A Research on Scheduling Policy for Multiple Parallel Environment

多重並列処理環境用のスケジューリング手法に関する研究

by

Yojiro Nobukuni

信国 陽二郎

A Senior Thesis

卒業論文

Submitted to

Department of Information Science

Faculty of Science

The University of Tokyo

on February 14, 1995

in Partial Fulfillment of the Requirements

for the Degree for Bachelor of Science

Thesis Supervisor: Kei Hiraki　平木 敬

Title: Assistant Professor for Information Science

ABSTRACT

A research on general puropose parallel operating systems for distributed or parallel multiprocessor environment, such as NUMA multiprosessor or workstation clustered systems, is done. The aim is to achieve the efficeint concurrent execution of multiple parallel applications in such systems.

2-level scheduling, in which scheduling is divided into user-level and kernel-level, is useful in multiple parallel environment. It is so powerful and allocating resources without violating applications' parallelism and using scheduling informations from user-level lead to a practical general puropose multiple parallel OS.

In the paper, we propose a kernel level scheduling method for a 2-level scheduling policy that takes advantage of informations on hierachical system structure and memory usage of each applicaton. Furthermore, its efficiency is evaluated by simulation studies and compared to other scheduling alternatives.

Reflecting the memory usage of each process to kernel-level scheduling, processes effectively migrate in complicated systems where memory accesses are nonuniform. Thus system performance can be enhanced. Moreover, kernel can use the hierarchical system structure to allocate resources according to scheduling constraints of processes.

論文要旨

NUMA 型並列計算機システムやワークステーションクラスタといった並列・分散計算機環境における、複数の並列アプリケーションの効果的な多重実行を実現する、汎用並列 OS の研究を行なう。汎用並列・分散環境では、 OS のスケジューリング方法が並列アプリケーションの有効性に大きく影響する。

多重並列環境では、スケジューリングをユーザレベルとカーネルレベルに分離することが考えられる。これにより、プロセスの並列性を損なわない資源の割り当てを行ない、ユーザレベルのスケジューリング制約情報を利用することで、汎用並列 OS の構築が可能となる。

本論文では、プロセス毎のメモリの使用状況及びシステム構成の階層性を利用するカーネルスケジューリング法を提案する。さらに、モデルを用いたシミュレーションによりその評価及び、他のスケジューリング法との比較を行なう。プロセス毎のメモリの使用状況を OS 核のスケジューリングに反映させることにより、メモリアクセスのコストが不均一なシステムにおけるプロセスのマイグレイションを効果的に行い、性能を上げることができる。さらにカーネルは、システム自体の階層構造を利用することで、各プロセスの通信やメモリのアクセスコストに関する制約をスケジューリングに導入することができる。

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank Associate Professor Kei Hiraki for helpful advices and heartful encouragements. I am grateful to Takashi Matsumoto for useful suggestions and accurate information for building the simulation model in the paper. In addition, I would like to thank to all other members of Hiraki Laboratory.

Chapter 1

Introduction

To acquire greater processing power, parallel computing systems have been gaining increasing attention. For that goal, an amount of Uniform Memory Access (UMA) systems have been constructed and much research were done. But as those researches showed, small-scale UMA system soon came up with marginal performance of processing power. System bottlenecks such as contention of memory access due to shared memory structure and busses that were frequently used as hardware for interconnection network were the primary reasons for that limitation. Moreover, making a large-scale UMA multiperocessor system is costly and it is hard to enlarge the number of processor elements.

But rapid progress in micro electronic technologies made it practical to construct a large-scale NUMA multiprocessor system with a large number of relatively low cost and simple components.

In addition, advanced techniques and methods gained by researches on parallel applications and UMA systems such as; improved compile techniques; program and data distribution method; useful scheduling method; and synchronization mechanisms, are helpful for effective use of large-scale NUMA multiprocessor systems and played important roles in making it practical. Consequently, those systems are not only coming near to meet our processing needs, but also showing possibility of replacing the so called mainframe computers and the power of parallel processing in distributed environment.

For this reason, providing a multiprogramming environment to achieve high system utilization in large-scale multiprocessor systems is supreme issue. Therefore, it is important to establish operating system for general use of NUMA multiprocessor system.

We are exploiting a general purpose multiple parallel OS kernel for Non-Uniform Memory Access (NUMA) multiprocessor systems called SSS-CORE. It is a time shared system that accomplishes multiple user and multiple jobs environment which is an inevitable property for an general purpose system. At the same time, it aims at not decreasing the high efficiency of parallel applications, which have been inconsistent with general purpose systems.

For a parallel application to gain high efficiency, it is very critical to schedule its threads simultaneously, close to each other, and onto processors and memories where it have been scheduled before. This gives importance to schedulling method.

However, some sorts of things differ between applications. For example, the frequency of synchronization and memory pages it requires through a execution. In addition, the timing applications produce synchronizations, the palcement of memory pages when a memory access occurs, and etc., cannot be statically predicted by scheduler.

Our idea is that providing user-level an interface to exchange scheduling hints, or constraints, with kernel-level enables scrupulous resource allocation that satisfy each application's demand. Thus overall performance is enhanced and constructing generall purpose parallel OS becomes practical.

In the paper, we propose a kernel level scheduling method for a 2-level scheduling policy that takes advantage of informations on hierachical system structure and memory usage of each applicaton. Furthermore, its efficiency is evaluated by simulation studies and compared to other scheduling alternatives.

The paper is constructed as follows. We briefly survey SSS-CORE operating system

kernel in chapter 3 together with its background and features. An overview of kernel-level scheduling is given in chapter 4 with our main ideas and problems. In chapter 5, details of kernel-level scheduling method implemented in the model is described. Chapter 6 explains construction of the model. Future work that include remaining studies of kernel-level scheduling in 2-level scheduling policy for SSS-CORE and required improvements of the simulator are showed in chpater 7. At last, we conclude in chapter 8.

Chapter 2

Related Work

In contrast to UMA multiprocessor systems, relatively small amount of research has been done on large-scale NUMA multiprocessors. Zhou and Brecht[1] proposed a scheduling algorithm based on processor pools. They divided processors into abstract groups and called them processor pools. Each pool has a single run queue shared by all the processors in the pool. Though the study stressed the effectiveness of using processor pools by analytic simulation on system and workload models, a problem can be pointed out. Large parallel jobs spanned to multiple pools and threads of such jobs were not guaranteed to be scheduled simultaneously. They paid little attention on this issue because they worked on so called fork-and-join execution model. But, for parallel applications that perform many communication between their threads, it is critical to schedule them simultaneously, because delayed time caused by synchronization do extraodinary harm. In [2], they gave another analysis on a NUMA system. they studied on stochastic scheduling for fork-and-join model. model in a NUMA system. Fork-and-join structured jobs can be expressed in our model by phase change.

As for exchanging scheduling information, useful methods to make interfaces between user-level and kernel-level is proposed in [3] and [4].

Chapter 3

SSS-CORE

3.1 What Is SSS-CORE?

SSS-CORE is a general purpose multiple parallel OS kernel for Non-Uniform Memory Access (NUMA) multiprocessor system. It is a time shared system that accomplishes multiple user and multiple job environment which is an inevitable property for an general purpose system. It provides a way for users to give OS kernel a set of scheduling hints called constraints for resource allocation. They are used when scheduling. Furthermore, optimization of execution code is supported by the low cost interfaces between parallel applications and SSS-CORE to exchange resource allocation information.

A prototype system is currently under construction.

3.2 Background

To acquire greater processing power, parallel computing systems have been gaining increasing attention. For that goal, an amount of Uniform Memory Access (UMA) systems have been constructed and much research were done. But as those researches showed, small-scale UMA system soon came up with marginal performance of processing power. System bottlenecks such as contention of memory access due to shared

memory structure and buses that were frequently used for network hardware were the primal reasons for that limitation. Moreover, making a large-scale UMA multiperocessor system was costly and it was hard to enlarge the number of processor elements in the system.

But rapid progress in micro electronic technologies made it practical to construct a large-scale NUMA multiprocessor system with; a large number of relatively low cost processors; and high performance and inexpensive network hardwares. In addition, advanced techniques and methods obtained by researches on parallel applications and UMA system such as; improved compile techniques; program and data distribution method; useful scheduling method; and synchronization mechanisms, are helpful for effective use of large-scale NUMA multiprocessor systems and played important roles in making it practical.

but also showing possibility of replacing the so called mainframe computers and power of parallel processing in distributed environment. Therefore, it is important to establish operating system for general use of NUMA multiprocessor system.

3.3 Features of SSS-CORE

To be a general purpose parallel OS, it must be a time sharing system and support multiple user and multiple jobs. In addtion, mechanisms are needed to support programmer and compiler level optimization. For example, free resource management in the range of resource protection; citation of constraints on resource allocation; and opening scheduling information to user-level for runtime optimization. The followings are chief features included in SSS-CORE.

- 2-level scheduling (Kernel-level and user-level scheduling) targetted to NUMA system.
  - Introducing topology and communication cost to kernel-level scheduling.

  - Reflecting memory access (to remote memories and secondary memory) cost to kernel-level scheduling.

- Management of memory resources and secondary memory.
  - Reflecting allocation information of physical memory pages to kernel-scheduling.

- Non-blocking I/O system.

3.4 Target System

The target system of SSS-CORE is NUMA multiprocessor system. It may be a distributed memory parallel computer, a destributed shared-memory computer, workstation cluster system, or etc.. The topology of the interconnection network can be of any kind, such as hierarchical clustered network, or ring network. In addition, the type of the actual hadware of the interconnection network ,suche as buses or switches, is not cared. But initially we only consider hogeneous cluster system for simplicity.

Remote memory access also can be supported by any type of system such as special hardware, software-based assynchronous communication, or virtual shared memory system, like IVY[5]. Futhermore, one job does not require physical memory more than total amount of memory in overal system, and secondary memory is assumed to be same distance away from every cluster. So the order of the cost of memory access increases in the order of in-cluster memory access, cluster-cluster memory access, and secondary memory access.

3.5 Execution Model

We call the end part of the system network **cluster**, which may be consisted of a processor with one memory bank or a small-scale UMA system. A stream of instructions allocated to a processor is called **thread**. A parallel process , a **process**, or job, is made

from multiple threads. A set of threads that belong to the same cluster is called **subprocess**. **Process** is the object for kernel-level scheduling. When a process changes its scheduling constraints, it changes its **phase**. For example, requesting more processor to kernel is a change of phase. A process can change phase at any time, however, if it makes execution impossible, the whole process will be preempted. To support this execution model; each resource must be preempted at any time ;and each process, when time quantum expired and on phase change.

3.6 Resource Management

2-level scheduling policy is adopted as SSS-CORE's scheduling policy. It is devided into 2 parts. One is kernel-level scheduling and another is user-level scheduling. Kernel-level scheduling focuses on resource allocation to processes, on the other hand user-level schecduling is interested in scheduling threads whithin a process.

Resources includes processors, memory pages, and tracks or sectors of hard disks. For allocating resources, we arrange a virtual resource management tree that is consistent in structure with actual physical system structure. The structural equivalence may work well to help each node of the tree to overview amount of resources below them in order to reflect communication and memory access costs to kernel-level scheduling. Among resources, we especially focus on memory pages. Amounts of pages are maintained every hierarchy of the scheduling management tree and every processes to utilize in allocating resources to parallel processes. Resources are allocated by kernel-level scheduler to user-level scheduler and returned vice versa. The kernel-level scheduling is done by kernel-level scheduler (KLS). It is one of mechanisms or functions in SSS-CORE.

3.7 Kernel-Level Scheduling

SSS-CORE operating system is constructed as a time-sharing system. Time slices come when time-quantum expires. Since we target on large-scale NUMA system and aime at scheduling multiple parallel applications, the time-quantum will be set somewhere between 100 msec and a few seconds. It is ralatively large when constrasted to that of current operating systems, but it fits to the goal of our system. It is assumed that cost of kernel-level scheduling can be ignored because of the relatively large value of the time-quantum.

When a time-slice comes, every processor is preempted and the kernel-level scheduler starts. It first re-compute the priorities of the processes using dynamical information of resource allocation and resource usege of each process. Processes with higher priorities are scheduled first according to the scheduling information and constraints given by each of them.

We assume that processor power is not consumed on data transfering between physical memory and secondary memory. So disk accesses needed for the next scheduling is done between time-slices as much as possible. In addition to this, data on TLBs are saved on preemption and they are used as working set information. Data needed for scheduling are prefetched on time-slice previous to the actual scheduling time, and schedulinig is done one time-quantum ahead.

Details of kernel-level scheduling, with which the paper is concerned, is described in the followng chapters.

Chapter 4

Kernel-Level Scheduling

4.1 Role of Kernel-Level Scheduling

Scheduling is one of a very important part of general puropose parallel OS. It greatly affect the performance of the system because of the nature characteristic to parallel applications on multiprocessor system.

Evidently, accessing remote memory takes more cost than accessing local memory. And, in a cluster structured system, on which the model target on, accessing memories out side of a cluster takes more cost than accessing memory within a cluster. Obiously, large cost for accessing further cluster. Furthermore, accessing secondary memory storage is more and more costly.

Communication also takes time. And as the distance between the sender and receiver becomes larger, communication take more cost. In addition, if a lot of communication take palce at the same time witin a area, the network becomes too crowded for too many network transactions, which is always the case with busses. This causes communication to cost more than when interconection networks are not crowded.

Now, parallel applications, especially fine-grained parallel applications, have a lot of synchronization point in a execution among their multiple threads, sometimes between threads of different applications. As number of synchronization increases, the possibility of occurrences of above situations arises. Suppose the execution of one of

the threads of a parallel appplication is delayed, other threads waiting for the delayed thread to synchronize with it is also delayed.

The more severe the synchronization condition is, the degree of the delayed time becomes greater. When one synchronization is delayed, it affect the nexts one that is independent of the previous one. If this repeats in chain reaction among synchronizations, the time of whole exectution of the application is enormously delayed.

Therefore, to avoid delayed synchronization, it is essential to schedule threads of a single parallel application as close to each other as possible and also simultaneously, because of the conditions that; different cost in accessing different classes of memories; and contention in network transaction caused by conccurent access to memories or mulltiple communications.

Moreover, parallel applications too have locality of memory access in space and time. It is a property that a fragment of memory is frequently accessed and a piece of memory once accessed will soon be accessed again. So, scheduling an application onto processors where it has been previously scheduled is also important.

In summary, it is very critical to schedule the threads of a parallel application simultaneously, close to each other, and onto processors and memories where it have been schedules before as possible as the scheduler can manage. This gives importance to schedulling method.

However, some sorts of things differ between applications. For example,the frequency of synchronization and memory pages it requires through the execution sequence. In addition, the timing applications produce synchronizations, the palcement of memory pages when a memory access occurs, and etc, cannot be statically predicted by scheduler.

Our idea is that providing user-level an interface to exchange scheduling hints with kernel-level enables scrupulous resource allocation that satisfy each application's demand. Thus overall performance is enhanced and constructing generall purpose parallel OS becomes practical.

We call the hints scheduling constraints. To achieve previously mentioned scheduling properties, the scheduling constraints must include each application's request for resource usage, such as number of processors and memory pages. And also, memory migration must be carefully treated to manage the tradeoff between execution time acquired by a job being scheduled and accessing remote memory, for pages of waited jobs may be removed from the memories where it have been scheduled before.

The algorithm for kernel-level scheduling in the 2-level scheduling policy must work well to achieve these properties. But it is not simple to construct such an algorithm.

In following sections, examples for scheduling constraints and priority computation scheme are given to concretely cover the concept of kernel-level scheduling in 2-level scheduling policy. Furthermore problems when constructing a kernel-level scheduling method is shown.

4.2 Scheduling Constraints

Examples of scheduling constraints given to kernel-level scheduler from user-level are as follows. Notice that they are just ideas. Variation of constraints and their application to scheduling method is one of problems that must be cleared in future.

1. Constraints on number of processors
   (a) Fixed nmber of processors: a process is always scheduled to fixed number of processors if possible.
   (b) Desired number of processors: a process is scheduled to number of processors between upper limit constraint and lower limit constraint. Giving as many processors as possible is a natural idea, but the decision is left to scheduling algorithm.
   (c) Variable initail number of processors: processes just forked get variable number of processors. This may fit to fork-and-join type of applications.
2. Constraints on communication cost

(a) Fixed form of area: a process is always scheduled to processor area that is same if shape. This constraint can possibly be expressed by communication cost.

(b) Upper limit of communication cost: processors allocated to a process can communicate with each other within the cost given by the constraint.

3. Constraints on memory access cost

(a) Constraint limiting every memory access within a clluster. No pages will be allocated outside the cluster during execution of the process.

(b) Upper limit cost of access to remote memory: memory pages allocated to a process is accessable within the cost given by the constraint.

4. Constraints on memory migration

(a) Upper limit cost of memory access on migration: after scheduling process mifration can occur within the access cost the constraint indicates. In addition, forced-migration constraint can be given. If a process has not completed migrating when time slice has come, it can proceed migrating if it's migrating to memories accessable within the cost given by the constraint.

(b) Lower limit rate of ramainning pages: a process whose physical pages remain less than the limit will become a candidate for migration in scheduling.

4.3 Priority Computation Scheme

Concepts of scheduling constaraints are introduced to reflect process' demand for resources to kernel-level scheduling. But the resources must be fairly shared among processes. Therefore a adequate way for computing priorities must be defined. That is, as a process occupies larger amount of resources and imposes harder scheduling constraints, it's priority must be forced to age more. To meet our goal, following items

are used for re-computing priority. Here again, they are just ideas. Variation of items and adequate formula are left to be researched.

1. Product of the number and used time of occupied processors ($C_p n_p t$).
2. Product of the number of physical pages acquisited within the scheduled area and the time scheduled to the area ($C_m n_m t$).
3. Degree of difficulty of scheduling constraints (R_c).
4. Degree of satisfaction of scheduling constraints (S_c).
5. Product of the number of waisted processors and waisted time destributed to processes according to their used number of processors ($C_{wp} n_{wp} t \frac{n_p}{n_p}$).
6. The exsistence of waiting processes ($f_{wait} = 0\ or\ 1$).
7. Reduction rate for forced-migration ($f_{mg} = C_{mg}\ or\ 1;\ 0 < C_{mg} < 1$).
8. Product of waited time and return_time coefficient ($C_r(1-t)$).
   These are for making priorities of waited processes higher.

For example, using above terms, the aging value of the priority of a process can be computed by

$$(C_p n_p + C_m n_m) r_c s_c + C_{wp} n_{wp} \frac{n_p}{n_p})\ f_{wait} f_{mg} t - C_r(1-t).$$

4.4 Constructing a Kernel-level Scheduling

Some details about kernel-level scheduling are mentioned in the previous section. However, some of them lacked accurate definition, computing method, or explicit usage. They lacked because they are still unsolved. And also, we did not touch the matters that are more directly related to scheduling method such as scheduling algorithms, their implementation, data used for scheduling management and their destribution, and etc. In this section, we clear out what must be considered to construct a kernel-level scheduling system as well as the lacked stuffs.

4.4.1 Problems of Scheduling Method

For allocating resources, we arrange a virtual resource management tree that is consistent in structure with that of actual physical interconnection network. For a flat system a virtual scheduling management tree is used. We must define data structues for each cluster, or node of the management tree, so that effective scheduling algorithm can be made and using tree can be advantageous. For example, data structure for root of the tree may contain; (1)scheduling constraints of each process, (2)total resource in nodes just below root, and (3)number o free processors in nodes just below root, and that for tree node; (1)IDs of processes running below the node, (2)number of occupied processors for each process, (3)number of occupied total physical memory pages for each process running or waiting, and (4)number of free physical memory pages the node has. Data used for scheduling as well as tree node data must be distributed in the system because; usually system does not have general computing resource on the part that corresponds to nodes in the management tree; and system must be scalable. Data structure and destribution is essential also because they are collected down from tree leaves upto tree root on time-slices before actual scheduling computation to keep all the scheduling data consistent all over system. So defining a way for mapping these data onto processors is important. Moreover, the computation stage of kernel-level scheduler itself will be carried parallel for the sake of scheduling performance and because of data destribution. As mentioned in section 3.7, cost of kernel-level scheduling is ignorable because of relatively large value of the time-quantum. But scheduling performance must be efficient enough not to violate this assumption. And just as ordinary parallel applications, data destribution and parallel computation scheme must be discussed together.

4.4.2 Problems of Scheduling Constraints

There are some problems in expressing and using scheduling constraints showed in 4.2. Constraint on memory access cost when migrating is an example of constraint that have

expression problem, since the memory access cost it restricts is not clear for on which processors it is defined. The value of constraint that limits lower rate of remaining page for deciding when to migrate a process is up to each application, but ther possibly be a good value for it. The usage of each constraint is related to scheduling algorithm and must be carefully studied. Furthermore, a variation of constraints must be studied to find out which is useful in scheduling for there may be another constraints to be used.

4.4.3 Problems of Priority Computing

The example formula for computing priorities cannot be used without modification. For a priority might diverge and there is no gurantee of fairness in resource allocation. Moreover, the values of coefficients showed, method for computing degree of difficulty of constraints, and that for degree of satisfaction of constraints are not defined.

Computing priorities must be a good enough for fair-sharing resources among processes. There possibly be affinity between the priority computation methods and scheduling algorithms. A set of terms and a formula may well suite to a scheduling algorithm and another sets to another algorithms.

Chapter 5

Scheduling Algorithm in The Model

In the chapter, the algorithm of kernel-level scheduling implemented in the model is shown. Data prefetching is not conserned. Before showing the algorithm itself, parts that are needed when constructing a kernel-level scheduling are mentioned. First, some conditions of the program that must be noted are explained. Then scheduling constraints and priority computation scheme used in the model are described. These are very important because data that must be collected when scheduling depend on them. After that comes the algorithm. Though it is a very simple version with only a few scheduling constraints are concerned and cannot handle process migration, some considerations that can be extracted from the algorithm to refine it is given next. Finally we observe the kernel-level scheduler itself.

5.1 Conditions in the Simulation Program

Some data for emulating parts of the system are global to the simulation program.

Root of the scheduling management tree is declared as a global variable. This is because so far we are not concerned about mapping strategy of distributing data structures representing tree nodes amon processors, and the tree exist independently from data structures for processors and memories except leaf nodes. Each leaf node and processor data structures are in relation of one-to-one correspondence.

Since the time required for kernel-level scheduling is ignored for the relatively large value of time quantum, it is not included in execution time of whole system. In addition, during time slice, every processor is preempted and wait for a process to be scheduled until scheduling is completely finished.

As mentioned before, interconnection network of the model is not explicitly implemented. So tracing through pointers to tree nodes and function calls called with these correspond to communication or data moves through the network in practical system.

Process queue is also maintained as global data. Though scheduling data that must be renewd on every memory actions when processes are considered to be running are saved in the data structure included in the leaf nodes that corresponds to the processors where the processes are scheduled, data which are unique to processes are kept in the queue. Scheduling constraints of a process is one example of the type of data.

Another global data are that for managing memory system. Informations of every virtual page, such as those of home page and copy pages, are kept in a place. FIFO order of copy pages are kept there too for page replacement.

5.2 Scheduling Constraints

Scheduling constraints used in the model are,

1. constraint on number of processors—fixed number of processors,

2. constraint on communication cost—upper limits of communication cost.

Those that are easy to be treated in algorithm are selected. Constraints 2 is calculated from the level of the highest node in scheduling management tree that data pass when communicating or accessing remote memory within the cluster.

Using many, complicated constraints results in complex algorithm. Such an algorithm is too difficult to be inplemented from the beginning. That is why constraint on migration is not taken. Migration is not considered in the algorithm.

5.3 Data Structures and Scheduling Management Tree

Data structures for representing scheduling constraints, computing priorities of processes, and managing scheduling must be defined to construct a kernel-level scheduling.

5.3.1 Data Structure of Scheduling Constraints

The algorithm uses scheduling constraints showed in section5.2. The data structure is simple, a set of the constraints.

```
typedef struct constraint_t {
    int             ul; /* requested number of processors. */
    float           comul; /* communication upper limit */
} Constraint;
```

5.3.2 Data Structure and Scheduling Management Tree

An image of scheduling management tree with data structures of the model can be acquired in figure 5.1. Every node except root of the scheduling management tree holds following items.

- Level. A leaf node is 0 level and incremented by one as going up one step.
- Maximum cost of communication within the cluster represented by the node.
- Maximum cost of remote access within the cluster represented by the node.
- Total amount of processors and physical pages in the cluster represented by the node.
- Total amount of free processors and free physical pages in the cluster represented by the node.
- Array of pointers to child node.
- A pointer to parent node.

- Array of data structures to maintain scheduling information for each process.

Data structures for maintaining each process's scheduling information in these nodes include,

- processs ID.
- total amount of processors and physical pages a processes occupies below the node.

Data structure of root of the scheduling management tree is almost the same as that of other nodes. The contents of data structure to maintain each process's scheduling information is different from that of other nodes. And root has queues of running processes and waiting processes. A process is running when scheduled and waiting when not scheduled and no other process state is defined in the model.

Data structure for maintaining each process's scheduling information in root includes,

- processs ID.
- priority.
- data structure of scheduling constraints.
- waited time. If this is 0 then the process was scheduled all the time before the time slice.
- total amount of processors and physical pages a processes occupies throughout the system.
- informations required to express each proccess's address space, execution specification, and stuffs needed for process execution. the node.

Information except that of resources which are specific to processes is globally maintained as part of data in proess queues of root.

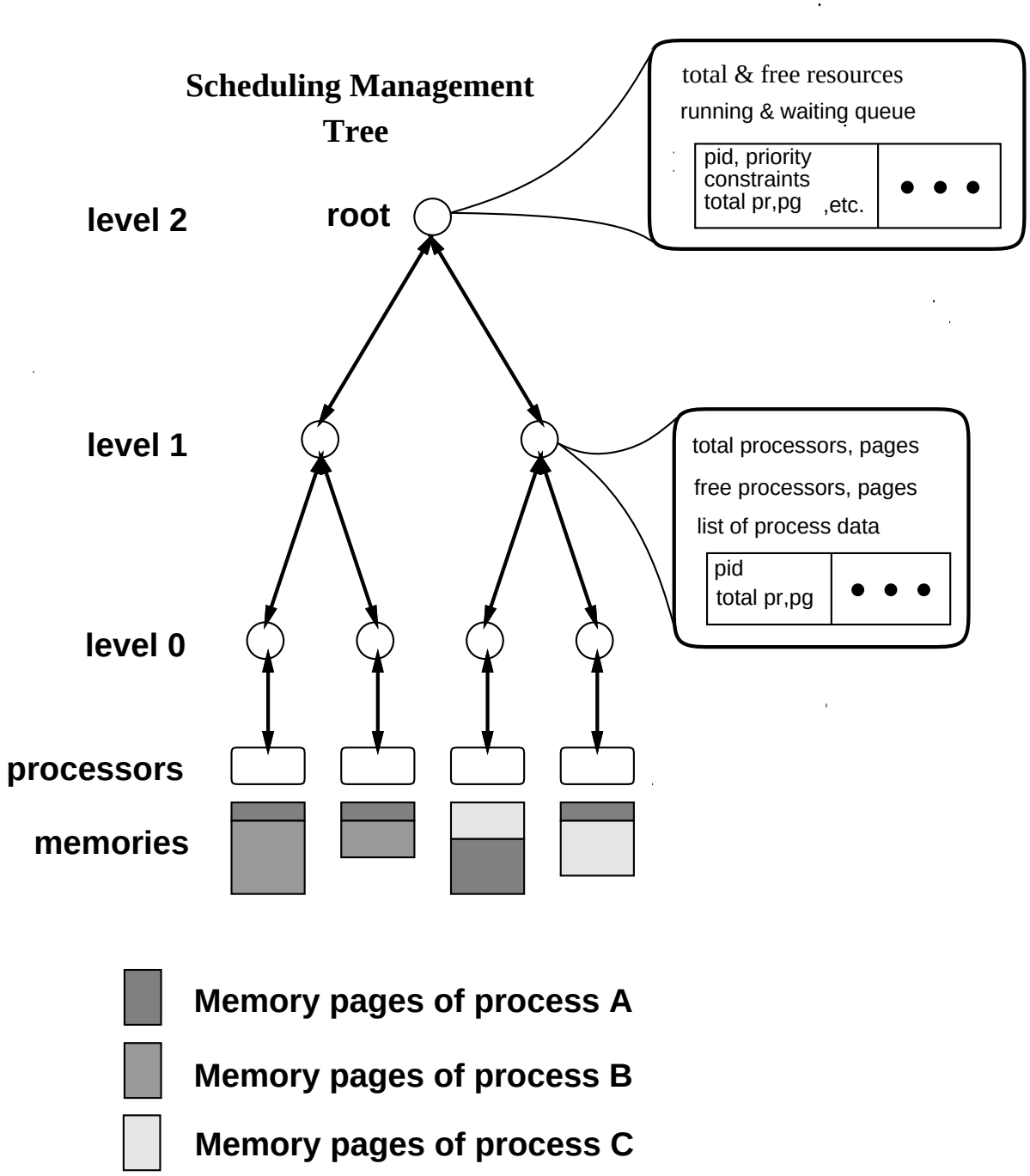

Figure 5.1: Model organization with data structures and 2-way 2-level scheduling management tree.

5.4 Priority Computation Scheme

Priorities are floating point data and calculated by following way.

5.4.1 Method for Computing Degree of Intensity of Constraints

Using the constraints (see 5.2), R_c is computed by,

$$R_c = \frac{\texttt{ul}}{\texttt{number\_of\_PEs}} * \frac{max\_communication\_cost}{\texttt{comul}}.$$

5.4.2 Method for Computing Degree of Satisfaction of Constraints

Processes that do not satisfy every constraints (see 5.2) is not scheduled by the algorithm. So, S_c is 0 or 1.

5.4.3 Parameters

When a process is forked, its initial priority is set to 0. Aging value calculated by following way will be added on each time slice.

Same terms that are proposed in section 4.3 except f_{mg} are used to calculate priorities. Using these terms, amount of resources used by a process is computed by

$$used\_resources = (C_p n_p + C_m n_m) R_c S_c.$$

See previous subsections for computing methos of R_c and S_c . The amount of resources waisted by a process is computed by

$$waisted\_resources = C_{wp} n_{wp} \frac{n_p}{n_p}.$$

Here, n_p is total number of processors used by all processes in the previous time quantum. It can be calculated with root data structure as,

$$n_p = \texttt{root->total.pr} - \texttt{root->free.pr}$$

Let t_{schd} represent the time the process was scheduled to the resources in fraction to the value of time quantum. That is, $0 \leq t_{schd} \leq 1$. If a process changes its phase (see

Coefficient	Value
C_p	1.0
C_m	1.0
C_{wp}	1.0
C_r	1.0

Table 5.1: Values used for coefficients for priority computing.

section3.5) before time slice comes, then $t_{schd} < 1$. With t_{schd} together with f_{wait}, the penal part of the aging value is computed by

$$(used\_resources + waisted\_resources)\, f_{wait} t_{schd}.$$

When scheduling, if there are one or more processes waited in the waiting queue during the last time quantum, and the process itself was scheduled for some time during that time, then the value make sense.

A process must profit from waiting. Using waiting time t_{wait} (naturally, $t_{wait} = 1 - t_{schd}$) of a process, the value,

$$retunrn\_time = C_r t_{wait}$$

will be subtracted from the aging value. But since the algorithm does not manage phase change, $t_{schd} = 1$ only if the process was scheduled, and $t_{wait} = 1$ only if the process waited. Otherwise, $t_{schd} = 0$ and $t_{wait} = 0$. In total, the aging value of a process is calculated by

$$aging\_value = (used\_resources + waisted\_resources)\, f_{wait} t_{schd} - return\_time.$$

Since priority is higher for lower value, aging value is added to old priority.

To avoid priorities from diverging, sum of aging value of processes (they are positive!) in running queue is evenly divided to processes in waiting queue. Coeficients C_p, C_m, C_{wp}, and C_r are parameters in the simulation program and must carefully be

set to addequately achieve the fair-sharing of resources among processes. But simple values showed in table 5.1 are used for initial experiments.

5.5 The Algorithm

The scheduling algorithm is literally expressed as follows.

1. Calculate values needed for computing priorities that are common to every process.
   (a) Number of waisted processors equals to the value of number of free processors kept in root data structure. It is renewd on every time slices and every phase change.
   (b) Number of processors used used by all processes (n_p) equals to difference between number of total processors and free processors kept in root data structure.
   (c) If queue of waiting processes is empty then $f_{wait} = 0$ else $f_{wait} = 1$
2. Correct resource information throughout the tree including root node. See figure.5.2.
   (a) Let each node's data of number of free processor equals to that of total number of processr.
   (b) Collect data of; each node's number of free pages; and each node's each process' number of total occupied processors and physical pages, down from leaf nodes up to root node. One scene from the step is showed in figure 5.2.
3. Compute and set new priorities to all processes. Information that must be calculated for each process is kept in their own data structure in root.
4. Merge the two queues, running queue and waitng queue, and sort by priority.
5. Select a process with highest priority and do next steps.
6. Set allocation mode. If the process is a new process or a process with no physical page, the allocation mode is NEW_PROCESS, otherwise NORMAL_PROCESS.

7. Strating from root of the scheduling management tree, move to the **home node** of the process. (A home node will be explained in the following part of the section.)

8. Try to **allocate** processors under the node. This is done by allocation algorithm shown in the following part of the section.

9. If failed, enqueue the process to waiting queue.
   If success, enqueue the process to running queue.

10. If there is no processors left, then put all processes into waiting queue. else, goto step 5 and select next process.

11. Finally, try to allocate failed processes that may migrate and new processes (processes who occupy no physical page in the system).

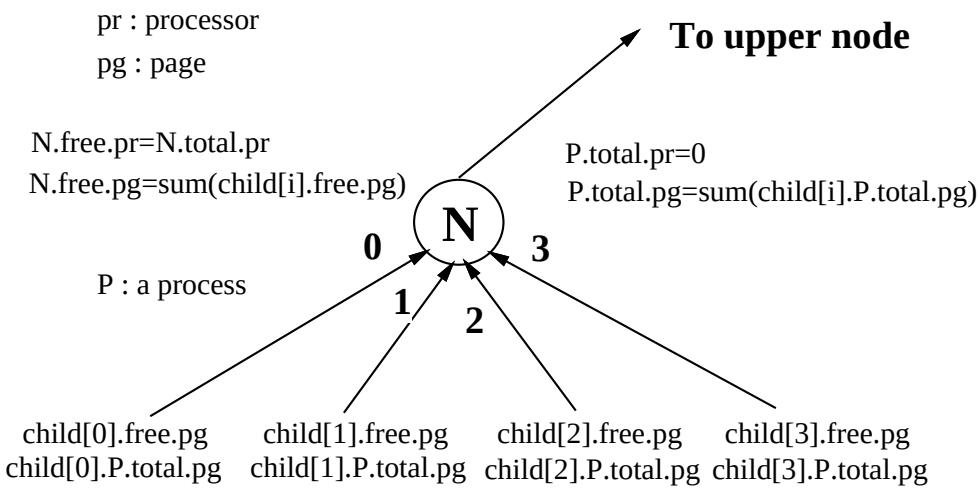

Figure 5.2: One scene from data collection at node N for process P.

Home Node Home node of a process is a node that includes all the processors previously it was scheduled in the subnetwork below the node. Figure 5.3 gives an

example. A simple algorithm used in the model for moving to home node is as follows.

1. If the node is a leaf (level = 0) then it's the home node. exit.
2. else
   (a) If the allocation mode (see step 6 of scheduling algorithm) is NEW_PROCESS, search child nodes in a fixed order and select a child that satisfies the constraints of the process with First Fit Algorithm. If no child could be selected, retun the node (it's the home node).
   (b) If the allocation is NORMAL_PROCESS, search among child nodes, in which the process has any physical page, in a fixed order and do the same as above.
3. Apply the algorithm to the selected child node.

Allocation Algorithm The algorithm for allocating a process starting from a node of the scheduling tree is literally expressed as follows. This algorithm is applied to a node in the tree called with number of processors as a request to it. It returns total number of processors allocated for SUCCESS, and 0 to imply UNSUCCESS.

1. Search the child nodes in a fixed order, for example left to right.
   (a) For a new process, (a process who occupy no physical page in the system) check each child and keep the number of free processors below it.
   (b) For other processes, check each child where the process occupy more than a page and keep the number of free processors below it.
2. If not enough free processors were found in total return 0.
3. Else, apply the algorithm to each child node with the number of free processors found in each, only if that's positive. (Avoiding total number exceeding the request, lesser of the number of processors found free in a child and what is left to satisfy the request is kept.)

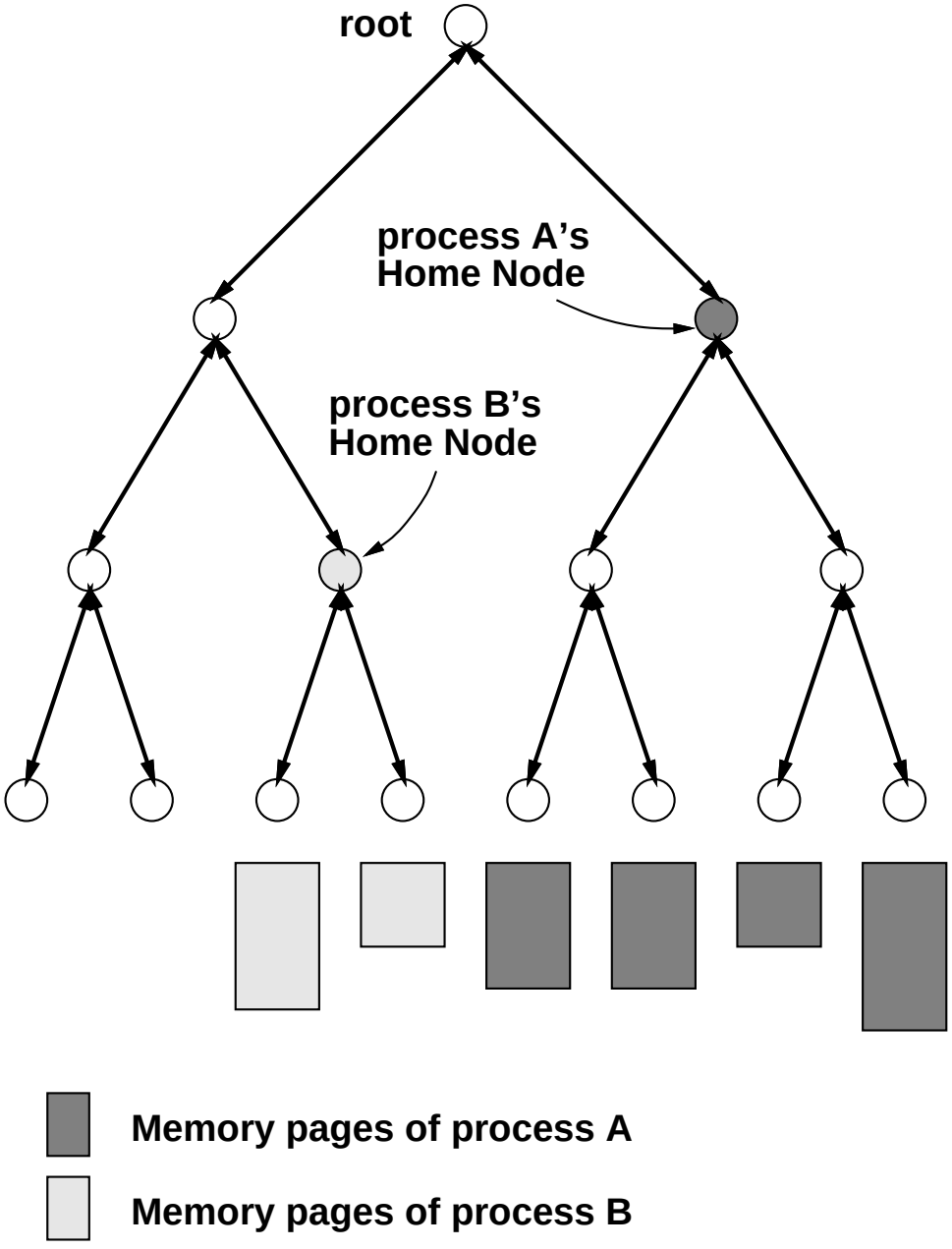

Figure 5.3: An example of Home Nodes.

4. Reaching this step implies allocation was wholly SUCCESS below the node. So, renew the information of free processor it has in its data structure.
5. Return the number of allocated processors to imply SUCCESS.

5.6 Possible Refinements of The Algorithm

The alogorithm showed is a simple version and process migration is not concerned. But carefully obaserving, some ideas or refinements possibly be taken into new version algorithms are found.

5.6.1 Problems when Checking Child Nodes

As the first step in allocation algorithm, child nodes are checked whether requested number of processors can be prepared in total. As for the step, two kinds of communication is required. One for requesting child nodes to check and the other for child nodes sending back the results of the checking to their parent node, the requesting node. Though they can be handles parallel, but when height of the tree, or the number of hierarchy in acctual system structure, is large, it may become bottleneck as for scheduling speed.

If a node have data of number of free processors of each child within its data structure, some of communication can be avoided. Checking child nodes is required only if they still have processors not allocated. So, as scheduling proceeds, many of the free processor data becomes 0 and redundant messages to child nodes who have no processor left to be allocated can be avoided. Correcting data in step **??** of scheduling algorithm must be slightly altered. That is, data of a child node in a node must be kept equivalent to data in actuall node.

Another idea is to let a node have every data needed to check child nodes. In this case, a communication is needed only when actually applying allocation algorithm to a child node. But those data include resource data of every process of all the child nodes.

So, the amount of that may not be ignorable.

5.6.2 Moving to Home Node

Step 7 of scheduling algorithm can be improved. The idea is to save the home node data of a process in process data in root node. Every time a process is scheduled, keep the home node information. Since a new process and a process that have no physical page in the system do not have their own home nodes, the algorithm of moving to a home node is applied to these processes. This way, we can move directly to the home node and start allocation right away. But in case that allocation was unsuccess, we must move up to have enough processors.

Chapter 6

Simulation Model

To estimate how effective the scheduling policy is, we made a model of a system and simulated on it. For preciseness, the model must reflect the practical situations as much as possible. But since we are trying to experiment on large-scale NUMA multiprocessor system that consists of several hundreds of processors at least, which may be too costly to simulate in a practical time, everything cannot be considered in the system from the very beginning of the studies. So far, not all of the features that SSS-CORE has is planning are implemented in the model.

6.1 System Model

6.1.1 Construction of Network System

The system is a NUMA multiprocessor characterized system that consists of clusters. The topology of the interconnection network of the system is k-way and n-level complete tree. Beginning from the root, the network branches into k different directions toward subnetworks, and this is repeated for n times. Each inner node represents a cluster. No processing unit is on inner nodes of the tree. On each of the end part of the network, that is, on each leaf node of the tree, single processor is connected. Processors are homogeneous and have the same amount of main memory and processing power.

The interconnection network is not directly implemented in the program so far. But

carefully choosing the values or computation method of parameters, the concept of difference in communication cost and remote memory access cost can be expressed, which is essential for the research of the paper. Furthermore, by altering the values of network parameters, differences in communication cost of parallel computers and worksation cluster system can be expressed.

Though SSS-CORE does not care about the actual hadware of the network system, clusters are assumed in the model to be connected by busses, which are cheap and most frequently used currently, especially for connecting workstations as well as in multi-processor systems.. But consequently, network transactions are not treated correctly. A bus might transfer more than one data, or messages, at one time.

6.1.2 Memory System

There are two important assumptions for memory system. One is about difference in memory access cost and other is about memory consistency method in the shared memory system. Hard disk is placed abstractly at somewhere in the system, where accessing it costs the same from each processr. It is too not explicitly implementesd in the model. Accessing hard disk is expressed by giving very lage value for the access. The order of memory access cost increases in the order of (1) in-cluster memory access, (2) cluster-cluster memory access, and (3) secondary memory access. This condition can be expressed in formula as,

$$(1) \ll (2) \ll (3).$$

A hardware mechanism to support shared memory system is assumed to be provided. Memories are treated by pages and pages are represented by sequencial virtual page numbers. FIFO is used as page replacement algorithm so far. The process that loaded the page to be replaced is called ower process. The owner process of each page must be maintained. This is because the total number of physical pages that a process owns is used in scheduling and must be kept precise.

The consistency of pages are managed by write-through policy. When a page is not found anywhere memory in the system, it is loaded from the disk and becomes home page, and the memory loaded it is called page home. IF a access to a page that are not in the memory occurs, the page is loaded from the page home of the page and the loded one is called a copy page. For a write to a page in a memory changing its contents, consistensy is somehow managed by the hardware and all the copy pages are kept valid. On page replace, If the page that is going to be replaced is a copy page, no action is taken. But if it is home page, then it is replaced and another copy page becomes home page in fifo order which is kept on loading.

Information on number of pages a process occupies in a memory is stored in the corespoinding data structure of leaf node of scheduling management tree. It is incremented when a page is moved into a memory and decremented when a page is removed from a memory.

6.2 Process Executoin Model

Processes are forked during a simulation in a fixed rate starting with random parallelism. Current model cannot cope with process destruction.

Execution of a process proceeds subject to clocks. In a one cycle of the clock, each processor may have both activities of input from other processors and an output to other processors. These activities are treated by a data structure called messages. A message mainly contains:(1) how many more clocks are needed to reach its destination; if this is 0, it's reached its destination, (2)type of the message that tells the received what action to take, and (3)ID of sender / receiver processor. Notice that the order of an input and an output at sender and receiver processors must not be reversed. So, process execution is devided into two steps, one for treating inputs (Step1) and another for outputs (Step2). Step1 is done at first for each processor and then Step2 is done.

Step1

1. If there is a message in receiving buffer whose time data is 0 (this implies that the message has arrived), then process the message according to its type. Their brief explanation are as follows.

   COM_MESSAGE Recieved a message to communicate. Change the processors' state to NORMAL.

   REQUEST_PAGE_HOME_READ Send back a copy page to reuested process. Rplace page when needed.

   REQUEST_PAGE_HOME_WRITE Same as above.

   READ_COPY_PAGE The requested page's copy arrived.

   WRITE_COPY_PAGE Same as above.

   HOME_PAGE The requested page arrived from the disk.

Step2

1. If the processor's state is NORMAL, go on to next steps. Otherwise, skip to 6.
2. If its time to communicate, then process communication and skip to step 6.
3. Selecet execution type uniformly from read, write, and no memory access. If it is no memory access, then increment the execution time of the processor.
4. Process the execution according to its type.
5. Send every message in the sending buffer.
6. Time data of every message is decremented if they are larger than zero.

Above steps are applied to each processors until next time slice comes. When time slice comes, the scheduler starts.

6.2.1 Memory Access Model

Address of a memory access is calculated randomly as follows. We prepare plural random nmbers that obey formal distribution with large valued variance to express locality. First, a choice is made and decided whether the access will be directed to the process' own address space or to that of other process'. If accessing other process, a process is selected uniformly among them. Then one of the distributions is selected uniformly. The average of the selected distribution is set at a fixed place within the address space of the destination process. Finaly the address is given randomly obeying the selected distribution.

6.2.2 Communication Model

The timing and partners of a communication is prepared for every processes beforehand. As soon as a communication occurs, messages are sent to other partner processors. That is, receiving processors are assumed to be desiring to have the message at the time it was sent. This implies that delayed time at receiving processor caused by memory accesses or another communication is not truely considered.

6.3 Experiment

Everytime a processor experiences an ordinary computation, that is, not waiting for communication completion or missed pages to come, its execution time is incremented by one. These values are summed when a simulation finishes. It can be regarded as total useful time all over the system spend. So, we focus on this value and make considerations according to it.

Chapter 7

Future Work

The simulator of current is a very simple version and many part of it must be improved to make it much closer to existing systems. The emulation must be as close to reality as possible to acquire reliable simulation resluts. In additon, it must be reformed to fit for experiments in which systems with quite large number of processors are concerned.

7.1 System Model

Current model uses FIFO for page replacement algorithm, but much smarter algorithm like LRU[6] must be implemented. And also, invalidate protocol must be introduced to memory model.

The model has no structure for interconection networks. So, contentions of memory accesses and network transactions are not considered in the model. This situation must be avoided to acquire reliable simultaion results.

A preferable way for expressing various process activities must be exploited. They include; rate of read / writes and their access pattern for processes; changing of phases; communication; and forking or destroying processes. Currently communication is treated as independent of processes and it must be implemented between processes or within a process.

7.2 Kernel-Level Scheduling

The algorithm presented in the paper is a very simple version and requires refinements followed.

One problem is that it considers only few scheduling constraints. More scheduling constraints must be studied to find out which is useful. Future algorithms must consider those constraints. As constraints change, the priority computation method must be changeds to fit the concept of resource fair-sharing.

Moreover, process migration is left to be included. When applying any kernel-level scheduling method to a practical system, it must be able to treat process migration.

The most important thing in scheduling method is that we must exploit a practical iterface to exchange scheduling constraints between use-level and kernel-level.

How the data prefetching affect the kernel-level scheduling is also a wonder. In SSS-CORE, it is assumed that processor power is not consumed on data transfering between physical memory and secondary memory. So, the scheduling is planned to be done with data one time slice before. We suppose the influence of it on system performance is less and the scheduling method is still effective. But this must be ascertained.

The research did not consider the actual mapping of data structures of scheduling management tree to processors. The tree is globally existing in the model. The computation of kernel-level scheduling itself is related to the mapping strategy. Since we assumed that cost required for scheduling is ignorable because of larger valued time quantum, the mapping and computation strategy must not violate the assumption.

Chapter 8

Conclusion

We proposed a kernel level scheduling method for a 2-level scheduling policy that takes advantage of informations on hierachical system structure and memory usage of each applicaton. In addition, studies remaining studies and problems that must be settled to construct a effective kernel-level scheduling method are mentioned.

Reference

[1] Songnian Zhou and Timothy Brecht. Processor pool-based scheduling for large-scale numa muliprocessors. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 25(5):133–142, February 1991.

[2] Anurag Kumar, Senior Member, IEEE, and Rajeev Shorey. Performance analysis and scheduling of stochasitc fork-join jobs in a multicomputer system. *IEEE TRANSACTION ON PARALLEL AND DISTRIBUTED SYSTEMS*, 4(10):1147–1164, October 1993.

[3] Thomas E. Anderson, Brian N. Bershad, Edward D.Lazowska, , and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 25(5):95–109, October 1991.

[4] Brian D.Marsh, Micheal L. Scott, Thomas J.LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 25(5):110–121, October 1991.

[5] Kai Li. Ivy: A shared virtual memory system for parallel computing. *1988 International Conference on Paralell Processing*, pages 94–101, September 1988.

[6] Abraham Silberschatz, James L. Peterson, and Peter B.Galvin. *Operating System Concepts*, chapter 8. Addison-Wesley Publishing Company, Inc., third edition, 1991.