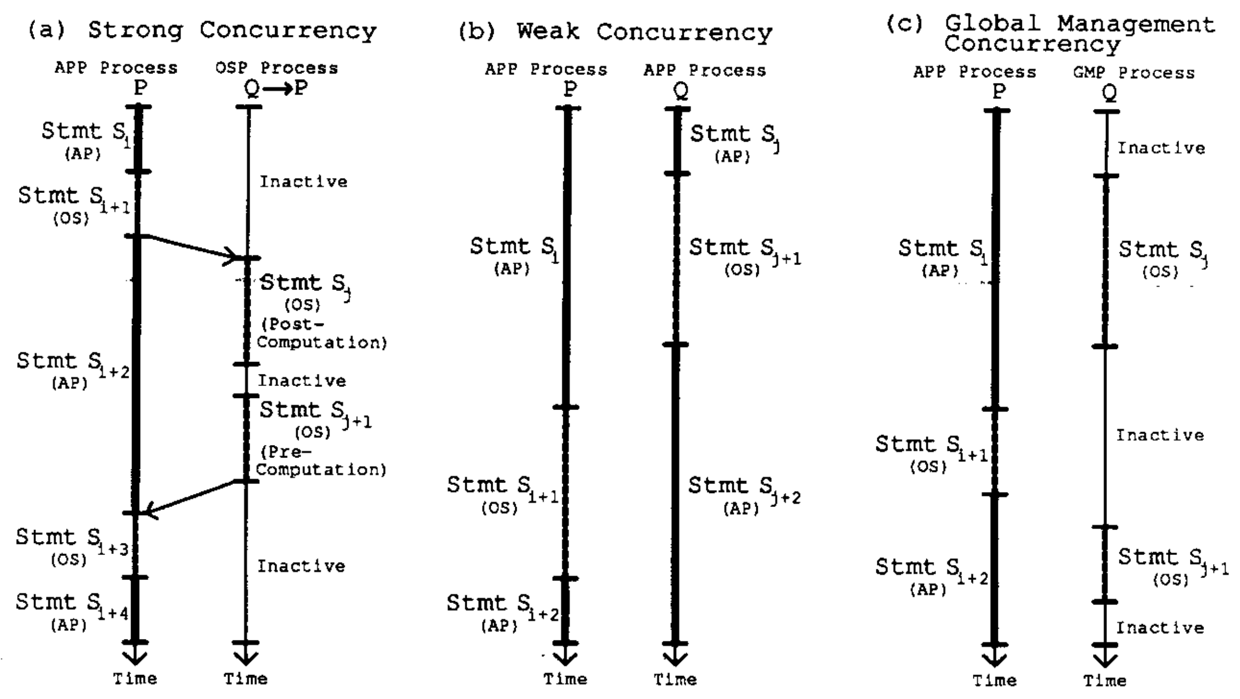

Figure 3-1: The Forms of OS/Application Concurrency

some values and writes them out. On the uniprocessor system (Figure 3-2(a)) this can be modeled as an APP process (*P*) and its associated interrupt handler process (*IH*). One compute/write cycle of the application is shown, although for convenience the cycle is taken from the beginning of one write request through the end of the following computation. The time for each statement in the cycle is given in units of "T". For example, $S_i(P)$ (Request "Write") takes 1T units of time.

After the device has started writing, it takes 3T units of time until it completes the operation and interrupts the processor. While the interrupt is being handled, *P* is inactive, waiting for the processor to become available again. Since there is only one processor in the system, there can be no concurrency. The total elapsed time for one cycle of the application is:

$$\begin{gathered} - \quad \text{Write + Compute + Handle Interrupt} \\ = (\text{Request ``Write''} + \text{Buffer Data} + \text{Start Device} + \text{Return}) + (\text{AP } S_{i+4}(P) + \text{AP } S_{i+5}(P)) + \\ (\text{Device Interrupt} + \text{Release Buffer} + \text{Return}) \\ = (1T + 2T + 2T + 1T) + (2T + 4T) + (1T + 2T + 1T) \\ = 16T \end{gathered}$$

When an I/O processor is added to the system (Figure 3-2(b)), all of the device handling can be done by that processor. This is modeled using an OSP process, *IOP*, where $IOP \rightarrow P$. Again, one cycle of the

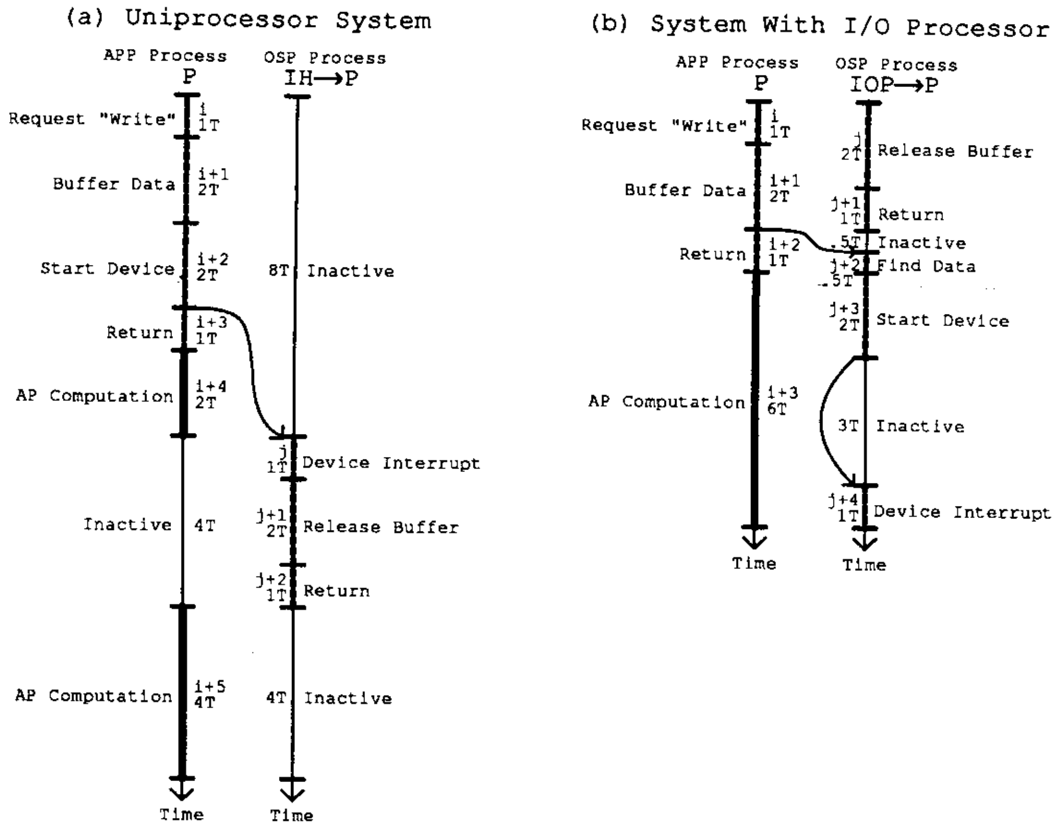

Figure 3-2: Performance Improvements From Increased Concurrency

application is shown, but note that $S_j(IOP)$ and $S_{j+1}(IOP)$ (Release Buffer, and Return) are concerned with the previous write operation. This is because the *IOP* cycle is skewed relative to *P*. Also note that a 1T delay has been introduced between Buffer Data ($S_{i+1}(P)$) and Start Device ($S_{j+3}(IOP)$). This reflects the delay involved in notifying the I/O processor and having it find the data. Since all of the statements in *IOP* are concurrent with statements in *P*, the total elapsed time for one cycle is:

$$\begin{aligned} &\text{Write} + \text{Compute} \\ &= (\text{Request ``Write''} + \text{Buffer Data} + \text{Return}) + (\text{AP Computation}) \\ &= (1\text{T} + 2\text{T} + 1\text{T}) + (6\text{T}) \\ &= 10\text{T} \end{aligned}$$

A convenient way to express the performance improvement is with the elapsed time ratio (ETR):

$$\text{ETR} = \frac{\textit{Elapsed Time on System with Added Processor}}{\textit{Elapsed Time on Original System}} = \frac{10\text{T}}{16\text{T}} = 0.625$$

This indicates that concurrency has reduced the elapsed time for the application to 62.5% of what it was originally. The savings come from making Start Device, Device Interrupt, Release Buffer, and Return (2T

+ 1T + 2T + 1T = 6T) concurrent with other statements in *P*. Half of that (3T) is strong OS/application concurrency, while the other half (3T, plus .5T for Find Data) is OS/OS concurrency.

4. Examples

We now briefly discuss a number of examples that illustrate how the OS/application concurrency model can be applied to actual system environments. The first three examples (UNIX, Accent, and Eden) demonstrate the model's expressiveness across a range of OS paradigms, from monolithic kernels, to message-based systems, to object-oriented systems. The remaining examples show how the model can be used to analyze specific instances of OS/application concurrency, and to predict the performance improvements resulting from that concurrency. Three examples are presented, covering the different forms of OS/application concurrency (strong, weak, and global management). These three examples are treated in detail and studied experimentally in [7].

4.1. Describing Different Systems

The OS/application concurrency model provides a common framework for describing the concurrency in different systems. In the following examples, three systems that appear on the surface to be very different are all modeled in a similar fashion. The purpose of these examples is to demonstrate the expressiveness of the OS/application concurrency model, rather than to analyze the concurrency and associated performace improvements. Analysis examples will follow later.

4.1.1. A Monolithic Kernel (UNIX)

The UNIX operating system [6] is an example of a monolithic OS kernel. Typical UNIX user processes would be modeled as APP processes, while the basic system processes (such as "init" and "swapper") would be GMP processes. Most OS processing in UNIX is handled synchronously with respect to the requesting user process. A user process requests OS processing by "trapping" into the kernel, and the process will only continue with application processing after the requested OS processing is completed. This type of OS processing can be modeled as a subsequence of OS statements, within the statement sequence of an APP process (see Figure 4-1(a)).

On a multiprocessor UNIX system [1], various forms of concurrency are possible. Multiple processes can be executing statements at the same time, giving application/application, OS/OS, or OS/application concurrency, depending on the types of the concurrent statements. Most OS/application concurrency would be in the form of weak or global management concurrency, as illustrated in Figure 4-1(a). However, strong concurrency is also possible, due to the buffered handling of I/O operations. For

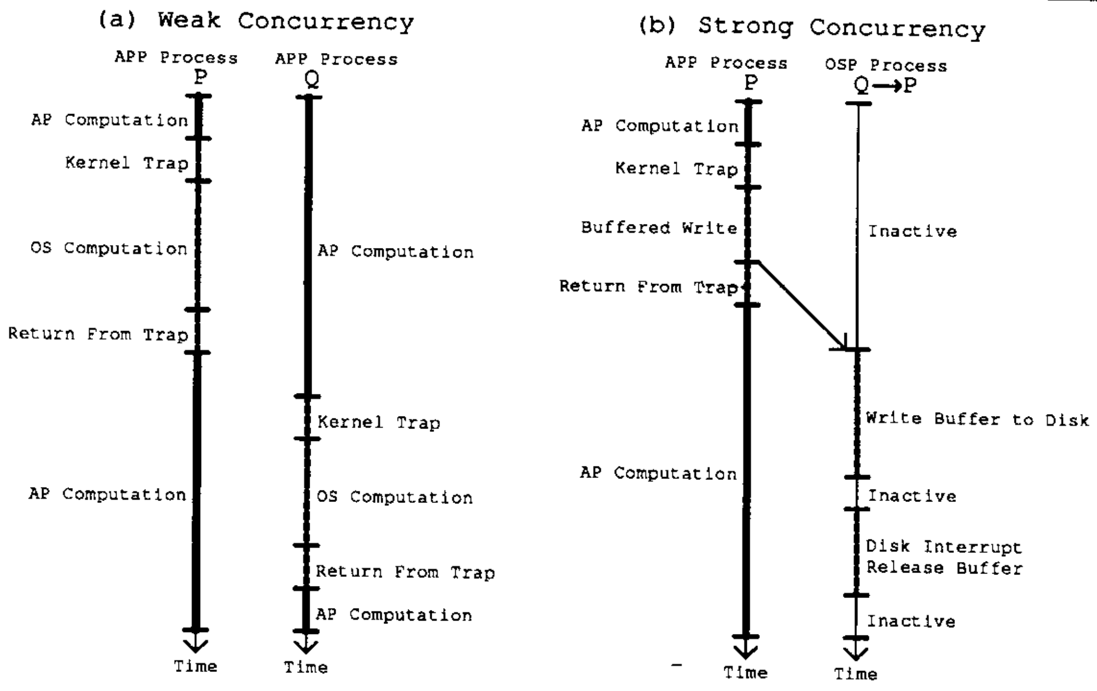

Figure 4-1: Modeling OS/Application Concurrency in UNIX

example, the buffered, delayed writing of file system data blocks can be modeled as a post-computation form of strong concurrency, by introducing an OSP process to represent disk write operations and the associated interrupt handling (see Figure 4-1(b)). Similarly, the prefetching and buffering of sequential file data blocks can be modeled as a pre-computation form of strong concurrency.

4.1.2. A Message-Based System (Accent)

Accent [5] is an example of a message-based operating system. A small message-passing kernel provides the foundation for such a system, and much of the traditional OS functionality resides in separate system processes outside the kernel. Most OS processing is then requested by sending a message to the appropriate OS server process. Modeling a message-based system such as Accent is similar to modeling UNIX, in most respects. A user's application processes would be modeled as APP processes, and global management functions (such as scheduling) would be modeled as GMP processes. Kernel functions (such as message primitives) would be modeled as OS statements within the statement sequences of processes (see Figure 4-2).

Since a single OS server process can do work on behalf of many different application processes, it should be modeled as multiple OSP processes, one for every APP client process. Although OSP processes

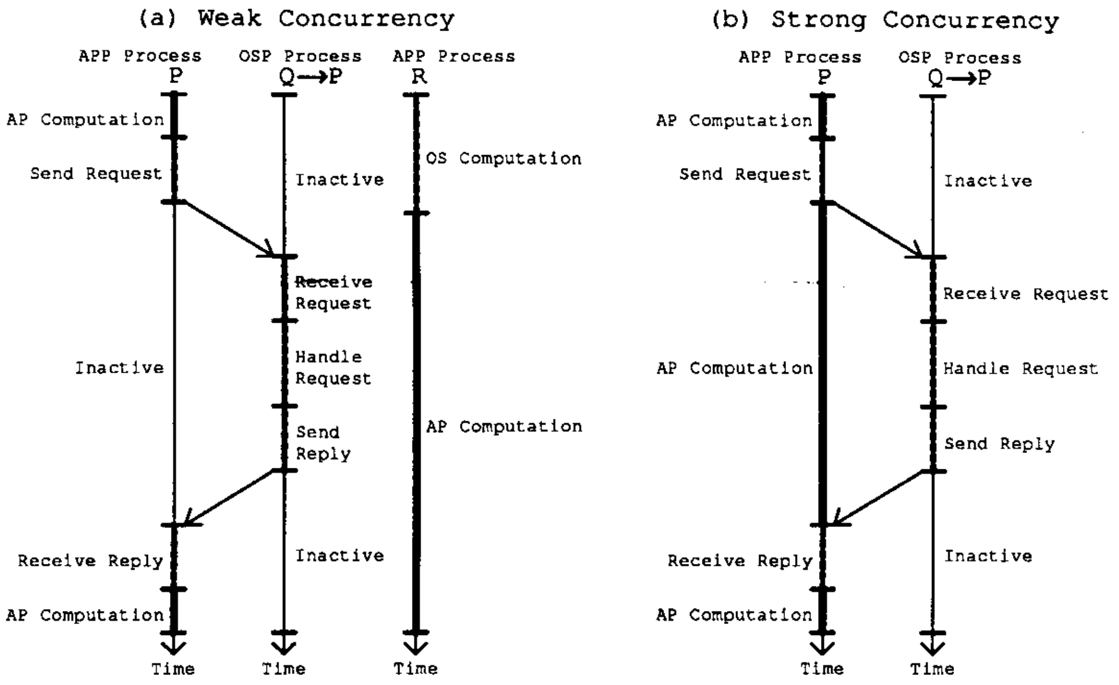

Figure 4-2: Modeling OS/Application Concurrency in Accent

are used, only weak concurrency may be possible (in a multiprocessor system) if the service requests are handled completely synchronously, as shown in Figure 4-2(a). For example, if a client sends a request and then waits for the result, and the server doesn't send the result until the requested OS processing is completely handled, then the application and OS processing will always be serialized with respect to each other. However, if the server buffers the request message, and allows the client to continue with application processing while the server handles the requested OS processing, then the system would be exhibiting strong concurrency (see Figure 4-2(b)).

4.1.3. An Object-Oriented System (Eden)

Eden [3] is an example of an object-oriented system. At the heart of the system is a small kernel supporting object management and operation invocation. Most of the traditional OS functionality then resides in separate system objects outside the kernel. OS processing is requested by invoking operations on the appropriate system objects. Since objects in Eden are active entities (each has a process associated with it), the similarities between the Eden object-oriented system and the Accent message-based system, at least from a modeling standpoint, should be evident. User-defined objects would be modeled as APP processes, and global management functions (such as scheduling) would be modeled as GMP processes. The modeling of system objects would then be done in much the same way as modeling server processes in a message-based system. Multiple OSP processes would be used, with each process corresponding to a different user-defined object that invokes operations on the system object. Weak concurrency and strong concurrency arise in the same way as described above for the Accent system.

4.2. Exploiting OS/Application Concurrency

The OS/application concurrency model aids in identifying the areas where increased concurrency may ~~be possible, and it can be used to predict the resulting performance improvements.~~ In the following examples, three specific techniques for exploiting OS/application concurrency are briefly analyzed. Each technique involves a different form of concurrency (strong, weak, and global management). The analyses follow the approach illustrated earlier in Section 3.

4.2.1. An IPC Processor To Exploit Strong Concurrency

A computer system with an added IPC processor (IPCP) can improve the performance of communicating processes by exploiting strong concurrency. In a message communication facility like that of Accent, the "Send" primitive for queueing a message on a kernel message queue would be modeled as a sequence of OS statements within the sending APP process (see Figure 4-3(a)). Similarly, receiving a queued message would be modeled as a sequence of OS statements within the receiving APP process. As an illustration, Figure 4-3(a) shows a single process that sends a message, does some computation, and then receives the message. Using the time values provided, this entire sequence takes:

$$\begin{aligned} &\text{Send} + \text{Compute} + \text{Receive} \\ &= (\text{Trap} + \text{Validate} + \text{Copy} + \text{Wakeup} + \text{Return}) + (\text{AP Computation}) + \\ &\quad (\text{Trap} + \text{Find} + \text{Copy} + \text{Wakeup} + \text{Return}) \\ &= (1T + 1T + 1T + 1T + 1T) + (6T) + (1T + 1T + 1T + 1T + 1T) \\ &= 16T \end{aligned}$$

To introduce strong concurrency in the message transfer sequence, the IPC primitives are modified. Instead of queueing messages on a kernel queue, messages will now be constructed in a designated buffer area, and newly arrived messages will be available for direct access from the same area. An IPC processor transfers messages from senders' buffers to receivers' buffers. Figure 4-3(b) shows how the send/compute/receive sequence is modeled on this system. The "Send" primitive is modeled as a single OS statement. It flags a message within the buffer as complete and ready for transfer. "Receive" locates a message that has arrived in the buffer area, and it too is modeled as a single OS statement. The IPCP is modeled as multiple OSP processes, one for each APP process that sends or receives messages.

In Figure 4-3(b), the statements of the *IPCP* process are all strong-concurrent-with "AP Computation"

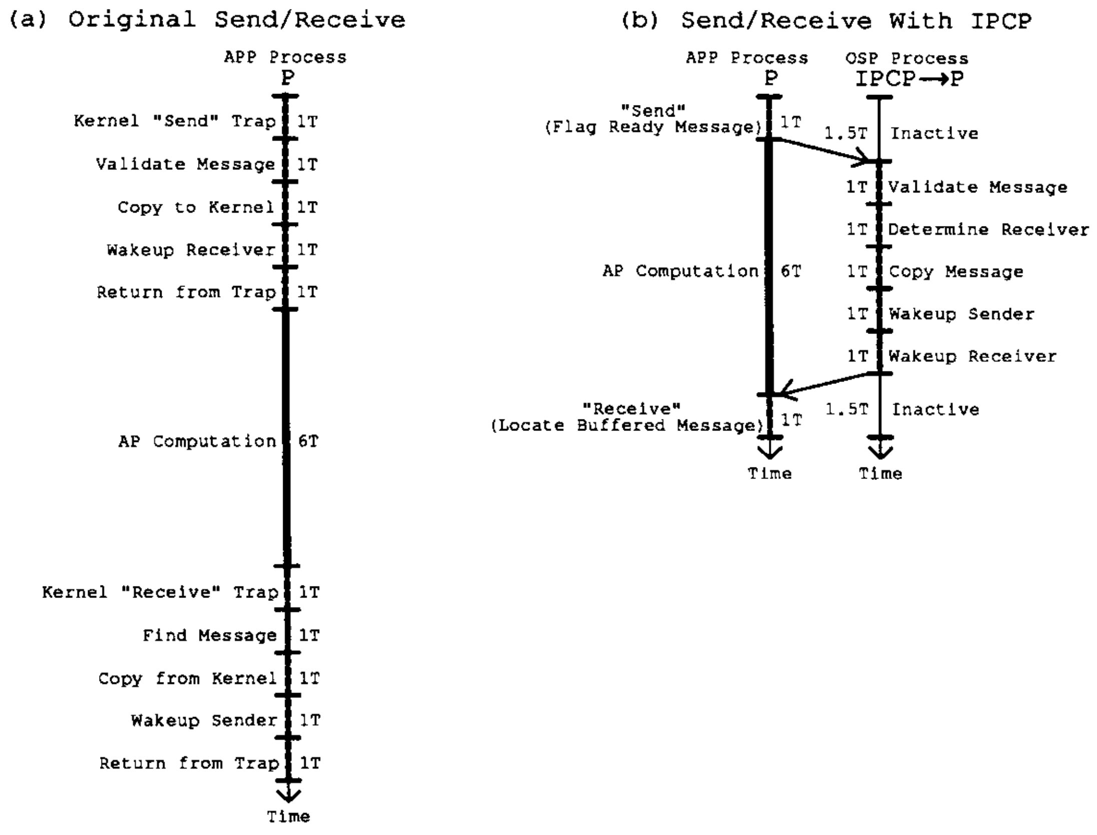

Figure 4-3: Exploiting Strong Concurrency With An IPCP

in Process *P*. The time required for the complete send/compute/receive sequence is:

$$\begin{aligned} &\text{Send} + \text{Compute} + \text{Receive} \\ &= (\text{Flag Message}) + (\text{AP Computation}) + (\text{Locate Message}) \\ &= (1T) + (6T) + (1T) \\ &= 8T \end{aligned}$$

The elapsed time ratio is ETR = 8T/16T = 0.5, indicating that the elapsed time has been reduced to half of what it was originally. The savings come from reducing the non-overlapped processing time for ''Send'' and ''Receive'' by 4T units each. More details concerning the design, implementation, and performance analysis of an IPCP similar to the one outlined here can be found in [7, 8].

4.2.2. An OS Processor To Exploit Weak Concurrency

As was indicated earlier, weak concurrency arises naturally in almost any multiprocessor system, since OS processing on behalf of one application process can proceed in parallel with the execution of other application processes. In some systems, such as Purdue's dual processor UNIX [2], all OS processing is handled by a designated OS processor (also called the ''Master''). Application processing can be done on either the Master or the "Slave(s)" (application processors). When a process is executing on a Slave and requests some OS processing, the Slave must switch processes. The Master can then handle the requested OS processing, while the Slave executes another application process.

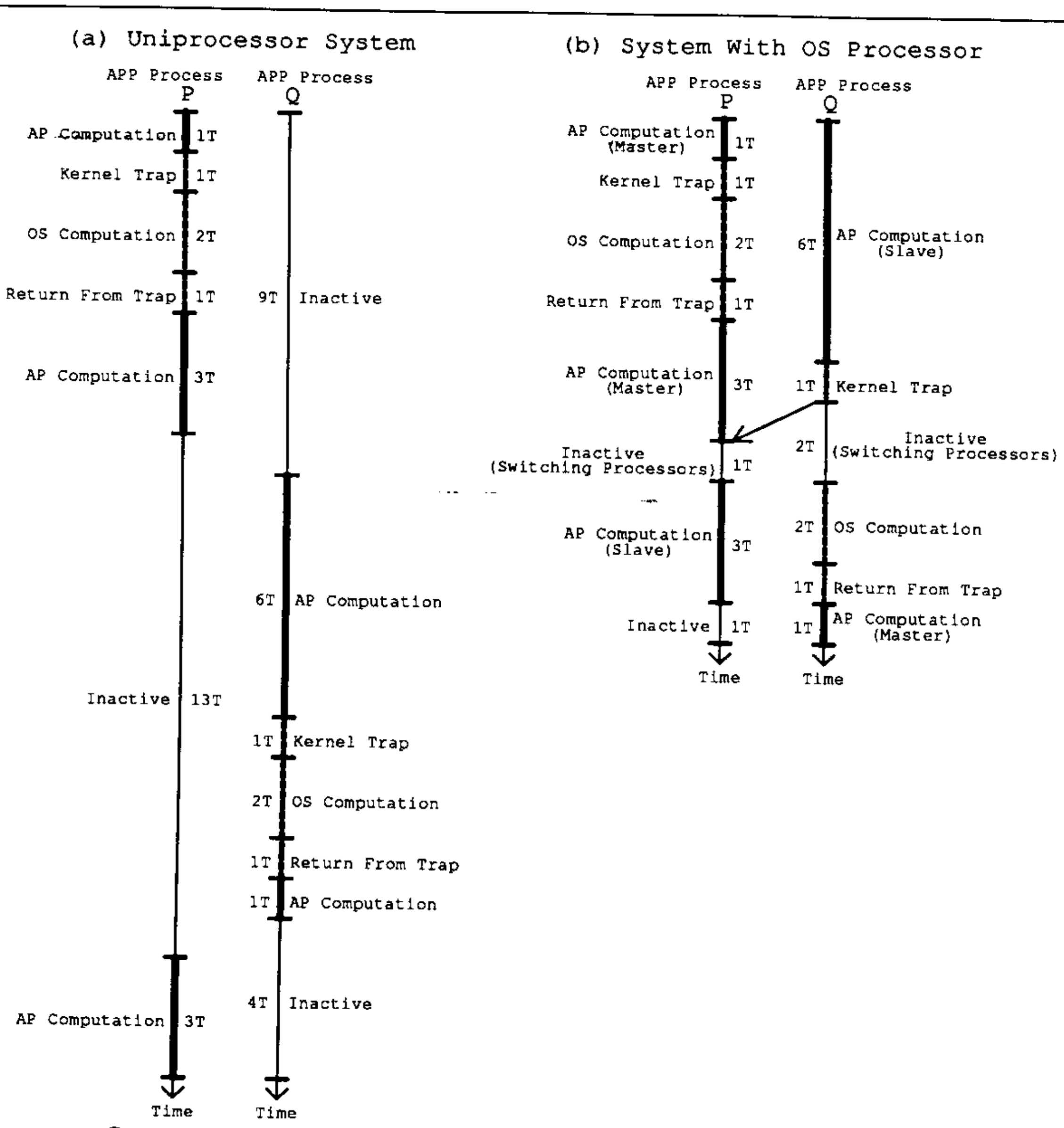

Figure 4-4: Exploiting Weak Concurrency With An OS Processor

Figure 4-4 illustrates how weak concurrency is achieved in a system with an OS processor, and shows the resulting performance improvements. Two APP processes, *P* and *Q*, are both ready to run. On a uniprocessor system (Figure 4-4(a)) their execution is interleaved, and hence there is no concurrency.

Both processes are simultaneously inactive for a 1T period of time whenever the processor is being switched between them. From the times shown in the uniprocessor model, the total time to complete the two processes is 24T.

On the system with an OS processor (Figure 4-4(b)), process *P* first executes on the Master while process *Q* executes on a Slave. Before *Q* can do its "OS Computation", it must switch to the Master. There is a 1T delay before the Master notices that *Q* is waiting for it, and then another 1T is needed for the Master and Slave to switch processes. All of the OS statements in *P* are weak-concurrent-with "AP Computation" in *Q*, and vice versa. There is also some application/application concurrency (2T units). As a result of the concurrency, the total time to complete the two processes is now 13T, and so ETR = 13T/24T = 0.542 (the elapsed time has been reduced by nearly half).

4.2.3. A Scheduling Processor To Exploit Global Management Concurrency

As a final example, consider a real-time system that uses a complex, computation intensive scheduling algorithm, such as described in [4]. Scheduling is modeled as a global management process (GMP), since it is not actually performed on behalf of a particular application process (see Figure 4-5). On a uniprocessor system (Figure 4-5(a)) there is no concurrency, and thus the scheduler's computation time contributes directly to the total elapsed time. In this particular example there are two APP processes, *P* and *Q*, that are both ready to run. *P* executes first, and when it "Waits" for an event the scheduler (*SCHED*) chooses to run *Q*, and switches to it. The total elapsed time to complete both *P* and *Q* is 15T.

When a scheduling processor is added to the system (Figure 4-5(b)), *SCHED* can make the decision to next run *Q* while *P* is still executing ("Choose Next Process" in *SCHED* is global-management-concurrent-with "AP Computation" in *P*). *SCHED* can also prepare for the switch to *Q* by preloading part of its saved execution state, assuming the system architecture permits it. The actual switch from *P* to *Q* can then be done more efficiently (.5T compared to 1T in the example). As a result of the global management concurrency, the total elapsed time to complete both *P* and *Q* is 10.5T, giving ETR = 10.5T/15T = 0.7 (the elapsed time has been reduced 30%).

5. Conclusion

This paper has presented a model for describing the processing performed on a computer system, and in particular the relationship between OS processing and application processing. The purpose of the model is to provide a common framework for describing the OS/application concurrency in different system environments, and to aid in identifying the areas where concurrency can be increased through appropriate

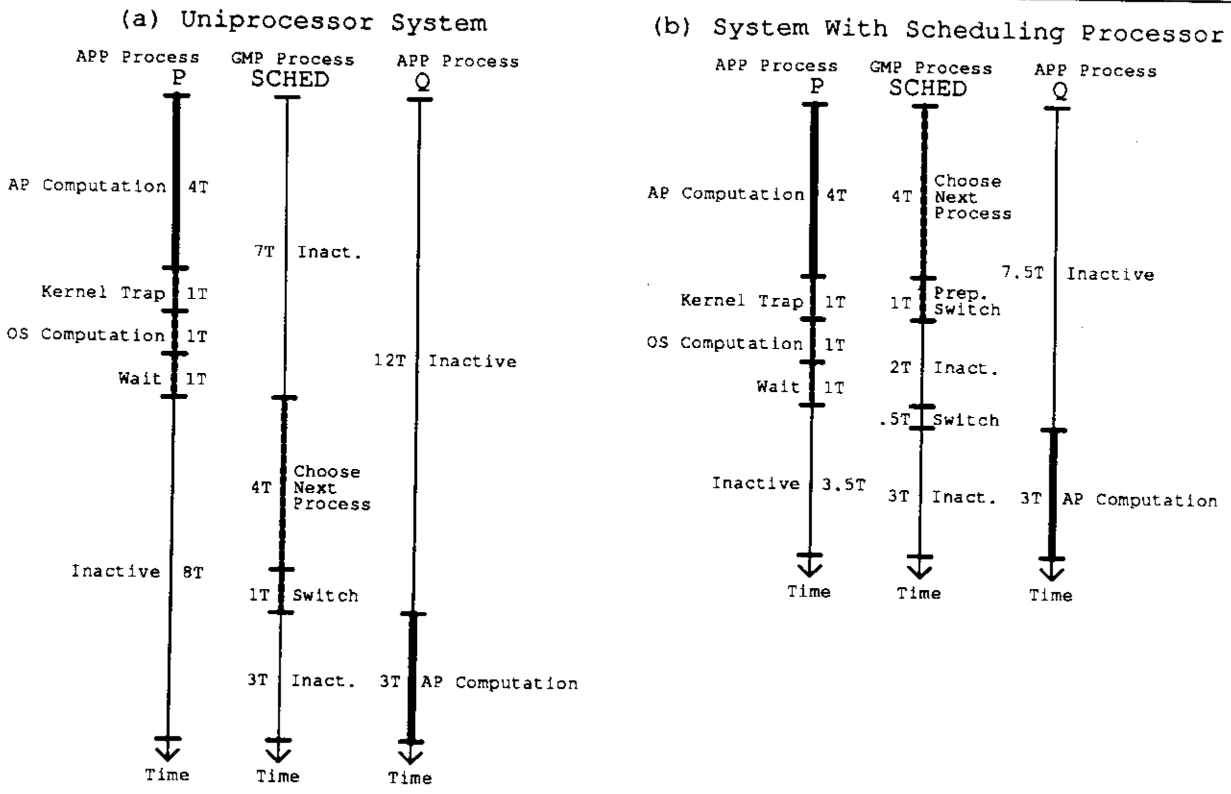

Figure 4-5: Exploiting Global Management Concurrency With A Scheduling Processor

system modifications. The model can also be used to predict the potential performance improvements resulting from the increased concurrency. The usefulness of the model for these purposes was demonstrated by means of specific examples. It was applied to a monolithic kernel (UNIX), a message-based system (Accent), and an object-oriented system (Eden). It was also used to analyze the effectiveness of three particular techniques for increasing the amount of OS/application concurrency. These three techniques demonstrated the different forms of OS/application concurrency: strong, weak, and global management.

Acknowledgments

I would like to thank Anita Jones for originally suggesting the development of a precise model, and for helping me understand what it means to be precise. The current model developed out of earlier discussions with Anita. I would also like to thank Hide Tokuda for his constant advice and encouragement, as well as Tom Lawrence of RADC and Debora Teasdale of NOSC for their support.

References

[1] Bach, M.J. and Buroff, S.J.
Multiprocessor UNIX Operating Systems.
AT&T Bell Laboratories Technical Journal 63(8):1733-1749, October, 1984.

[2] Goble, G.H. and Marsh, M.H.
A Dual Processor VAX 11/780.
Technical Report TR-EE 81-31, Purdue University, School of Electrical Engineering, September, 1981.

[3] Lazowska, E.D., Levy, H.M., Almes, G.T., Fischer, M.J., Fowler, R.J., and Vestal, S.C.
The Architecture of the Eden System.
In *Proc. of 8th Symp. on Operating Systems Principles*, pages 148-159. ACM, December, 1981.

[4] Locke, C.D.
Best-Effort Decision Making for Real-Time Scheduling.
PhD thesis, Carnegie Mellon University, Computer Science Department, 1986.

[5] Rashid, R.F. and Robertson, G.G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. of 8th Symp. on Operating Systems Principles*, pages 64-75. ACM, December, 1981.

[6] Ritchie, D.M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[7] Wendorf, J.W.
Operating System / Application Concurrency in Tightly-Coupled Multiple-Processor Systems.
PhD thesis, Carnegie Mellon University, Computer Science Department, 1987.
(In preparation).

[8] Wendorf, J.W. and Tokuda, H.
An Interprocess Communication Processor: Exploiting OS/Application Concurrency.
Technical Report, Carnegie Mellon University, Computer Science Department, ART Project, March, 1987.