Translation from Transactions of IPSJ

A Method for Analyzing the Mutual Exclusion Overhead of Tightly Coupled Multiprocessors

MASAAKI IWASAKI*, YOSHIFUMI TAKAMOTO* and SEIICHI YOSHIZUMI**

This article reports an analytical model for the mutual exclusion overhead in a tightly coupled multiprocessor (TCMP) system. In mainframe TCMPs used as on-line transaction database systems, it is important to reduce mutual exclusion overheads caused by simultaneous access requests to shared system resources.

The analytical model expresses the relationship of the shared resource utilization, the number of processors, and the increase in the number of dynamic-steps. The equations representing the relationship can be solved by the iterative method. Therefore, the method presented here can compute the increase in the number of the dynamic-steps associated with mutual exclusion.

1. Introduction

The rapid growth of today's on-line transaction processing (OLTP) requires greater processing power than a single high-end processor can provide. The tightly coupled multiprocessor (TCMP) is one means of satisfying this requirement, and is presently being used in the OLTP field. In a TCMP system, many processors sharing the main storage operate concurrently under the control of a single operating system, and each processor can execute the operating system code simultaneously.

Though the performance of a TCMP system should be proportional to the number of processors in the system, lock contention and other factors degrade the actual system performance [1–6].

On the other hand, queuing theory is widely applied to the performance evaluation of OLTP systems. Some extensions of the queuing model have been proposed for the purpose of analyzing the behavior of multiprocessor systems [7–9]. However, these analyses assume that the mutual exclusion costs are negligible, so the number of executed instructions (dynamic steps[1]) required in order to achieve a transaction is considered to be constant.

However, to design and implement a mutual exclusion mechanism for an operating system or OLTP system, numerical evaluation of the increase in the number of dynamic steps caused by the mutual exclusion control is required. If the symmetric TCMP system has four or more processors, this dynamic-step increase becomes especially influential in determining system performance [10–12].

This article presents a numerical analysis method for calculating the mutual exclusion overhead of symmetric TCMP systems.

2. Mechanism of the Overhead Increase

In actual TCMP systems, many problems not found in uniprocessor systems restrict the system performance. These problems can be classified into three categories. The first is the increase in the waiting time for each task to acquire the shared-resource lower processor utilization [7, 8]. The second is the increase in the number of mean instruction execution cycles (MIECs) of each processor. The hardware cache mechanism maintaining consistency among the local caches of processors increases the MIECs of each processor in a shared-memory multiprocessor system [4–6]. The third is the increase in the number of dynamic steps per transaction caused by the mutual exclusion control among tasks trying to obtain shared resources.

This article focuses on the problem of the increase in the number of dynamic steps caused by the mutual exclusion control among concurrent tasks. The following sections describe the basic mutual exclusion scheme and the mechanism of dynamic-step increase, and details the two mutual exclusion schemes appearing in the next chapter.

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 11 (1990), pp. 1627–1635.

*Hitachi Central Research Laboratory.
**Hitachi Systems Development Laboratory.

[1] In this article, 'dynamic steps' are the executed instructions required in order to complete a procedure.

2.1 Basic Mutual Exclusion Scheme

Many mainframe TCMP systems employ a wait/post scheme for mutual exclusion of shared resources. This setup consists of the wait procedure and the post procedure. It mutually excludes the shared resource between tasks as follows. If the shared resource is exclusively used by a task ($task_1$), the wait procedure suspends the execution of another task ($task_2$) that tries but fails to acquire the shared resource. When $task_1$ releases the shared resource, it invokes the next procedure ("post procedure"), which resumes the execution of $task_2$.[1]

Each task invokes the wait procedure if it fails to lock the resource. The wait procedure invokes the dispatcher after it has suspended the execution of the task. The dispatcher allocates another ready state task to the processor on which the suspended task had been running.

The task releasing the shared resource checks the existence of waiting tasks. If a waiting task exists, the post procedure is invoked in the context of the posting task. It then changes the state of the waiting task to ready. After this state transition, the posted task is dispatched to a processor. The posted task becomes the running state once again, and locks the shared resource.

In this article, a 'posting task' means a task that invokes a post procedure, and a 'posted task' means a waiting task that is awakened by a posting task.

2.2 Problem of Mutual Exclusion

The dynamic-step increase associated with the wait/post procedure is a two-stage process. In the first stage, the contention among tasks trying to exclusively acquire the shared resource used by the application program or database management system increases the frequency of wait/post procedure invocation. In the second stage, this frequent invocation results in frequent context switching. Thus, each dispatcher running on a different processor competes with all the others to lock a scheduling queue.

In a TCMP environment, many tasks may simultaneously try to acquire the shared resource. Thus, shared resource utilization effectively rises in proportion to the number of processors in a system. If we define the shared resource utilization per transaction α as follows,

$$\alpha = \frac{\text{dynamic-steps with locking shared resource}}{\text{average dynamic-steps per transaction}}$$

the shared resource utilization β in a uniprocessor environment is almost equivalent to α. However, in a TCMP system, the shared resource utilization β is close to the product of α and the number of processors P.[2]

Therefore, the influence of mutual exclusion overhead becomes significant in a TCMP system with many processors. The dispatcher must lock the scheduling queue that is shared among all processors in a system when context switching has taken place. The frequent invocation of wait/post procedures and dispatcher lock contention among instances of the dispatcher running simultaneously on the different processors lead to a dynamic-step increase.

2.3 Tuning Technique

To reduce the mutual exclusion overhead, it is necessary to use shared resources less and to minimize the costs of wait/post procedures. In addition, the following three issues must be considered for the TCMP system:

1. Frequency of context switching
2. Rise in shared resource utilization when lock contention occurs
3. Frequency of attempts to lock shared resource

First, let us consider item 1. Two alternative control schemes are available after completion of the post procedure, depending on whether context switching occurs or not. After the wait procedure is complete, the operating system usually dispatches another task to prevent the processor from becoming idle. After the post procedure, two different schemes are available. One scheme does not invoke the dispatcher, and continues execution of the posting task. The other scheme invokes the dispatcher and preempts execution of the posting task, then switches to another task (for example, to the posted task).

In the multiprocessor system, even though the posting task is not interrupted and continues execution, the dispatcher on another processor can dispatch the posted task. Many processors share a single scheduling queue in the TCMP system, so if the number of processors increases, the dispatcher lock contention will result in a performance bottleneck. Thus, it is preferable that no context switching should take place after completion of the post procedure.

Next, let us consider items 2 and 3. In the uniprocessor environment, if the posted task is dispatched prior to other tasks, it can definitely acquire the shared resource; that is, if the dispatcher immediately switches the context to the posted task after completion of the post procedure, other tasks cannot acquire the shared resource before this posted task.[1]

On the other hand, in the multiprocessor environment, even if the dispatcher immediately switches the context to the posted task after completion of the post procedure, there is no guarantee that the posted task can use the shared resource before other tasks. This is because the tasks running on the other processors (for

[1] Precisely speaking, $task_2$ is not always posted by $task_1$. Generally, if many tasks are waiting for the release of a shared resource, they may be queued and posted sequentially.

[2] Generally, the denominator of the above expression must include the term representing the time spent waiting to acquire the shared resource. We assume that the waiting time becomes negligible, as explained in Section 3.1.

[1] Actually, if the external interrupt occurs before the posted task has locked the shared resource, the dispatcher may allocate the processor to other tasks. We ignore this case in the following discussion.

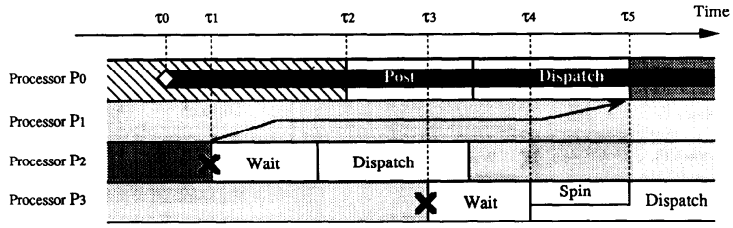

Fig. 1 Mutual Exclusion by First-In/First-Out Scheme.

example, task X) may acquire the same shared resource before the posted task is actually dispatched and begins to run.

Two alternatives are available for requesting the shared resource from task X:

1. Allow task X to use the shared resource before the posted task.
2. Suspend execution of task X until the shared resource is released.

In the first scheme, the posted task must again try to lock the shared resource. If it fails, the posted task is suspended again. Such a failure increases the number of further attempts.

In the second scheme, on the other hand, the posted task can definitely lock the shared resource at the first attempt. However, task X cannot use the shared resource from the beginning of the post procedure to the end of the dispatch procedure. (The posted task does not use the shared resource during this period, because it is not yet running) This means that when the post procedure is invoked (in other words, when lock contention occurs), the busy ratio of the shared resource effectively increases. To avoid this problem, it is necessary to shorten the time interval in which task X cannot use the shared resource. For example, if a processor on which the posting task has run switches the context to the posted task immediately after the post procedure, conclusion of the exclusive use of the shared resource by the posted task should take place as soon as possible.

2.4 *FIFO* Scheme and *Quick Release* Scheme

Before proceeding to the numerical analysis of *FIFO* and the *Quick Release* schemes, this section explains the relation between these schemes and the three issues described in the previous section.

FIFO mutual exclusion corresponds to the second scheme described in the previous section. This scheme avoids an increase in the rumber of further tries. The dispatcher switches the context to the posted task immediately after completion of the post procedure.[1] An example trace of mutual exclusion with *FIFO* is shown in Fig. 1. The task running on processor P_0 locks the shared resource during the period from τ_0 to τ_2. The task on processor P_2 fails to lock the shared resource at time τ_1. The posting task on processor P_0 begins posting to the waiting task at τ_2, and the posted task resumes execution after completion of the dispatch procedure at τ_5.

The advantage of *FIFO*[2] is that the posted task is always able to acquire the shared resource at the second attempt. The drawback is that this scheme raises the utilization of the shared resource, as shown in Fig. 1. Other tasks cannot lock the shared resource during the period from τ_2 to τ_5 even though the posted task is not yet actually using the shared resource. The task running on processor P_3 fails to lock the shared resource at τ_3. In addition to this problem, the spin overhead is increased because of the contention for the dispatcher lock, since the dispatcher is executed on processor P_3 during the period from τ_4 to τ_5.

The *Quick Release* mutual exclusion scheme corresponds to the first scheme described in the previous section. This scheme avoids increasing the context switching and raising the shared resource utilization while the post procedure is executed. Figure 2 shows an example trace of mutual exclusion with the *Quick Release* scheme. The task running on processor P_0 locks the shared resource during the period from τ_0 to τ_2. The task on processor P_2 fails to lock the shared resource at τ_1. The posting task on processor P_0 releases the shared resource at τ_2, and continues execution after the completion of posting to the waiting task. The task on processor P_3 can outstrip the posted task and lock the shared resource at τ_3.

The advantage of the *Quick Release* scheme is that no context switching takes place after completion of the post procedure, and tasks other than the posted task can acquire the shared resource before the posted task actually uses it. The drawback is that the resumed posted task may fail to lock the shared resource again, as shown in Fig. 2. If the task running on processor P_3 releases the shared resource before τ_4, the posted task can acquire the resource at the second attempt.

[1] This context switch decreases the cache hit ratio of the processor. However, discussion of this problem is beyond the scope of this paper.

[2] *FIFO* means that the posted task is never outstripped, and that all tasks requiring the shared resource can be served on a first-in/first-out basis.

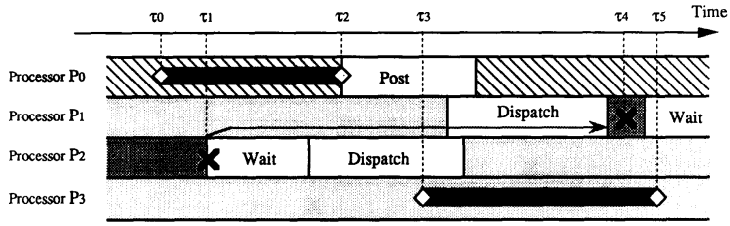

Fig. 2 Mutual Exclusion by Quick Release Scheme.

3. Modeling

In this section, we derive the equations representing the mutual exclusion overhead, then explain their solution.

3.1 Assumptions

In order to derive the equations, we make the following assumptions:

1. The system is a symmetric shared-memory tightly coupled multiprocessor. There is only one scheduling queue in the whole system and it is shared by all processors. The spin lock (busy wait) mechanism [1, 4] is used for the mutual exclusion of this scheduling queue. All kernel codes, including the dispatcher, can run on any processor. No interruptions are allowed during the execution of a kernel code.
2. The system operates stably with a sufficient number of ready state tasks, maintaining the utilization of all processors at 1.0. Each task may travel over many processors during its life, and thus the load is balanced and each processor requests an equivalent number of locks for the shared resource.
3. The variation in the length of the mean instruction execution cycle (MIEC) is negligible. The increase in the length of the MIEC in a multiprocessor environment due to serialization or cache coherency control is not significant.
4. The task that locks the shared resource is dispatched to a processor before other tasks. The task is rarely suspended when it has locked the shared resource.
5. The execution of the task locking the shared resource is rarely interrupted.

These assumptions are not always true in an actual system. In particular, the second assumption makes utilization of the shared resource higher than it is in actual systems. In actual systems, if the utilization of the shared resource approaches 1.0, many tasks are suspended and wait for the release of the shared resource. This increases the processor's idle time. In this case, the dynamic-step increase is less important than the waiting time increase.

Our aim is to understand the relationship between the dynamic-step increase and other factors such as the number of processors, shared resource utilization, and mutual exclusion costs. Thus, in the following discussion, we assume that the TCMP system is tuned up so that the processors are never idle, even if the shared resource utilization becomes high.

3.2 Derivation of Equations

First, we define the ratio of uniprocessor to multiprocessor dynamic-steps, E_P, as

$$E_P=\frac{T_1}{T_P}, \tag{1}$$

where T_1 is the average number of dynamic-steps per transaction in a uniprocessor environment, and T_P is the average number of dynamic steps per transaction in a multiprocessor environment. Subscript P represents the number of processors. Since we denote the retry overhead (the increase in the number of dynamic steps due to the retry procedure) per transaction as δ_{LK}, and the dispatch overhead (the increase in the number of dynamic steps due to the dispatch procedure) per transaction as δ_{OS}, T_P is represented as follows:

$$T_P=T_1+\delta_{LK}+\delta_{OS}. \tag{2}$$

Denoting the cost (the number of dynamic steps per execution of a procedure) of the post procedure as ε_{POST}, and the cost of the wait procedure as ε_{WAIT}, the retry overhead per transaction, δ_{LK}, is represented as

$$\delta_{LK}=\Delta(\varepsilon_{POST}+\varepsilon_{WAIT}), \tag{3}$$

where Δ is the mean number of lock acquisition failures per transaction. The value of Δ depends on the mutual exclusion scheme. The values of Δ for the *FIFO* scheme and the *Quick Release* scheme are not the same.

When using the *FIFO* scheme, as shown in Fig. 1, the task resumed by the posting task can always acquire the shared resource. Thus, the number of lock acquisition failures per transaction Δ can be expressed as

$$\Delta=\beta_{LK}N_{LK}, \tag{4a}$$

where β_{LK} denotes the probability that the task fails to lock the shared resource, and N_{LK} denotes the number of lock acquisitions necessary to complete a transaction.

When the *Quick Release* scheme is used, all tasks ex-

cept the posted task can lock the resource in the period from τ_2 to τ_4, as shown in Fig. 2. Thus, the posted task cannot always acquire the resource. Therefore, the mean failure count of the first lock acquisition tries is $\beta_{LK} N_{LK}$, and the mean success count of the first lock acquisition tries is $(1-\beta_{LK})N_{LK}$. Each failed try will cause a retry, where the mean failure count is $\beta_{LK}^2 N_{LK}$. The same is true of the third and subsequent tries. Hence, the accumulated lock failure count per tansaction is represented by the following geometrical series:

$$\beta_{LK} N_{LK}+\beta_{LK}^2 N_{LK}+\beta_{LK}^3 N_{LK}+\cdots = N_{LK}\sum_{k=1}^{\infty}\beta_{LK}^k = \frac{\beta_{LK}}{1-\beta_{LK}} N_{LK}. \quad (5)$$

This accumulated count corresponds to the mean retry count per transaction Δ.

$$\Delta = \frac{\beta_{LK}}{1-\beta_{LK}} N_{LK}. \quad (4b)$$

The lock acquisition failure ratio β_{LK} equals the probability that another task running on another processor has already locked the resource when the initial task issues a lock acquisition request. Assuming that the processor utilization is 1.0, β_{LK} is represented as

$$\beta_{LK}=(P-1)\alpha_{LK}, \quad (6)$$

where P is the number of processors, and α_{LK} is the shared resource utilization per transaction. We will examine α_{LK} at the end of this section.

Since we denote the number of context switches per transaction in a uniprocessor environment as N_{OS}, the increases in context switching due to lock contention as Γ, the dispatcher lock acquisition failure ratio as β_{OS}, and the cost of the dispatch procedure as λ_{OS}, the dispatch overhead δ_{OS} is represented as follows:

$$\delta_{OK}=\Gamma\lambda_{OS}+\frac{\beta_{OS}}{1-\beta_{OS}}(\Gamma+N_{OS})\lambda_{SPIN}. \quad (7)$$

The first term on the right is the product of the increase in the context switching count and the cost of a dispatch procedure. The second term is the spin overhead due to the dispatcher lock contention.

λ_{SPIN} is the number of dynamic steps for single loop execution of the SpinLoop1 shown below.

```
SpinLoop1   Load              R5, #Locked
SpinLoop2   Load              R4, Lockword
            Compare           R4, #Free
            BranchNotEqual    SpinLoop2
            Load              R4, #Free
            CompareAndSwap    R4, R5, Lockword
            BranchNotZero     SpinLoop1
```

The inner loop SpinLoop2 of this nested loop waits for the release of the shared resource. The outer loop SpinLoop1 tries to lock the shared resource. Each O mark in Fig. 3 represents an execution of SpinLoop2, and each X mark represents a lock acquisition failure at the CompareAndSwap instruction in SpinLoop1. λ_{SPIN} is the number of dynamic steps between two X marks. The number of dynamic steps of SpinLoop2 is equal to λ_{OS}, not counting the first execution of this loop. Though the number of dynamic steps of this first execution actually depends on the dispatcher lock failure ratio β_{OS}, in the following discussion we assume that λ_{SPIN} and λ_{OS} are equal. Using this approximation, the dispatch overhead δ_{OS} is expressed as follows:

$$\delta_{OS}=\Gamma\lambda_{OS}+\frac{\beta_{OS}}{1-\beta_{OS}}(\Gamma+N_{OS})\lambda_{OS}=\frac{\lambda_{OS}(\Gamma+\beta_{OS}N_{OS})}{1-\beta_{OS}}. \quad (7)$$

A context switch takes place after the completion of each wait or post procedure in the *FIFO* scheme. The increased count of context switching Γ is represented as follows, using Δ from Equation (4a):

$$\Gamma=2\Delta=2\beta_{LK}N_{LK}. \quad (8a)$$

On the other hand, a context switch does not take place after completion of the post procedure in the *Quick Release* scheme. Thus, Γ is represented as follows, using Δ from Equation (4b):

$$\Gamma=\Delta=\frac{\beta_{LK}}{1-\beta_{LK}}N_{LK}. \quad (8b)$$

The probability of the dispatcher lock acquisition failure β_{OS} appearing in Equation (7) is represented as follows:

$$\beta_{OS}=(P-1)\alpha_{OS}. \quad (9)$$

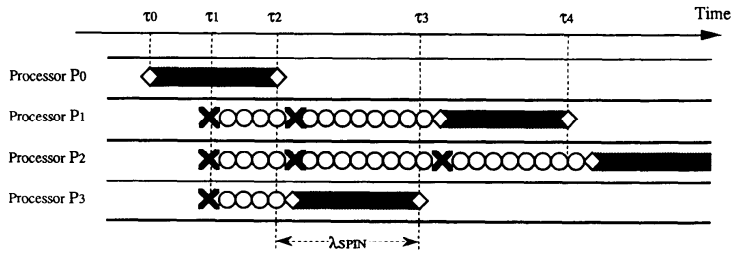

Fig. 3 Spin Overheads.

Finally, we derive the equations representing the shared resource utilization for each transaction, α_{LK}, and the dispatcher lock utilization for each transaction, α_{OS}. Assuming that the processor utilization remains at 1.0, we can define the utilization for each transaction α as follows:

$$\alpha = \frac{\text{dynamic-steps with locking shared resource}}{\text{average dynamic-steps per transaction}}$$

Thus, the shared resource utilization for each transaction, α_{LK}, becomes

$$\alpha_{LK} = \frac{L_{LK} N_{LK} + \beta_{LK} N_{LK} \left(\varepsilon_{POST} + \lambda_{OS} + \frac{\beta_{OS}}{1-\beta_{OS}} \lambda_{OS} \right)}{T_P}.$$

that is,

$$\alpha_{LK} = \frac{L_{LK} N_{LK} + \beta_{LK} N_{LK} \left(\varepsilon_{POST} + \frac{\lambda_{OS}}{1-\beta_{OS}} \right)}{T_P}. \quad (10a)$$

The denominator T_P on the right is the average number of dynamic steps for each transaction appearing in Equation (2). The first term $L_{LK} N_{LK}$ of the numerator is the number of dynamic steps while the shared resource is being used by the task. The second term of the numerator indicates that no task can lock the shared resource during the period from τ_2 to τ_5, as shown in Fig. 1.

In contrast, when the *Quick Release* scheme is used, the shared resource is actually released before the post procedure invocation. Thus, α_{LK} becomes as follows:

$$\alpha_{LK} = \frac{L_{LK} N_{LK}}{T_P}. \quad (10b)$$

Similarly, the dispatcher lock utilization for each transction in a multiprocessor environment, α_{OS}, becomes

$$\alpha_{OS} = \frac{\lambda_{OS}(\Gamma + N_{OS})}{T_P}. \quad (11)$$

where N_{OS} denotes the number of dispatcher lock acquisitions for each transaction in a uniprocessor environment, and Γ denotes the increased count of dispatcher lock acquisitions in a multiprocessor environment.

3.3 Solutions

This section gives the solutions of the equations derived above. To calculate the mutual exclusion overhead requires the values of the following eight parameters appearing in Equations (2) to (11). We use the measured or design value of these parameters to calculate the overhead.

- P: number of processors
- T_1: number of dynamic steps for each transaction in a uniprocessor environment
- N_{LK}: number of shared resource locks for each transaction in a uniprocessor environment
- L_{LK}: length of a shared resource lock
- ε_{POST}: cost of a post procedure
- ε_{WAIT}: cost of a wait procedure
- N_{OS}: number of dispatcher locks for each transaction in a uniprocessor environment
- λ_{OS}: length of a dispatcher lock

The aim of our analysis is to calculate the values of the following nine variables in a multiprocessor environment:

- T_P: number of dynamic steps for each transaction
- δ_{LK}: retry overhead for each transaction
- δ_{OS}: increase in the dispatch overhead for each transaction
- Δ: number of wait/post procedure invocations
- β_{LK}: probability of shared resource acquisition failure
- α_{LK}: shared resource utilization for each transaction
- Γ: increase in the number of context switches
- β_{OS}: probability of dispatcher lock acquisition failure
- α_{OS}: dispatcher lock utilization for each transaction

For the *Quick Release* scheme, the biquadratic equation of T_P is derived from Equations (2) to (11) by erasing the other variables, using a computer-assisted formula manipulator such as *Mathematica* [13]. Fortunately, the usual iterative method can calculate the value of T_P for both schemes. We derive the equation $T_P(\alpha_{OS}, \alpha_{LK}) = 0$ by erasing the other variables, and solve this equation by the iterative method, with the initial values of α_{OS} and α_{LK}, which satisfy $0 < P\,\alpha_{OS} < 1$ and $0 < P\,\alpha_{LK} < 1$.

4. Numerical Examples

This section gives two results of calculation using the equations derived in the previous section. The result given in Section 4.1 includes a comparison of calculated values and measured values. The result shown in Section 4.2 predicts the system performance of a symmetric TCMP with more than ten processors.

4.1 Comparison of Measured Data and Calculated Data

The graph in Fig. 4 shows the evaluation results of the benchmark test, which simulates the on-line system in the mainframe TCMP system. The horizontal axis represents the number of processors, and the vertical axis represents the product of E_P and the number of processors.

Case 1 of the benchmark test employs the *FIFO* scheme, and the lock utilization for each transaction, α_{LK}, in a uniprocessor environment[1] is set to about 0.12. The number of dynamic steps in this benchmark test increases drastically. The calcualtion assumes a processor utilization of 1.0, so the calculated value of E_P becomes

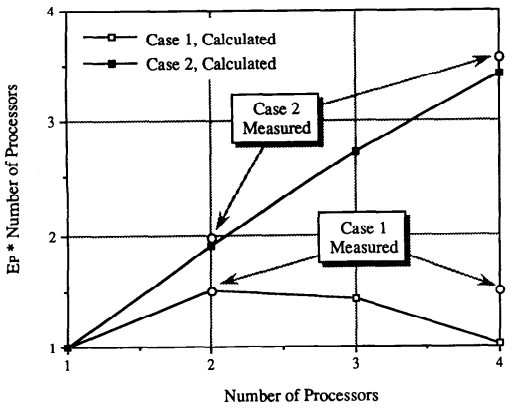

Fig. 4 Comparison of Calculated Values and Measured Values.

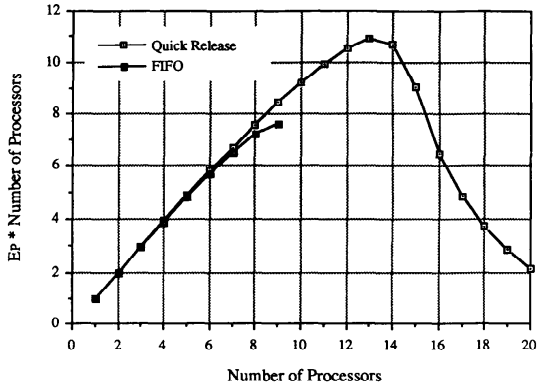

Fig. 5 Number of Processors-Performance Curve.

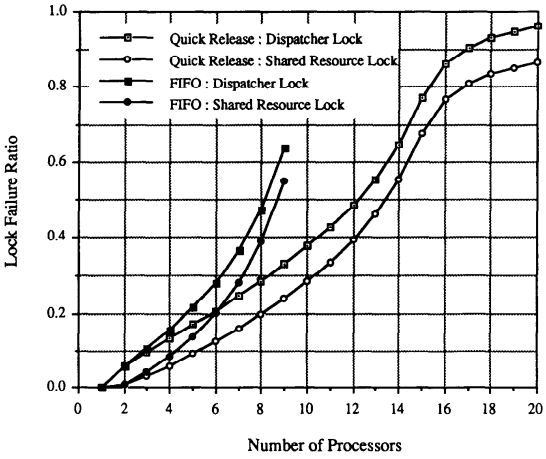

Fig. 6 Number of Processors-Lock Failure Ratio Curve.

greater than the measured value in a 4-processor environment. In an actual system, the increase in the number of dynamic steps is saturated because of the increase in the processors' idle time.

Case 2 of the benchmark test employs the *Quick Release* scheme, and the lock utilization for each transaction, α_{LK}, in a uniprocessor environment is set to about 0.05. The difference between the calculated and measured values of E_P for this benchmark is about 5.5%.

4.2 Large-Scale TCMP

Figures 5 and 6 show the results of calculation for a large-scale symmetric TCMP system. These results indicate that it is necessary to split a critical resource into parts to prevent performance degradation occurring as a result of the frequent contention for the resource in a system with many processors.

It is almost impossible to solve the equations for the *FIFO* scheme by the iterative method when the number of processors exceeds 10. The lock utilization β_{LK} of systems using the *FIFO* scheme reaches 0.7 when the number of processors is only nine. Thus, the processor utilization of an actual system with more than 10 processors will remain very low.

5. Conclusion

An analytical model for the mutual exclusion overhead in a tightly coupled multiprocessor system has been presented. Our analytical model can explain the relationship among the shared resource utilization, the number of processors, and the increase in the number of dynamic steps. The equations derived from this model can be solved by the iterative method.

The developed estimation method allows fast and easy evaluation of the mutual exclusion overhead. The derived equations show the mechanism for increasing the number of dynamic steps in a tightly coupled multiprocessor system. This mechanism is useful for designing and implementing the mutual exclusion scheme of an operating system or an OLTP system.

[1]The lock utilization for each transaction in a uniprocessor environment is defined as follows:

$$\alpha_{LK}=N_{LK}L_{LK}/T_1$$

References

1. HWANG, K. and BRIGGS, F. A. Computer Architecture and Parallel Processing, McGraw-Hill, 1985.
2. HOLLEY, L. H., PARMELEE, R. P., SALISBURY, C. A. and SAUL, D. N. VM/370 Asymmetric Multiprocessing, *IBM Syst. J*, **18**, 1 (1979), 47-70.
3. TETZLAFF, W. H. and BUCO, W. M. VM/370 Attached Processor and Multiprocessor Performance Study, *IBM Syst. J*, **23**, 4(1984), 375-385.
4. BECK, B., KASTEN, B. and THAKKAR, S. VLSI Assist for a Multiprocessor, *Proc. 2nd ASPLOS* (1987), 10-20.
5. GOODMAN, J. R. Coherency for Multiprocessor Virtual Address Caches, *Proc. 2nd ASPLOS* (1987), 72-81.
6. RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W. and CHEW, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor,

Proc. 2nd ASPLOS (1987), 31–39.

7. NELSON, R., TOWSLEY, D. and TANTAWI, A. N. Performance Analysis of Parallel Processing Systems, *IEEE Trans. Soft. Eng.*, **14**, 4 (1988), 532–540.

8. BALBO, G and BRUELL, S. C. Combining Queuing Networks and Generalized Stochastic Petri Nets for the Solution of Complex Models of System Behavior, *IEEE Trans. Comput.*, **37**, 10 (1988), 1251–1268.

9. HEIDELBERGER, P. and LAKSHMI, M. S. A Performance Comparison of Multimicro and Mainframe Database Architectures, *IEEE Trans. Soft. Eng.*, **14**, 4 (1988), 522–540.

10. ANDERSON, T. E. Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, *IEEE Trans. Para. & Dist. Systems*, **1**, 1 (Jan. 1990), 6–16.

11. DINNING, A. A Survey of Synchronization Method for Parallel Computers, *IEEE Computer* (Jul. 1989), 66–77.

12. GRAUNKE, G. and THAKKAR, S. Synchronization Algorithms for Shared-Memory Multiprocessors, *IEEE Computer* (June 1989), 60–69.

13. WOLFRAM, S. Mathematica, Addison-Wesley, 1988.