Study on Kernel Level Scheduling in *SSS–CORE*: A General-Purpose Massively-Parallel Operating System

汎用超並列オペレーティングシステム *SSS–CORE* における カーネルスケジューリング方式の研究

by

Yojiro Nobukuni

信国 陽二郎

Master Thesis

Submitted to

The Graduate School of

The University of Tokyo

in Partial Fulfillment of the Requirements

for the Degree of Master of Science

in Information Science

February 5, 1997

Thesis Supervisor: Kei Hiraki

Title: Professor of Information Science

Acknowledgments

One day in our laboratory room, which is located at 4th floor in the building 7 of faculty of science, I finally faced with an incident. I was taking a break. I left the Ultra Parallel Project room and went back to the room 414 with a coffee cup in my hand. I wanted a cup of coffee, just as usual break time. When I stepped into room 414, it happened. HE WAS THERE. Pouring the very last spoon of coffee powder into his cup. A small bottle of ordinal instant coffee had been bought to fill our stomach. I came up to in front of his face but he never stop pouring hot water. He knew that I was there. Loughing and talking to me. When he has done with his project, he said “Well, I’ll give you back a bootle of good coffee beans.” What? If you are saying it, how come you didn’t use that beans for yourself and consume the maybe not good instant coffee that could have delight myself.

Well, he is my boss. He has done a lot of work to help my study and refine the paper. I must apprecite that truth. I must express my gratitudes to Mr. Matsumoto of our lab. The project could not be proceeded without his help. Finally, I thank all of my lab collegues, my friesnds and my family for supporting my life for I could spent few years at the department.

ABSTRACT

Execution performance of a parallel process in general-purpose NUMA systems is greatly affected by how resources are allocated to it through its lifetime. Concurrently running multiple parallel processes will exhaust physical memory. We propose two resource management mechanisms. One is a scheduling policy that reflects resource consumption states. A process is scheduled to clusters where it has physical pages. The other is a memory-replacement strategy based on page classification under distributed shared memory system. Shared copy pages of currently not running processes are first victimized. The performances of the two mechanisms are evaluated by a probabilistic simulation. It allows to simulate a variety of process sets and finite resources are manipulated with concrete management methods. The results show the superiority of our resource management mechanisms.

論文要旨

分散メモリ環境で並列プロセスの効率的な実行を妨げることなくマルチユーザ／マルチジョブ環境を構築するには、メモリページなどの実資源の使用状況を考慮したスケジューリングを行ない、システム全体の性能をあげることが有効である。また複数の並列プロセスが並行に動作する汎用的環境では、実メモリが溢れる場合を想定したシステム構築が求められる。分散環境では、参照頻度及び再アクセスのコストにより実メモリページを区別すれば、効率的なメモリ置換が可能である。

本研究ではメモリアクセスベースの確率モデル上で、具体的なメモリ管理方式／アクセス頻度／アクセスコストを付加したシミュレーションにより、並列プロセス毎に所有する実ページ情報を利用したスケジューリング方式、及びメモリ置換方式の評価を行う。今後は *SSS–CORE* へのプロトタイプ・スケジューラの実装を行なう予定である。

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

NUMA[30, 7] systems are characteristic for that they may achieve very high performance by making huge systems by simply connecting many processing elements.

There have been studies on supporting mechanisms[33, 8, 25] and optimization techniques for parallel applications[37, 24]. These techniques optimize execution of a single parallel application. However, in general-purpose environment, running every multiple parallel process with every resource allocation requirement satisfied is impossible. Running a process efficiently disturbs other process to run efficiently. Operating system level optimization is required to coordinate processes to run each process efficiently.

We propose two resource management mechanisms that allow to concurrently and efficiently run multiple parallel processes. One is a process scheduling policy that utilizes physical page usage information. The other is a memory replacement strategy based on page classification. The performances of these mechanisms are evaluated by using a detailed probabilistic simulation model.

Previous operating systems[15, 29] that allowed gang scheduling with dynamic repartitioning did not use resource informations and limited in scheduling flexibility. DHC[12] designed for UMA uses management structure close to ours but only uses load informations[11]. Our idea is to maintain the information on resource usage and operating system make decisions according to the information.

Chapter3 describes the resource management mechanisms. The methodology and the results of the simulation are given in Chapter4 and 5 respectively. We finally conclude in Chapter6.

Chapter 2

Related Works

2.1 Scheduling Systems

For efficient computing, simultaneously running frequently communicating threads is a relatively good scheduling policy. Gang scheduling runs all threads of a parallel process simultaneously. Many studies have showed the superiority of gang scheduling to other scheduling policies [20, 19, 16, 6].

Gang scheduling has been implemented on various systems. Cedar[17] is a multiprocessor with Alliant FX/8 systems connected to a shared memory. Xylem operating system on Cedar[9] uses global and local queues to implement 2-level scheduling. An Alliant FX/8 in Cedar is called a cluster. First level scheduling allocates clusters to tasks from the global queue. Scheduling within a cluster is done from local queues and parallel tasks that spans multiple clusters are not gang scheduled[26].

CM-5[18] partition the machine at hardware level. Processes can be gang scheduled within a partition. This allowed it to utilize various hardware supported mechanisms. CM-5 provided 'all fall down' mode to guarantee no activity on the network when multi-context-switching within a partition to implement gang scheduling.

Medusa operating system on CM*[29], the gang-scheduler of the BBN butterfly[15], and the gang scheduling runtime library of Makbilan operating system[14] implement the matrix algorithm[28] of gang scheduling with dynamic repartitioning. Rows of the matrix represents the scheduling slots and processors are time-shared by the rows. A row may contain several gangs. All of these implementations required coordination across groups of processors to context switch

threads simultaneously. The “Distributed hierarchical Control” (DHC) scheme[12, 13], designed for UMA systems, reduces this synchronization problem by partitioning with a buddy system. Each partition is associated with a logical controller. Upper controllers first schedules larger gangs and scheduling algorithm proceeds down to lower level controllers. Only at returning to the top of the scheduling cycle requires coordination of all processors. The controlling structure is close to *SSS–CORE*, but even a extended version of DHC[11] only uses load related informations. In addition, its partitioning scheme limits the partition sizes to powers of two.

The Intel Paragon[7] is a distributed memory multicomputer with a 2D mesh topology. Each processors run the OSF / 1 Mach-based microkernel with UNIX server[38]. It provides a hierarchical flexible partitioning scheme. Each partition has protection modes and scheduling characteristics associated. Gang scheduling is enabled within partitions with gang scheduling character. Scheduling decisions are made from upper level partitions like DHC. Higher priority applications are scheduled first at each level of the hierarchy and the scheduling algorithm continues down to higher priority sub-partitions. However, since applications with lower priorities must wait for higher ones to finish to be scheduled, it is hardly thought as fully time-shared system. (Unix) processes of an application on a single node is scheduled using standard Unix timesharing system. This feature is close to family scheduling described later.

In case of enough processors cannot be prepared, coscheduling[28] schedules only some part of all threads, that is the ’gang’, simultaneously. However, it is not clear how much beneficial scheduling part of a gang.

Family scheduling[5] is close to scheduling policy of *SSS–CORE* in the sense that scheduling is divided into two level. It is implemented in Mach microkernel on IBM RP3[5] and uses global and local queues to provide 2-level scheduling. Parallel processes are coordinated and mapped onto processors at first level scheduling. However, second level scheduling that decides the scheduling of process internal threads are also done by the operating system.

SSS–CORE lets a process to instruct the kernel to allocate variable number of processors. Process can schedule the subset of its threads as it prefers since second level of the scheduling is left to user-level in *SSS–CORE*. Process can internally preempt threads and can schedule threads of alternative groups. Grouping of process threads can more carefully be done at user-level than kernel-level. It is denoted as user-level scheduling in *SSS–CORE*.

There are other systems that implement 2-level scheduling. Mach[3, 2] uses partitioning

processors and a global shared queue within partitions. The operating system provides 2-level scheduling policies by partitioning and partition internal thread queues. Hector[32] connects clusters of processing modules by hierarchical ring network. Hurricane on Hector provides "processor-pool" based scheduling[40, 4] Its "Hierarchical Clustering"[35] scheme enables to schedule threads of an application close to each other with granularity of each level of operating system structure hierarchy. However, the threads are not gang scheduled and no resource related informations are used.

Affinity scheduling[36, 31] mostly used for UMA systems tries to schedule threads on the same processors on which they ran previously. This is done by using resource information on the amount of data remaining in caches. [1] However, the performance improvement was typically small for caches are relatively small sized and running some applications flushes most of working sets of other applications[34, 16, 36].

[1]Simple implementations however emulate this by boosting priorities of threads that have run on a scheduling target processor.

Chapter 3

Resource Management Mechanism

3.1 Scheduling Policy

3.1.1 Resource Management Tree

To take resource consumption state into scheduling account, resource related information must be managed. Our approach is to construct a data structure called *resource management tree* (RMT) to maintain system-wide resource usage and each process's resource consumption state.

RMT is virtually hierarchically structured to be scalable. The scalable nature enables to build a massively parallel systems. In addition, variants of parallel systems can be supported by adopting real structure of RMT to each specific systems, from workstation clusters to parallel super computers with flat networks. Scheduling decisions based on the structure naturally reflects the distances and hence the access costs between distributed resources. RMT has further advantages. It reduces the quantity of required physical resource for storing resource information. Bottlenecks of accessing the information is avoided.

Each node of the resource management tree logically holds information for resources seen below the node. They are number of processors and physical pages, number of total free processors and physical pages, and number of using processors and physical pages and ID of processes. The root node additionally has priority, scheduling constraints and home node of each process. Figure 3.1 shows an example image for a four processor system with RMT.

A process can achieve maximum performance by freely using allocated resources and by making the use of application level optimization. This can be done by using 2-level scheduling. The kernel allocates resources to each process by looking into the *resource management tree*. This

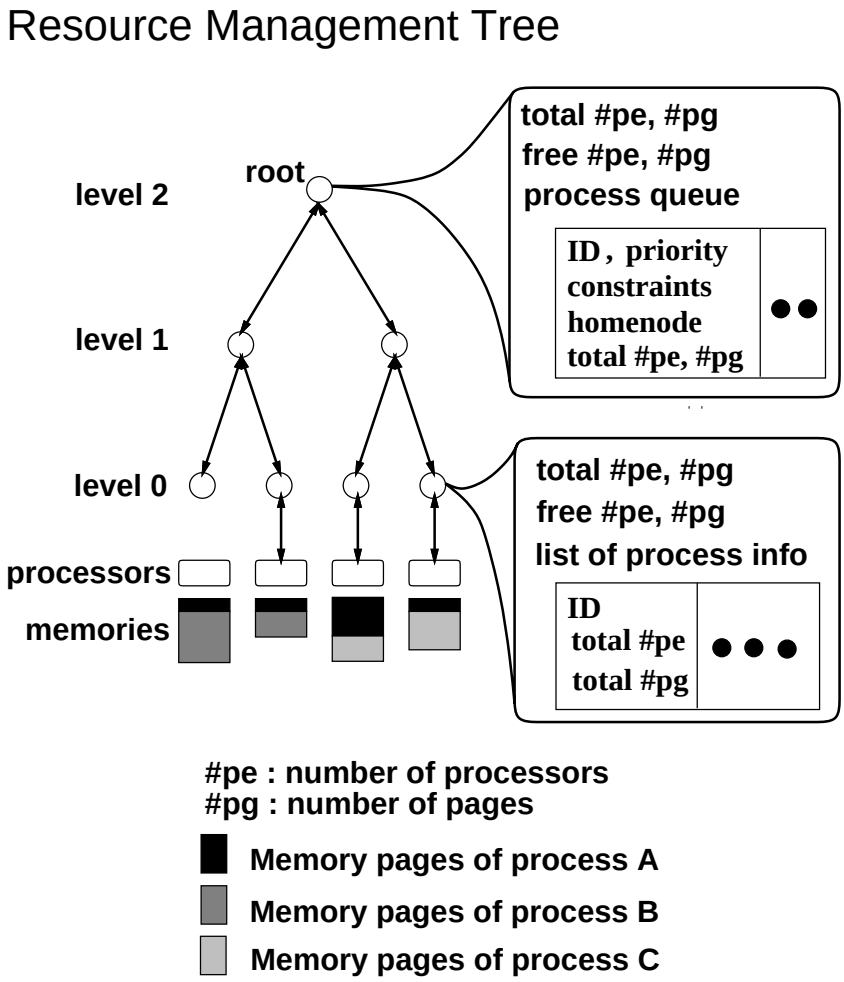

Figure 3.1: Resource Management Tree

way, resources that most fit for the use of a process can be allocated. The resource allocation within a process is left to user level scheduler. Each process freely re-allocate process internal resources.

3.1.2 Scheduling Constraints

A parallel optimizing compiler[37] generally assumes that system resources are used by a single application. Our goal is to run object codes these compilers output in a multiple user / multiple process environment. In such an environment, a process can achieve higher speedup when allocated resources satisfy its requirements and preferences. Scheduling constraints are used by processes to specify these informations to the kernel. The kernel follows the given constraints so

that the requirements of a process can be satisfied as much as possible.

A process can use scheduling constraints to specify its requirements of and preferences for the number of processors to use, communication cost between processors, memory access costs, and process migration. The *fixed processors constraint* expresses a constant number of processors a process requires. The *variable processors constraint* enables a process to be allocated variable number of processors. For a much parallelizable process which is programmed to be executed on variable number of processors, it works fine and reduces processor idle time.

3.1.3 Priority Computation

Since resources are allocated to satisfy each process requirements, a mechanism must be arranged to coordinate fair sharing of resources. Fairness can be maintained by managing priorities according to the amount of used resources and strength of given scheduling constraints and scheduling in priority order. Aging priorities according to these terms realizes fairness.

Priority is based on following values; (1)amount of used resources: U_r, (2)strength of scheduling constraints: R_c, (3)degree of constraints satisfaction: $S_c = 0\ or\ 1$, (4)amount of wasted resources: W_r, (5)presence of waiting process: $f_w = 0\ or\ 1$ The aging value of a process is computed from next expression.

$$aging = (U_r R_c S_c + W_r) f_w * t - C_r(1 - t)$$

Smaller the value, higher the priority. C_r is the aging coefficient. To prevent priority values to divert, the sum of the aging values of processes that were running before the time slice is equally divided and distributed to waiting processes.

3.2 Memory Replacement Strategy

Under general environment where multiple processes execute simultaneously, system must be designed to bear situations when physical memories are full. Selecting victim pages from those that are less frequently accessed and have less re-reference cost will enhance system performance.

Pages that belong to the currently running process are usually more referenced. Generally, local pages are more referenced than those shared by threads. Suppose a shared copy page has been replaced, it can be obtained with small cost through network from remote cluster. However, replacing pages that invoke disk accesses on re-reference can be a source of slow down.

Chapter 4

Simulation Methodology

4.1 SSS–CORE

The performance of the resource management mechanisms introduced are evaluated by simulating system with an operating system called *SSS–CORE.*

SSS–CORE[22] is a general purpose massively parallel operating system for NUMA parallel distributed systems. It provides multiple user / multiple job environment with timesharing and space partitioning. At the same time, it tries to achieve maximum performance of each parallel application.

For performance enhancement of the system under general purpose environment, coordination over processes and operating system level optimization is a must. *SSS–CORE* provides a mechanism that allows information transfer between kernel and user level. It supports *SSS–CORE* to realize the resource management mechanisms proposed in the paper.

SSS–CORE has more features. *SSS–CORE* implements distributed memory system on the framework called asymmetry distributed shared memory (ADSM)[23]. Efficient protected user-level communication is implemented[23]. Inter cluster synchronization of processors are not required to implement gang scheduling. These features are also assumed on constructing the simulation model.

4.2 Model Description

4.2.1 Overview

Operating system level simulation of parallel architecture by highly detailed model is practically impossible[27]. A simplified simulation model must be provided. To simulate operating system resource management mechanisms, amount of resources must have finite limitations. Memory must not be infinite sized to simulate memory-replacement strategies, for example. In addition, executing a particular suite of applications on the simulator is not enough for evaluating a general purpose operating system. Even instruction level simulation does not fit for the aim. We use a detailed probabilistic model for operating system simulation.

4.2.2 Interconnection Network

Network is not infinite resource. Network is one of resources shared by processes. Network defines the distributedness of resources. It will affect many system characteristics. For example, how much cost it will take to access a remote memory, what protocols will best fit to maintain shared pages consistent, and so on. Communication activities of a process within some part of the system may affect the activity of other process at the same part. The sharing pattern of network affects the system performance. Network must be modeled not to mislead to unreliable results.

The simplest modeling method is expressing network by a single cost parameter. Whatever cost it represents, it is not enough for the modeling for a few reasons. First, collisions on the network is not modeled. Sometimes a message blocks for another message passing the same network line or for short of message buffers. The acquired result will not include the impact of communication pattern. Second, large scale NUMA systems with hierarchical network cannot be simulated. Each level of those network may be constructed of different hardware cost, since higher level network tend to contention. It will easily become bottleneck if constructed with as little hardware cost as lower level network. Third, network sharing pattern cannot be modeled. When communication load at some part of the system becomes high temporary, activity of process running at the same part of the system will be affected, while processes at other part will still be able to communicate as usual. This network sharing pattern phenomenon is important for evaluating process scheduling performance in general environment.

The problem is in expressing network by a single cost parameter. The actually distributed

and separated network becomes shared by every process at every part of the system. This means incorporating load or busy parameter and calculating network cost from these values and the cost parameter is not enough.

We have constructed a tree structured switching network. Large scale NUMA system's network must be somehow hierarchical. RMT naturally fits to tree structured network. We do not have to consider the mapping of the RMT and worry about the affect of mismatch between the mapping and the underlining network structure.

A pair of a processor and a memory constructs a cluster. The nodes have buffers and do one-hop communication. The number of output buffer entry equals to the number of input lines to the node. There are three types of messages; page, update, and communication. Page message is for transferring copy pages and migrating pages. Update message is for use of update protocols of the distributed shared memory system. Remaining messages are the smallest messages and mostly used for transferring control. Synchronization and acknowledgment are this type of messages. The size parameter for each of these types of messages are given. The bandwidth can be given for each level of the network. The value a message actually takes for moving one-hop is computed from the basic transfer cost and the bandwidth of the network level where it is passing.

4.2.3 Process Model

A process has as many number of threads as the number of processors it requests. It uses the *fixed number of processors scheduling constraint* , and thus its parallelism never changes through its life time. Threads here denotes the execution context of a parallel process at a cluster. Each thread of a process has own local memory space and a shared memory space which is shared among threads of the process. Both memory space is provided with a reference frequency table that describes how frequently each page of the space is accessed. The ratio of the value on a entry corresponding to a page to the total frequency count of all the entries in a table describes the probability of an access to the page occurs within the space expressed by the table.

Executing a particular suite of applications on the simulator is not enough for evaluating a general purpose operating system. Exploring many types of processes suites is required. Preparing many type of processes for instruction level simulation is not easy because writing a new program must be involved.. Same can be said for real applications and it supports the simulation. Furthermore, we cannot describe the real characteristics of the applications executing from instruction

streams. We cannot describe to what particular class of process each application belongs. The biggest problem for instruction level simulation is in how we obtain the trace. Traces from systems other than our target environment cannot be used as it is to represent the application on our target environment. The timings of memory references and of synchronizations are different and instruction stream may differ when the application run on our system. Since we do not have real system implemented, we cannot obtain traces. If we have the system, we will not do the simulation. It is the “chicken and egg” problem.

Making memory reference addresses from a distribution curve is easier to prepare many types of processes. However, using a single distribution function is not flexible enough to express reference locality. Not to mention, it cannot be uniform distribution to model process execution that usually has locality in reference. Controlling the function and making particular part of the curve higher or lower is very hard.

We use per page reference frequency table to make memory reference addresses. Access locality can be expressed by giving higher scores at particular part of a process memory space. In addition, this makes much clear to what characteristics an application using the frequency table has as for memory reference. The disadvantage is that ordering of memory accesses cannot be expressed. Re-computing the distribution of frequency at each time tick[1] requires too much computing power. The distribution of reference rate to pages within a memory space will not change during a simulation.

The rate of access to local space and shared space is control-ed by per process parameter. This way, distribution of frequency rate within both spaces can be made independently. The type of an access, read or write or other activity like register computation, can be given for each page by the frequency table.

Process execution is clock-based probabilistic model. Processes make memory reference actions at each clock. With given interval of effective execution clocks, randomly selected threads of a process synchronize by a simple barrier Effective execution means the time or clocks spent for other than waiting for synchronization to complete or for memory access processing to end.

4.2.4 Memory Management

To simulate memory-replacement strategies, memory must be finite sized. Local pages and shared pages must be distinguished. It is important for modeling execution of parallel processes

on NUMA systems. Concrete management schemes for each type of pages affect the execution performance of parallel processes.

Pages of shared space are managed by distributed shared memory system. Shared write accesses give large impact on process execution efficiencies. Managing write accesses with concrete memory system enables to model

Sequential consistency memory model with an update protocol is used. Every write access starts update processing by sending update messages to every copy. The processor stops until it collects all acknowledges. To model NUMA systems, memory access cost and basic communication costs are set as to satisfy "local access $\ll$ inter-cluster access$\ll$ disk access". When threads change clusters on which it executes, its local pages are moved on-demand[10, 39] through network. Shared copy pages that do not reside on currently allocated clusters are removed without any cost. Accesses to unloaded virtual pages will cause disk accesses.

4.3 Scheduling Policy

The performances of five kernel level scheduling policies are evaluated. Every policy computes process priorities according to resource consumption state at each time slice and schedules processes with highest priorities. Processors are looked for within a particular area and allocated to a process if enough number of processors are found in the are. The policies are described below.

Policy0 allocates randomly selected requested number of clusters

Policy1 allocates requested number of continuous clusters in a fixed order

Policy2 first allocate clusters in home-node area where pages of target process exist, then clusters in whole area will be tried on failure.

Policy3 same as **Policy2**, but only home-node area is tried

Policy4 same as **Policy3**, but clusters that actually has target process's pages are allocated

The home-node of a process represents a subtree of the resource management tree that includes its requested number of processors. It somewhat corresponds to the area where the process was previously scheduled. Figure4.1 gives a home-node example. Process A in the figure, which has pages at marked clusters in area4, takes a node as its home-node which represents the subtree in area3. Area3 is called *home-node area* of process A.

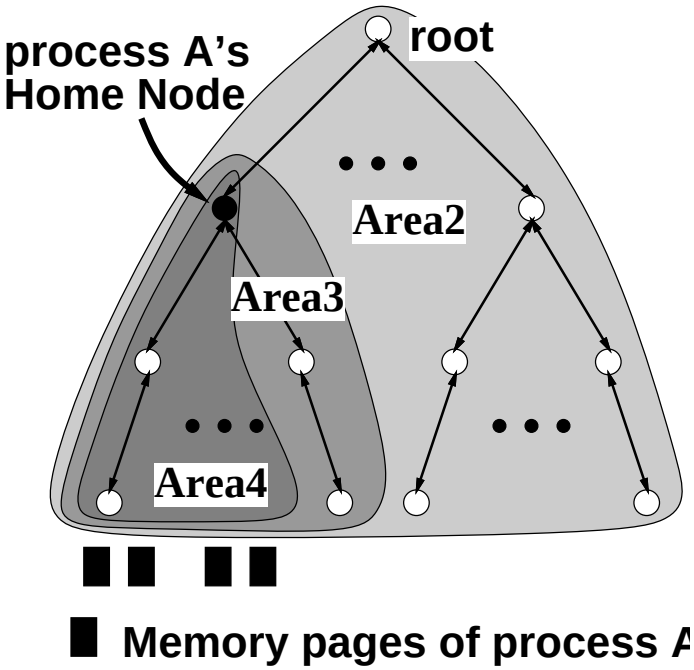

Figure 4.1: Scheduling Target Area

Policy2, **3**, **4** use resource management tree.

The difference among these three policies is in how much they persist in allocating processors from clusters where process's currently using physical pages are located. The difference is in the action taken when enough processors cannot be prepared by those clusters. **Policy2** will look for processors for all clusters. **Policy3** will only try to allocate from clusters in home-node area; the subtree below the home-node of target process. **Policy4** gives up scheduling the target process. Figure4.1 shows the area where each of policies look for processors to allocate. Area 2, 3, 4 corresponds to the area for **Policy2**, **3**, **4** respectively. **Policy4** mostly schedules a process to the same processors time to time. Chances that processes will be scheduled to clusters where they hold physical pages are greater in **Policy4**, **3**, **2** order. And, more processors may be utilized in reverse order.

Defining specific user-level scheduling policy for each process decreases simulation flexibility. A single policy is defined and used by all processes. It will schedule the identical threads to the processors that were also allocated to the process at previous allocation by the kernel level scheduler. Figure4.2 shows an example of thread scheduling. In the example, processors that are

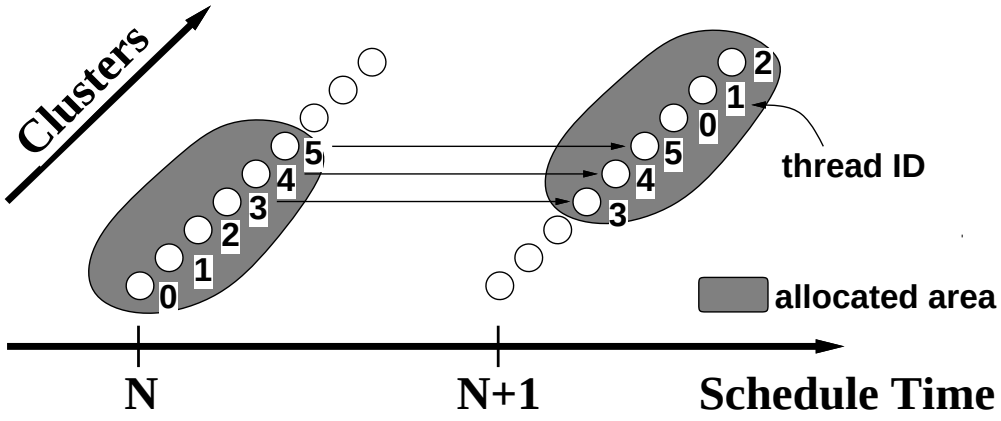

Figure 4.2: An Example of User Level Scheduling

allocated to the process by $(N+1)$st scheduling as well as (N)th scheduling will run threads 3, 4, 5 again. Processors that are newly allocated to the process at $(N+1)$st scheduling will run remaining threads (threads 0, 1, 2) in thread ID order. When thread is scheduled to different clusters, its local pages must be transferred through the network. Distributed shared memory system is responsible for properly transferring the shared pages of the thread. Clearly, the more overlaps in allocation area, the lesser the amount of page transfers.

Note that when time quantum is large enough, required time for computing scheduling itself is relatively small. *SSS–CORE* will use larger value for time quantum. The time required for scheduling is ignored in the simulation.

4.4 Page Replacement Strategies

Two page replacement strategy is evaluated and compared with each other.

Strategy0 Simple LRU without page classification

Strategy1 Uses page class. Processes are scanned in reverse priority order.

Assuming distributed shared memory system, memory pages can be classified into 6 groups by pointing whether a page; (a) belongs to currently running process or not, (b) is shared page or local page, and (c) has other copy pages or not. The page classes for **Strategy1** are; (1) copy page of not running process, (2) copy page of running process, (3) last one page of not running process,

(4) local page of not running process, (5) last one page of running process, and (6) local page of running process. “Last-one” page in the list means a shared page without any copy thus requires a disk access on next access. Ordering between classes 4 and 5 cannot be given trivially. Class 4 is prior to 5 in the list to maximize the efficiency of currently running process.

Since processes are scheduled in priority order, pages of lower priority process are possibly less referenced. **Strategy1** utilizes this characteristic. Both strategies will not select coherency processing shared pages as the victim page for replacement.

Chapter 5

Simulation Results

The parameters used in simulation are shown in Table 5.1. System with as many as 256 processors is evaluated. The topology of the network is three and four leveled tree structure. The former expands at root level into 4-way, then 8-way, and 8-way at the bottom level (**w488**). The latter expands 4-way at each level of the network(**w4444**). Table 5.2,5.3 describes the sets of parallel processes simulated.

Each experiment is carried on until one of the processes in a set stops execution [1] or 100 time slices has passed. The results are shown in Figure 5.1 through Figure 5.4. Graphs on the upper rows are results of **w488** system, and on the lower are of **w4444** system.

Each group in a graph is the results of scheduling **Policy0** through **Policy4**. Three left bars of a group are the results of **Strategy0** and the others are of **Strategy1**. Three evaluated values are plotted; (1)net effective execution rate, (2)calibrated effective execution rate. (3)maximum effective execution rate, and (1) is total efficiency of processors when processes are scheduled. Idle processors to which no processes are scheduled are not included. All processor idle times are accumulated to (3) and (2) is computed by following expression, (1) * (1.0 + idle time rate). Processor idle times are accumulated with the ratio of net effective execution rate.

5.1 Kernel Level Scheduling

Policy4 shows the best performance even when compared by net effective execution rate, which is disadvantageous for **Policy4** because processor idle times are not included. When compared

[1] This will be after 20 times time-quantum clocks of effective execution.

Table 5.1: Parameters and Costs

Parameters	Values
Number of processors	256
Disk access cost	10000 clk
Page transfer base cost	50 clk
Pages per cluster	400 pages
Page size	4096 Byte
Total memory	409.6 Mbyte
1 quantum	100000 clk

by other evaluations, the difference becomes larger. On real systems, processor idle time can be reduced by following two methods.

1. using *variable number of processor scheduling constraint* (will be introduced to *SSS–CORE*), processors can be flexibly utilized for number of processors.

2. un-allocated spaces caused by scheduling oriented processor fragmentation can be utilized by another process not included in a particular set of processes.

In case 1, the calibrated performance can generally be expected because processes will utilize the newly allocated processor space by *variable number of processor scheduling constraint* as much efficiently as they used the same space when allocated by *fixed number of processors scheduling constraint*. When an additional process is assumed for a particular set of processes, it can use the processors in formerly fragmented space as much efficiently as maximum effective execution rate, depending on its characteristics as a parallel process. Thus maximum effective execution rate can be expected for case 2. Evaluating cases for variant process sets other than those experimented is inevitable and important for describing the performance of general purpose operating system and prospecting how the performance of *SSS–CORE* will be. Comparison by calibrated or maximum effective execution rate is validated from this point of view.

The results shows that be more inclined to scheduling processes to where they have their own pages, the greater performance achieved. Considering set A, where memory replacement

Table 5.2: Process Sets (a)

Process Sets	number of processes	parallelism (number of processes)
A	12	16,36,48,50(2),64,70,96,100,128,192,200
K	12	48,96,208,256
L	11	64(5),128(2),192(2),256(2)
M	23	16(16),50(6),256(1)

Table 5.3: Process Sets (b)

	Process Sets			
Items	A	K	L	M
number of processes	12	12	11	23
total paralelism	1050	1184	1472	812
local space size	35500	49920	68480	35480
shared space size	10120	6360	7280	30320
physical page requirements	1.00	1.30	1.68	1.70
syncronization interval	1000–2000	1000	1000	1000

quantity is small [2], **Policy4** is the best in efficiency by any of the three estimations. The quantity of page transfer is larger among **Policy2**, **3**, **4** in the order. Table 5.4 shows that the time spent in synchronization or communication get larger for policies in the same order. This comes from taking reduction of the impact of transferring local pages, which are generally referenced frequently, into scheduling account. Changing processor allocation space time to time cause each memory to be filled with pages from many processes. When **Policy2** or **Policy3** is used, processes scramble for the physical pages and result in lower in efficiency than **Policy4**.

5.2 Memory Replacement Strategies

Strategy1 always outperforms **Strategy0**. Table 5.5 is the breakdown of replaced counts for each class of pages. Results of the scheduling **Policy4** on **w488** system is shown.

As for **Strategy1**, mostly copy pages are replaced. Process sets A and K, which impose small

[2]Some memory become full because of the processes overlapping each other.

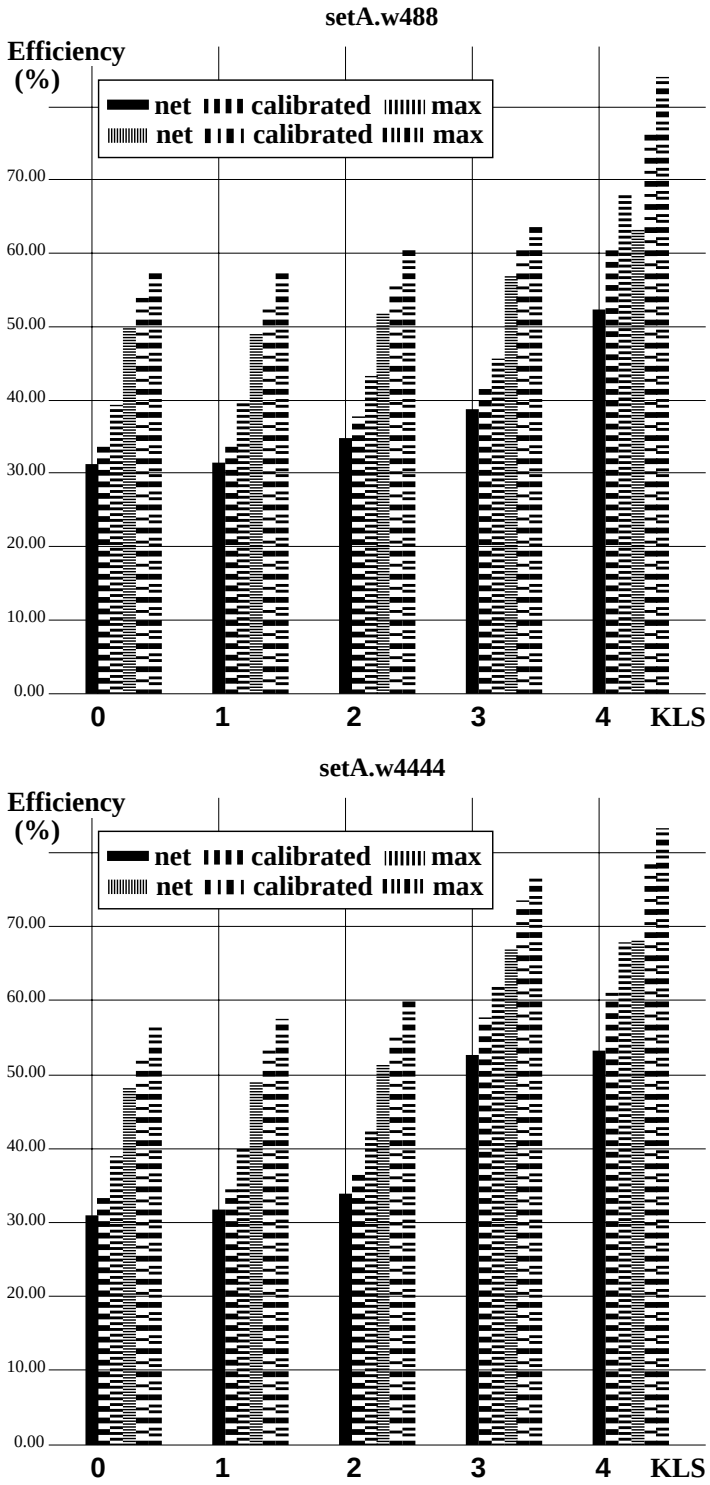

Figure 5.1: Results of Process Set A (**w488**, **w4444**)

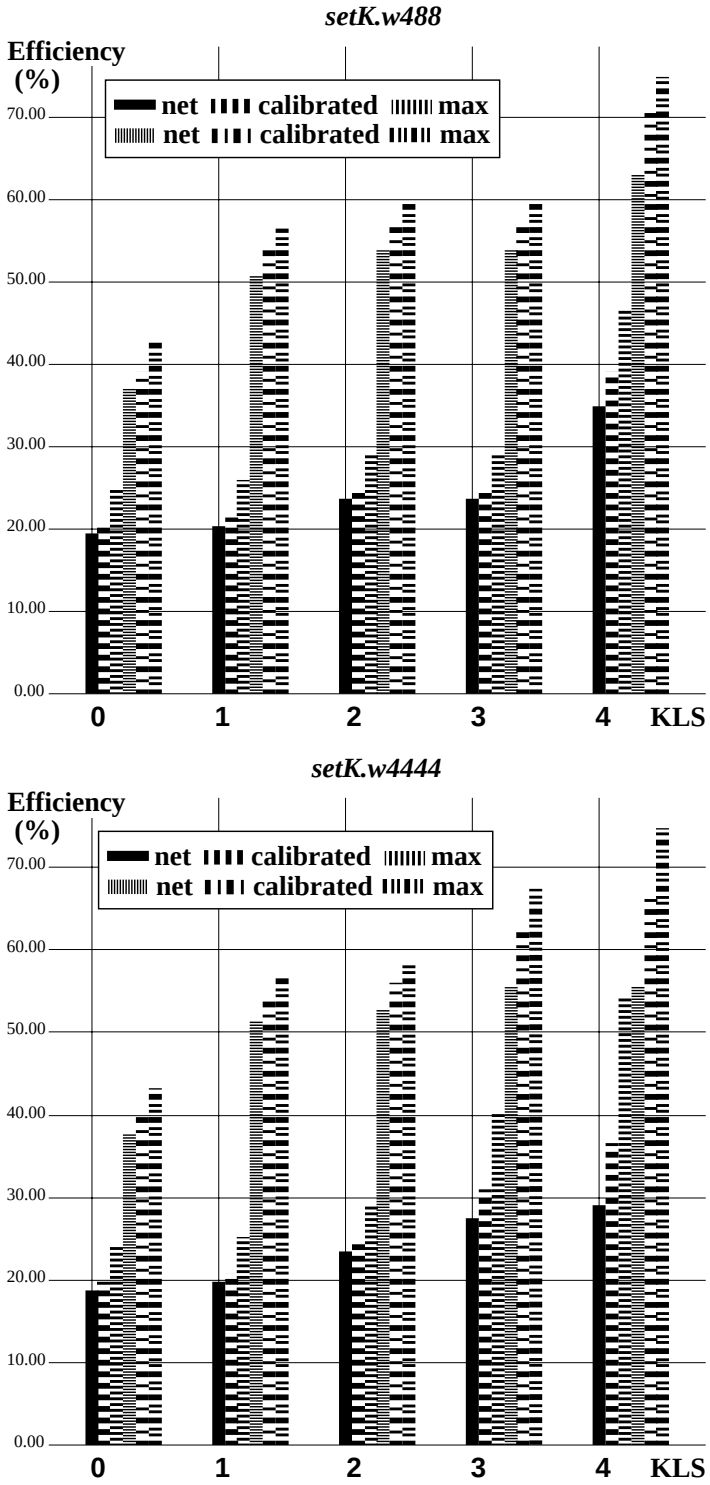

Figure 5.2: Results of Process Set K (**w488**, **w4444**)

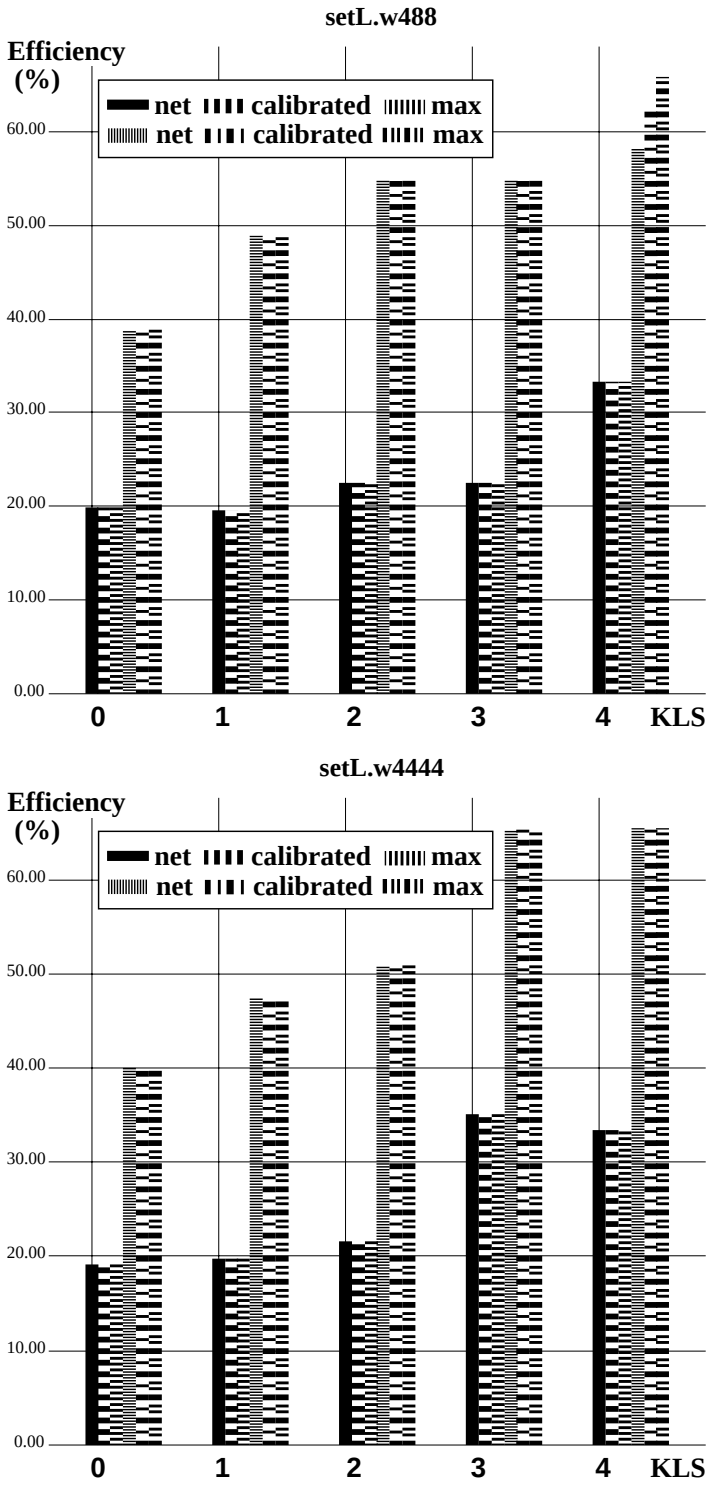

Figure 5.3: Results of Process Set L (**w488**, **w4444**)

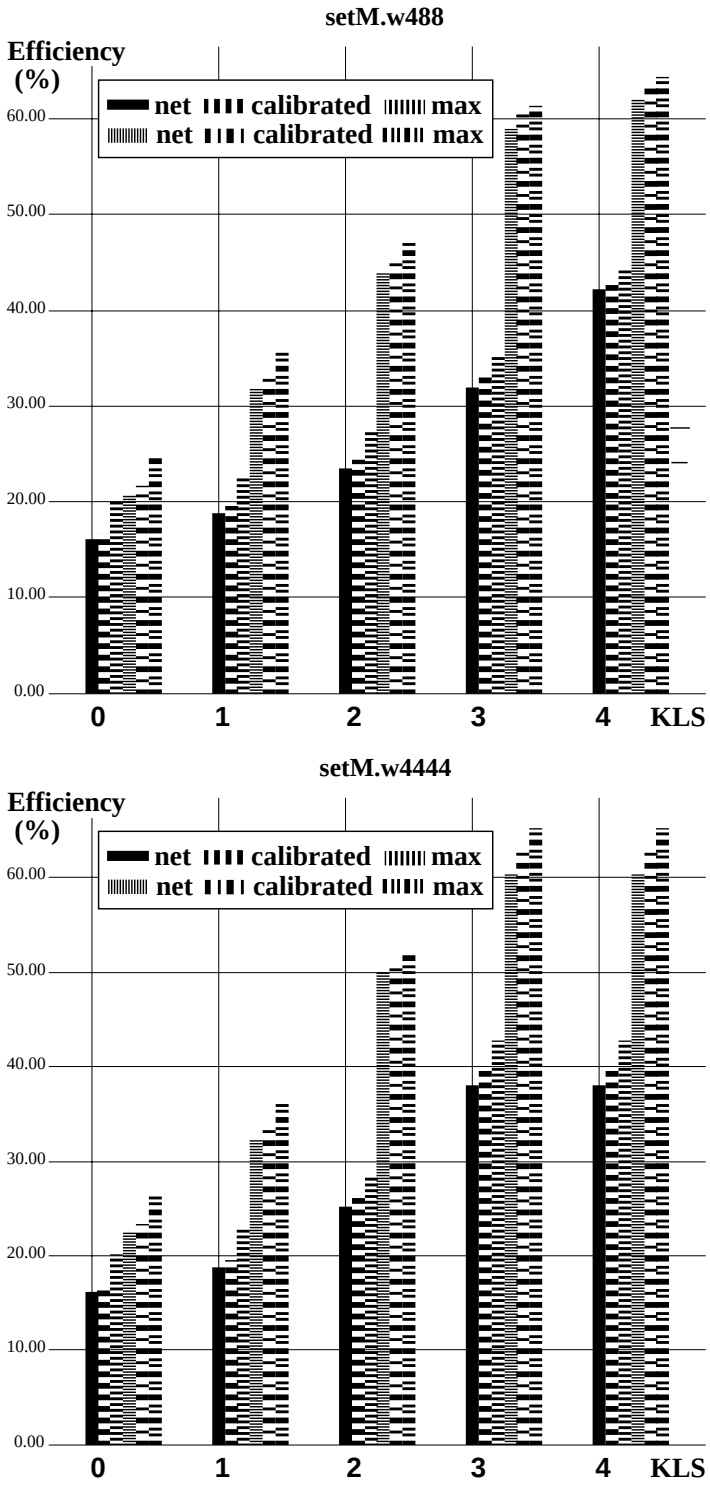

Figure 5.4: Results of Process Set M (**w488**, **w4444**)

Table 5.4: Execution Time Breakdown(%)(**w488**, **Strategy1** replacement)

Process Set	Type	Scheduling Algorithm				
		Policy0	**Policy1**	**Policy2**	**Policy3**	**Policy4**
A	Max	57.89	57.36	60.41	64.22	84.02
	Sync	19.98	19.59	20.84	20.03	12.45
	Comm	22.10	23.01	18.72	15.70	3.49
K	Max	42.65	56.57	59.53	59.53	74.91
	Sync	25.87	22.08	23.24	23.24	17.82
	Comm	31.46	21.32	17.20	17.20	7.22
L	Max	38.81	48.93	54.98	54.98	65.85
	Sync	28.52	24.07	23.58	23.58	22.23
	Comm	32.64	26.96	21.39	21.39	11.87
M	Max	24.79	35.73	47.42	61.47	64.37
	Sync	22.83	19.03	18.68	16.57	16.06
	Comm	52.36	45.21	33.86	21.90	19.51

Disk assessing time is not included.

physical memory requirement, see only copy pages of not running processes victimized. But for **Strategy0**, local pages and last-one shared pages are replaced.

The results show that selecting victim pages according to the classification enhances system performance. Even when the system is somewhat highly loaded, efficiency is preserved by not kicking local pages that are more frequently referenced out of memory.

5.3 Considerations on Realizability of General Environment

Maximum efficiencies of the results are roughly between 65% to 85% for the experiment of scheduling policy **Policy4** and **Strategy1** replacement strategy pair. The lower results come from sequential consistency memory model. The time waiting for preceding accesses to complete is very large. Update processing time is not very large compared to the waiting time. Cooperating more relaxed memory model and lighter consistency managing system solves the problem. In addition, performances of parallel applications with large shared access frequencies can be enhanced with

Table 5.5: Replaced times for each class of pages(**Policy4**, **w488**)

Page-Replacement Strategy	Process Sets	Page Class					
		other's copy	own copy	other's last	other's local	own last	own local
Strategy0	A	10944	4770	723	4763	303	1130
	K	33105	10487	830	22598	65	3803
	L	92253	18063	4911	63885	136	9978
	M	46394	23581	6873	11725	4054	6577
Strategy1	A	38976	0	0	0	0	0
	K	77665	0	0	0	0	0
	L	414287	0	0	0	0	0
	M	245705	11578	0	0	0	0

various compilation techniques and by introducing useful communication techniques, such as hierarchical multicasting and acknowledge combining. Thus, lower simulation results does not negate general environment on parallel distributed system.

Chapter 6

Conclusion

The paper has described kernel level scheduling policy that uses information of resource consumption state and memory replacement strategy that uses page classification upon distributed shared memory system. The performances of various methods for these mechanisms are evaluated by simulating on detailed probabilistic model.

As for the kernel level scheduling, the simulation results showed the superiority of those policies that use resource management data structure and allocate processors of clusters with memory affinity to a process. Replacing pages according to the page classification saw the system outperforming by far against when pages are replaced in simple LRU order without the classification. When these two mechanisms are incooperated together, the effective execution rate of the system is higher than 65% for highly loaded cases. This value can be much raised by using more sophisticated mechanisms, such as released consistency memory models, hierarchical multicasting, and acknowledge combining[21].

The results of simulation given in the paper showed the possibility of practical operating system that provides highly effective general environment on NUMA systems. A prototype operating system *SSS–CORE* is currently under development on workstation clusters.

References

[1] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. on Computer System*, Vol. 4, No. 4, pp. 273–298, November 1986.

[2] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, Vol. 23, No. 5, pp. 35–43, May 1990.

[3] D. L. Black. Processors, Priority and Policy: Mach Scheduling for New Environments. In *Proc. Winter USENIX Technicl Conf.*, pp. 1–12, January 1991.

[4] T. Brecht. On the Improtance of Parallel Applications Plcacement in NUMA Multiprocessors. In *Proc. of the 4th Symp. on Experiences with Distributed and Multiprocessor Systems*, pp. 1–18, September 1993.

[5] R. M. Bryant, H-Y. Chang, and B. S. Rosenburg. Operating systen support for parallel programming on RP3. *IBM J. Res. Dev.*, Vol. 35, No. 5/6, pp. 617–634, Sep/Nov, 1991.

[6] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *ACM, ASPLOS VI*, pp. 12–24, October 1994.

[7] Intel Supercomputer Systems Division. *Paragon User's Guide*, order number 312489-003 edition, June 1994.

[8] F. Douglis and J. K. Ousterhout. Process Migration in Sprite Operating System. *Proc. of the 7th Inter. Conf. on Distributed Computer Systems*, September 1987.

[9] P. A. Emrath, M. S. Anderson, R. R. Barton, and R. E. McGrath. The Xylem Operating System. In *Intl. Conf. Paralell Processing*, Vol. I, pp. 67–70, August 1991.

[10] M. R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems. In B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, pp. 414–424. IEEE Computer Society Press, 1995.

[11] D. G. Feitelson. Packing schemes for gang scheduling. In *Proc. IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 54–66, April 1996.

[12] Dror G.Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, Vol. 23, No. 5, pp. 65–77, May 1990.

[13] Dror G.Feitelson and Larry Rudolph. Mapping and Scheduling in a Shared Environment Using Distributed Hierarchical Control. *ICPP*, Vol. I, pp. I1–I8, August 1990.

[14] Dror G.Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, Vol. 16(4), pp. 306–318, December 1992.

[15] B. C. Gorda and E. D. Brooks III. Gang Scheduling a Parallel Machine. Technical Report UCRL-JC-107020, Lasrence Livermore NL, December 1991.

[16] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *ACM SIGMETRICS, Conference on Measurement & Modeling of Computer Systems*, pp. 120–132, May 1991.

[17] J. Konicek and et al. The Organization of the Cedar System. In *Intl. Conf. Parallel Processing*, Vol. I, pp. 49–56, August 1991.

[18] C. E. Leiserson, A. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hills, B. B. Kuszmauk, M. A. St. Pierre, D. S. Wells, M. C. Wong, S-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *4th ACM Symp. Parallel Algorithms and Architecutres*, pp. 272–285, June 1992.

[19] S. T. Leutenegger and X-H. Sun. The performance of multiprogrammed multiprocessor scheduling policies. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, pp. 226–236, May 1990.

[20] S-P. Lo and V. D. Gligor. A comparative analysis of multiprocessor scheduling algorithms. In *7th Intl. Conf. Distributed Comput. Syst.*, pp. 356–363, September 1987.

[21] T. Matsumoto and K. Hiraki. A Shared Memory Architecture for Massively Parallel Computer Systems. *IEICE Japan SIG Reports*, Vol. 92, No. 173, pp. 47–55, August 1992. (In Japanese).

[22] T. Matsumoto, S. Huruso, and K. Hiraki. General Purpose Massively Parallel Operating System SSS–CORE. In *Proceedings of 11th Japan Society for Software Science and Technology*, pp. 13–16, October 1994. (in Japanese).

[23] T. Matsumoto, S. Uzuhara, and K. Hiraki. Asynmetry Distributed Memory System with Memory Based Communication. In *Proceedings of Japan Society for Software Science and Technology*, pp. 37–44, November 1996. (in Japanese).

[24] Takashi Matsumoto. Synchronization mechanisms and processor scheduling on multiple processors. *IPS Japan SIG Reports*, pp. 1–8, November 1989. (in Japanese).

[25] Takashi Matsumoto. Elastic barrier: Generalized barrier synchronization mechanism. *Trans. of IPS Japan*, Vol. 32, No. 7, pp. 886–896, July 1991. (in Japanese).

[26] C. Natarajan, S. Sharma, and R. K. Iyer. Impact of Loop Granularity and The Self-Preemption on the performance of loop parallel applications on a multiprogrammed shared-memory multiprocessor. In *Intl. Conf. Parallel Processing*, Vol. II, pp. 174–178, August 1994.

[27] Y. Nobukuni, T. Matsumoto, and K. Hiraki. Simulation Models for Performance Estimation of Parallel OS. *IPS Japan SIG Reports*, Vol. 96, No. 23, pp. 19–24, March 1996. (In Japanese).

[28] J. K. Ousterhout. Scheduling techniques for concurrent systems. *3rd Intl. Conf. Distributed Computer Systems*, pp. 22–30, October 1982.

[29] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: an experimant in distributed operating system structure. *Comm. ACM*, Vol. 23, No. 2, pp. 92–105, February 1980.

[30] J. Palmar and Jr. G. L. Steele. Connection Machine model CM-5 system overview. *4th Symp. on Frontiers Massively Parallel Comput.*, pp. 474–483, October 1992.

[31] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, pp. 131–143, February 1993.

[32] M. Stumm, Z. G. Vranesic, R. White, R. Unrau, and K. Farkas. Experiences with the Hector Multiprocessor. In *Proc. Intl. Parallel Processing Symp. Parallel Systems Fair*, pp. 9–16, January 1993.

[33] T.E.Anderson, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Proc. of the 13th ACM Sympo. on Operating Systems Principles*, Vol. 25, No. 5, pp. 95–109, October 1991.

[34] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *J. Parallel and Distributed Computer*, Vol. 24(2), pp. 139–151, February 1995.

[35] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *USENIX*, 1992.

[36] Raj Vaswani and John Zahorjan. The implication of cache affinity on processor scheduling. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 26–40, October 1991.

[37] R. P. Wilson and et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, Vol. 29, No. 12, pp. 31–37, December 1994.

[38] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proc. Winter USENIX Conf.*, pp. 449–467, January 1993.

[39] E. R. Zayas. Attacking the Proces Migratino Bottleneck. In B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, pp. 433–444. IEEE Computer Society Press, 1995.

[40] Songnian Zhou and Timothy Brecht. Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors. In *Proc. of the ACM SIGMETRICS, Conf. on Measurement and Modeling of Computer Systems*, pp. 133–142, May 1991.