Data Sharing Conscious Scheduling for Multi-threaded Applications on SMP Machines

Shlomit S. Pinter and Marcel Zalmanovici

IBM Haifa Research Lab, Haifa University, Mount Carmel,
31905 Haifa, Israel
{shlomit, marcel}@il.ibm.com

Abstract. Extensive use of multi-threaded applications that run on SMP machines, justifies modifications in thread scheduling algorithms to consider threads' characteristics in order to improve performance. Current schedulers (e.g. in Linux, AIX) avoid migrating tasks between CPUs unless absolutely necessary. Unwarranted data cache misses occur when tasks that share data run on different CPUs, or are far apart time-wise on the same CPU. This work presents an extension to the Linux scheduler that exploits inter-task data relations to reduce data cache misses in multi-threaded applications running on SMP platforms, thus improving runtime, memory throughput, and energy consumption. Our approach schedules the tasks to the CPU that holds the relevant data rather than to the one with highest affinity. We observed improvements in CPU time and throughput on several benchmarks. For the Chat benchmark, the improvement in CPU time and cache misses is over 30% on average.

1 Introduction

The demand for greater computing capacity has lead to an increased use of multiprocessor machines. Symmetric multi-processing (SMP) is a specific implementation of multiprocessing in which multiple CPUs are physically connected via a common high-speed bus and share resources such as memory, peripherals, and OS. With the rise in the number of parallel multi-threaded applications, the popularity of SMP has increased as well because it provides a way to utilize the application level parallelism for performance gain.

Schedulers in many operating systems, such as Linux, UNIX and AIX, implement variations on processor affinity thread (task) scheduling. The observation behind this choice is the desire to reuse data (and instructions) remaining in the processor's cache from a previous dispatch of the thread. We observed that in SMP machines unnecessary cache misses occur when tasks that share data run on different CPUs.

Several studies have examined the affinity of a task to a processor based on how fast a task can run on a processor in heterogeneous processor environments [4] [7] [15]. Those studies provide a variety of algorithms for different timing metrics and conditions for scheduling tasks to CPUs. Another type of processor affinity looks at the resources bound to processors [8] [15]. These studies attempt to optimize a certain

metric under constraints, e.g. the task must execute on a processor that has access to the required resource. Another type, which has received far less attention, is based on tasks' affinity to cache contents, i.e., data affinity. The difficulty in utilizing such affinity is in its dynamic nature. Our work suggests and measures new methods that produce scheduling based on data affinity information.

The potential for performance improvements from exploiting data affinity is disputed. Squillante's theoretical work and simulations [14] have exhibited promising potential. Conversely, Gupta [6] in his simulations and Vaswani [12] in his measurements and tests have concluded that exploiting data affinity has a negligible effect for multi-threaded applications. These results should be rethought in view of architectural advances and the ever-growing use of multi-threading in today's applications.

We found that most applications can benefit from data affinity, regardless of the pessimistic claims mentioned above. Applications consisting of long-living, frequently synchronizing, and memory-intensive threads benefit the most. Moreover, the steady growth in cache sizes implies that a large portion of a task's data will reside in processor's cache, allowing optimizations to ignore the exact data access patterns of threads, thus simplifying data affinity based optimizations. Furthermore, the increase in the relative cost of cache misses [1] makes optimizations that reduce them, such as ours, attractive. In addition, the likelihood that the instructions and data remain in the cache between consecutive dispatching decreases as the number of threads in the system grows; this is a problem which can be alleviated by improving data affinity. Our scheme, as opposed to CPU affinity, maintains cache hotness within an époque by batching together threads that share data. However, data affinity may introduce additional thread preemption or migration that should be carefully traded-off with the extra cache misses contributed by CPU affinity.

We propose an algorithm that endeavors reduction of data cache misses by applying a paradigm whose essence is 'run the task on the processor holding the currently required data', as opposed to CPU affinity. This paradigm ignores the processor on which the task previously ran and focuses on what data it is about to work on and its location. Furthermore, our approach batches together tasks that use the same data and runs them in succession to maximize cache hotness utilization. The scheduler maintains information on data fragments (DF) shared by multiple tasks. A newly implemented syscall provides DF hints at strategic locations. Hints are generated by the compiler, the user, or potentially by the scheduler. A DF can be a set of variables, parts of arrays, etc. Based on available hints, the scheduler dynamically batches together ready-to-run tasks according to their current DF. Each batch is assigned a CPU and its tasks run in succession to minimize data cache misses. When a task accesses a different DF it is migrated to the appropriate batch. The load balancer attempts to preserve batches during migration.

We formally defined our optimization problem and implemented our scheduling algorithm in the Linux kernel version 2.6. Experiments were conducted on a few benchmarks. The results are very encouraging; cache misses were reduced by up to an order of magnitude on several tests, throughput in benchmarks such as the `Chat` benchmark [16] doubled in some cases and the total application runtime and cache misses were reduced on most tests.

2 Model

The scheduler of an OS handles the lists of running, waiting, and blocked threads. Its responsibilities include scheduling threads onto CPUs, determining their execution order and load balancing the system. In this section we present a system model and use it to describe how threads are dynamically mapped for execution, thereby allowing us to identify sources of overhead incurred by threads contending on the data cache.

2.1 Hardware Model

Our model of an SMP machine consists of processors and caches.

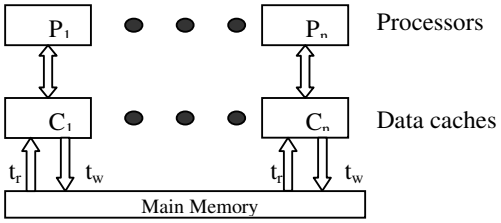

Fig. 1. SMP Architecture

For simplicity, the following assumptions are made on the model:

- Each processor has a single cache and all caches are of equal size.
- The hardware's "snoopy protocol" is 'write-invalidate'.
- The cache can contain all the data for the present run of a thread.

The SMP hardware utilizes some variation of a "snoopy protocol" in order to keep the data in caches coherent. Common policies are: *Write-invalidate* protocols that allow multiple readers, but only one writer at a time. Every write to a shared cache line (block) must be preceded by the invalidation of all other copies of the same line. Local writes to exclusive lines are cheap. *Write-update/(broadcast)* schemes follow a quite an opposite approach. The word to be written to a shared line is distributed to all others, and caches containing that block can update it, thus preventing the stale state.

2.2 System Model

Threads use the machine resources on a need basis. The threads' running order and run time on each époque depend on their resource usage. Once a thread is selected for execution on some CPU, it can use all its resources (e.g. memory, caches and bus). The scheduling process controls when and on which CPU the thread will execute.

The following notations are used:

- Each cache is divided into *k* lines, which can be individually filled, whereas the write back to memory is done for the whole cache by the flush operation.

- Time to fill a cache line is t_r and t_w to write it. T_r and T_w are for *whole* cache.
- The execution time of a single thread v is $ex(v)$.
- Using a cache takes one of the following four forms:

Read	Write	Description
$t_r = 0$	$t_w = 0$	No writes were done by the previous thread thus cache is not dirty, and the needed data is already in cache.
$l \cdot t_r$	$t_w = 0$	l needed data lines are not present in memory; reading the l lines takes $l \cdot t_r$ time units; no writes are performed.
$t_r = 0$	$j \cdot t_w$	j data cache lines are dirty and flushing them back to memory takes $j \cdot t_w$ time units. Needed data is in cache.
$l \cdot t_r$	$j \cdot t_w$	The thread needs to fill the cache with l lines of data ($l \cdot t_r$), and flush changed data requiring write ($j \cdot t_w$).

The following example demonstrates how the cache influences scheduling results. To simplify the example, the following assumptions are used:

- The whole data cache is flushed if the new thread uses different data fragment
- During execution the thread utilizes all the cache attached to the CPU.
- Each thread uses a *single* data fragment.

Example: Assume a cache with a single cache line. Let t_0 be the time in which thread v is mapped to processor P with cache C_P. The execution of v can start at $t_1 = t_0 + T_r$ when the required data is read to cache, and end at $t_2 = t_1 + ex(v) + T_w$. In our example:

- Threads set, TH = $\{v_0, v_1, v_2, v_3, v_4\}$, $ex(v_n) = 1$, $0 \le n \le 4$, $T_r = T_w = 4$
- Number of processors (P) = number of caches (C) = 2

The following constraints are given for the whole run of the application:

- v_2, v_3 and v_0 use the same data fragment – DF1.
- v_1 and v_4 use the same data fragment - DF2

Threads' story: The program starts with thread v_0 running. It spawns four additional threads $v_1 - v_4$ and finishes. In a CPU affinity based scheduler, by default $v_1 - v_4$ are scheduled to run on the same CPU as their parent, v_0. Assume v_0 ran on P_1. Since having all four remaining threads run on P_1 causes imbalance, the load balancer will move two threads to P_2. Two possible scheduling scenarios are presented in Figures 2 and 3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
P_1	$T_r(v_0)$				$ex(v_0)$	$T_r(v_1)$				$ex(v_1)$	$T_w(v_1)$				$T_r(v_2)$				$ex(v_2)$	$T_w(v_2)$			
P_2						$T_r(v_3)$				$ex(v_3)$	$T_w(v_3)$				$T_r(v_4)$				$ex(v_4)$	$T_w(v_4)$			

Fig. 2. Pure CPU affinity based scheduling

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P_1	$T_r(v_0)$				$ex(v_0)$	$ex(v_2)$	$ex(v_3)$	$T_w(v_3)$													
P_2						$T_r(v_1)$				$ex(v_1)$	$ex(v_4)$	$T_w(v_4)$									

Fig. 3. Data affinity based scheduling following the constraints on data sharing

As can be seen from the figures above, not having to read the data for threads v_2, v_3 and v_4 decreases the total run time from 22 time units, to *only* 14! Next section presents a scheduling optimization problem that integrates the parameters in our model.

3 Scheduling to Reduce Cache Miss Penalty

Assume a set $V = \{v_1, v_2, ..., v_n\}$ of currently running tasks to schedule. The number of CPUs available to the scheduler is denoted by *m*. Since, in our model, every CPU has its own cache, *m* is also the number of caches. $DF(v_i)$ is the data fragment used by v_i. $f(v_i, v_j)$ is an asymmetric penalty function for the context switch from v_i to v_j. Its characteristics, in our model, are defined below. *l* is the number of cache lines read by v_j, ϕ denotes an empty CPU and the times t_r and T_w are as defined in Section 2.2.

$$f(v_i, v_j) = \begin{cases} 0 & DF(v_i) = DF(v_j),\ v_i \text{ did not invalidate data.} \\ l \cdot t_r & v_i = \phi \text{ or } DF(v_i) \neq DF(v_j),\ v_i \text{ did not write, } v_j \text{ read } l \text{ lines.} \\ T_w & DF(v_i) = DF(v_j),\ \text{flush of } v_i\text{'s data required.} \\ l \cdot t_r + T_w & DF(v_i) \neq DF(v_j),\ v_i \text{ invalidated data, } v_j \text{ read } l \text{ lines.} \end{cases} \quad (1)$$

To improve system throughput we seek to minimize the penalty induced by the context switches between the tasks of *V* on *m* CPUs, based on the data fragment they use. We define X_W to be the set of all permutations over a subset $W \subseteq V$ and let $w \in X_W$. $F(w)$, the penalty of permutation *w* is defined by: $F(w) = \sum_{i=1}^{|w|-1} f(v_i, v_{i+1})$. The minimal penalty F^* over *W* is $F^*(W) = \min_{w \in X_W} F(w)$. The collection of all partitions of V into *m* equal-sized subsets is denoted by $\Pi_{(V)}$, where each partition in this collection is of size $\left\lceil \frac{n}{m} \right\rceil = k$ (switching to padding task costs 0). Given a partition $S_m \in \Pi_{(v)}$, the minimal penalty for scheduling the partition is: $\tilde{F}(S_m) = \max_{X \in S_m} F^*(X)$ and our optimization problem is to find *T* such that $T = \min_{Z \in \Pi_{(V)}} \tilde{F}(Z)$.

Solving equation *T* when the system's state changes, is impractical. Next we consider a special simplified case where during a thread switch all the cache contents are replaced if the new thread uses a different data fragment. The assumptions previously made still hold; mainly that each task uses exactly one DF and that fragment fits precisely into a processor's cache. In the simplified penalty function *g* we pay a price only when switching between tasks that use different data fragments.

$$g(v_i, v_j) = \begin{cases} 0 & DF(v_i) = DF(v_j) \\ T_r + T_w & v_i = \phi \text{ or } DF(v_i) \neq DF(v_j) \end{cases} \quad (2)$$

An optimal algorithm for a single CPU would be: Go over all ready threads and put each thread in the bin (batch) corresponding to the DF that it uses (*d* bins), and then schedule the bins in arbitrary order. Changing currently running bin is done **only** after all tasks in the bin finished their quota. The runtime of the algorithm on a set of *n* tasks is *O(n)*. Adding and removing a task to the ready list can be performed in *O(1)*.

For multiple CPUs an additional step for distributing the bins between CPUs is added after partitioning the threads into bins. This general partitioning problem is NP-complete. However, based on the dynamic programming pseudo-polynomial partitioning by Cieliebak et al. [17] for $m=O(1)$ CPUs our algorithm will run in $O\left(\frac{d \cdot n^m}{m^{m-1}}\right)$.

4 Data Affinity Based Algorithm

Our scheduler implementation is an enhancement of the Linux 2.6.x scheduler. The Linux scheduler allocates a time slice to each thread/task in the system. An époque, i.e. the time it takes for all tasks to get a chance to run, may vary due to the many heuristics utilized. Tasks that spend much of their time submitting and waiting on I/O requests (I/O bound) have their time slice enlarged. Tasks that tend to run until preempted, spending most time executing code (CPU bound) receive time slice penalties.

The scheduler keeps the tasks in run-queues. A run-queue is a list of runnable tasks which may run in arbitrary order. There exists one list per processor and each task can be on exactly one. It contains two priority arrays; an active and an expired array. Each array contains one queue of runnable tasks per priority level. The 2.6.x scheduler can locate the next highest priority task and pull it off the priority list in constant time.

The scheduler is called explicitly by kernel code that is about to yield CPU and also whenever a task is to be preempted. The scheduler performs the following steps: it recalculates the time slice of the tasks that ended theirs and moves them to the expired array, determines the next highest priority task in the active array and switches to it.

The load balancer complements the scheduler. It ensures that the run-queues are balanced by moving tasks from the busiest run-queue to the relatively under utilized one that invoked it. It is invoked when a run-queue is idle and also via timer interrupt. An imbalanced run-queue contains 25% more tasks than the one on whose behalf the load balancer runs. From the tasks allowed to migrate, the load balancer prefers *expired* ones, since these are probably cache cold. It also favors high priority tasks because of their importance. Tasks are moved as long as an imbalance still exists.

4.1 The Implemented Algorithm

The basic idea is to schedule the current set of tasks in a way that will minimize cache misses, thus resulting in an overall reduction of execution time. This is done by supplying additional information to the scheduler. The algorithm's goal is as follows:

- Tasks that use the same data fragments (DFs) at some point in time are mapped to the same CPU. If a task accesses multiple DFs, it may be reassigned to a different CPU each time it changes DFs; this is done before actually accessing the data (i.e. yield and reschedule).

The mapping of a task to a processor (run-queue) occurs in the following cases: when it is first created (parent's processor is the default), whenever the load balancer is called, and when it returns from a wait queue. We would also like a mapping to occur

when the task changes DFs. From the initial mapping onward until (if at all) it is migrated, the task's time slice is calculated according to the existing Linux policies.

The theoretic formulation in the previous section provides optimal scheduling when all information is known a priori. Unfortunately, real-life systems are dynamic in nature; therefore, there is a need to devise heuristics that use dynamically available info.

The information about which DF each task is using at a given time is passed from user space to the scheduler via a syscall. The syscall is called with the application ID, current (if any) and next DF IDs when a task is created and whenever it changes DFs.

Our scheduler works *online*, holding a list of application descriptors, one for each application that provides DF information. Each descriptor contains a map between DFs and <CPU, priority group> pairs. At any point in time all the tasks in a priority group use the same DF and run in succession on the same CPU. The load balancer is used to rectify 'mistakes' made when in the initial state.

The algorithm consists of several procedures that decide what to do in the following situations: a new DF is encountered, a new task enters the system, a task switches DFs, a task returns from a wait queue, a task dies, and when the system is imbalanced. We next describe those procedures as applied to a single application task.

- When a task arrives with a new DF, the load is verified for all CPUs, in terms of number of currently running tasks. Since the tasks' time-slices are similar in length, this constitutes a good measure for selecting the least loaded CPU.
- Upon arrival of a task with known DF, the scheduler locates its entry in the application map, moves it to the DF's CPU (if it is not there already), and places it in the priority group for that DF.
- When a task changes DFs, the scheduler removes it from its old priority group, preempting it if necessary (this may actually be cheaper than the cache misses), then treats it as if a new thread has arrived with a new or known DF.
- Upon return from a wait queue, a task's current DF entry is looked up in the application map. From there on, as before, it is handled based on whether or not its current DF is known.
- When a task dies (and when changing DFs), a counter in the DF's structure is updated. When it reaches 0, the priority group is flagged as 'may be removed'.
- When the load balancer is called, it checks if the run-queues are imbalanced. If tasks need to be moved, the scheduler tries to identify whether the source of the imbalance is in the managed or unmanaged tasks. When managed threads cause the imbalance, an attempt is made to move an entire priority group. If this is impossible, for example because a group's task is running, the load balancer reverts to unmanaged threads. In rare cases, when the problem is sustained, separating a priority group is allowed.

5 Experimental Results

To assess the performance of our scheduler we tested a custom made benchmark and a few known multi-threaded benchmarks that utilize shared data; achieving significant improvement on the well known Chat benchmark, and a more modest gain on the Hack benchmark, which uses processes. The custom benchmark tests scenarios that

are not exploited by the other benchmarks. Syscalls were added manually to the benchmarks. We discuss the results and provide reasons for the large improvements.

All benchmarks were run on an Intel Xeon, dual processor at 3.2GHz, hyper threaded with L1 cache of 8KB, L2 of 256KB, L3 of 2MB and main memory of 1GB. The hyper-threading ability was turned off for most tests thus creating a 2-way machine. All tests were run using the 2.6.4 Linux distribution of the Linux kernel.

5.1 Chat Benchmark

The Chat benchmark (http://lbs.sourceforge.net) simulates chat rooms with multiple users exchanging messages through TCP sockets. It is based on the Volano Java benchmark that was used in prior papers to show limitations of the 2.4 scheduler [9].

A room consists of 20 users each sending 100 byte messages to the server, which broadcasts them to every other user in the room. Four threads are created per user (80 per room) two on the client side and two on the server side. 100 messages sent by a user translate to 20*100*(1+19)=40,000 transmitted messages per room. At the end of a run, the client side reports the total time and the throughput in messages per second. A lower run-time and higher throughput indicate a more efficient kernel scheduler.

The Chat benchmark was tested with all pairs of parameters from these sets: rooms = {10, 20, and 30} and messages per room = {500, 1000, and 1500}. The results are displayed in Table 1 and represent the average over five runs for each pair.

Table 1 demonstrates that the total gain increases as the number of messages grows. The results for some combinations are less than *half the original scheduler time* with *twice the throughput*! Another statistic worth mentioning is that the standard deviation over the five runs of each combination is considerably smaller for the new algorithm.

We further investigated the pairs that exhibited the largest improvement using Oprofile to count L2 cache misses. As can be seen from Table 2, there is a strong correlation between the number of cache misses and the runtime/throughput.

Table 1. Chat benchmark results. Number of messages ranges from 2 million for the 10 rooms with 500 messages combination to 18 million for 30 rooms and 1500 messages per room

Room Number		10			20			**30**		
Message Number		500	1000	1500	500	1000	1500	500	1000	1500
Avg. Time	Vanilla	5.794	15.303	26.86	11.01	30.7	55.01	16.83	38.93	72.0
	New	4.954	10.935	15.85	7.89	16.85	**24.6**	10.81	22.4	35.2
Avg. Through-put	Vanilla	352281	263621	67594	370625	121874	64336	103133	87981	11559
	New	420382	373323	108010	509062	221189	**139266**	158516	152722	23413

Table 2. Chat benchmark Oprofile results averaged over 5 runs; L2 misses divided by 3000

Room \ Message		10, 1500	20, 1500	30, 1500
Client side	Vanilla	49	88	133
	New	16	34	45
Server side	Vanilla	100	125	206
	New	43	58	68

5.2 Custom Benchmark

Our benchmark consists of a small, highly configurable, application whose parameters include: number of threads (using pthread), number of distinct data fragments (DF), size of those DF, and amount of work done by the threads on the common data. Threads are started in a loop and never sleep voluntarily. We used the Oprofile sampling tool to count the L2 and L3 cache misses (L1 ignored because of its size). L3 may be larger than L2 due to unused pre-fetch into L3

Table 3. The effect of the number of threads on runtime and cache misses. All other parameters are unchanged; iteration number = $10x10^6$; DF number = 4; Oprofile numbers divided by 3000

Thread Number		**8**			**32**			128		
DF size		**1000**	**8000**	**20000**	**1000**	**8000**	**20000**	**1000**	**8000**	**20000**
Avg. Runtime	**Vanilla**	30.63	31.71	32.18	123.7	128.7	129.5	511	510	509
	New	27.84	27.56	27.44	113.5	110.5	110	439	442	439
Avg. L2 Miss	**Vanilla**	11236	6844	8509	72467	50232	48096	334290	247784	219611
	New	1610	305	95	9934	2445	899	49347	9630	3491
Avg. L3 Miss	**Vanilla**	11247	11648	11015	78941	63422	48697	367062	272297	241738
	New	1611	334	70	12467	2389	821	48952	9394	3125

Table 4. The effect of the number of iterations on runtime and cache misses. All other parameters are unchanged; DF number = 4; DF size = 8000; Oprofile numbers divided by 3000

Iterations Number		**$1x10^6$**			**$10x10^6$**			$100x10^6$	
Thread Number		**8**	**32**	**128**	**8**	**32**	**128**	**8**	**32**
Avg. Runtime	**Vanilla**	2.94	12.5	51.11	31.71	128.7	510	312	1283
	New	2.75	11.1	44.2	27.56	110.5	442	277	1101
Avg. L2 Miss	**Vanilla**	630	5572	25781	6844	50232	247784	2537530	501624
	New	31	246	965	305	2445	9630	96430	24230
Avg. L3 Miss	**Vanilla**	771	5555	27132	11648	63422	272297	2732690	634042
	New	32	240	955	333	2389	9394	94145	23910

The results in Table 3 emphasize the fact that if a system is not conscious of data affinity, unnecessary CPU cycles are lost in moving data from one CPU cache to another or during cache replacement. Many scheduling cycles are saved in our method. The results in Table 4 are obvious. The longer the threads run repeatedly accessing the same DF access, even a tiny gain gradually increases to noticeable size.

6 Related Work and Future Enhancements

Processor affinity scheduling has been extensively studied. Squillante [14] and Gupta [6] showed its potential through simulations on several affinity-scheduling algorithms and measuring metrics. Vaswani [12] focused on quantifying the effect of processor

reallocation on performance. Devarakonda [5] revealed a number of problems related to exploiting cache affinity in Unix-like systems.

Affinity based on how fast a task can run on a processor in a heterogeneous processor environment has been studied in [4] [7] [15]. Affinity that looks at the resources that are bound to a processor has been studied in [8] [15]. Affinity based on cache contents, closest to our work, was studied by Torrellas et al [3].

Linux has been widely used for scheduler experiments, especially in version 2.4 trying to deal with the queue lock contention bottleneck. For example, [10] proposed the multi-queue scheduler to enhance scalability on large scale SMP machines. Molloy et al. [11] proposed the ELSC scheduler. There was also Priority Level Scheduler (PLS). Yamamura et al [13] tackled the cache miss problem occurring in kernel code during the walk over the task structures held in a CPU's run-queue.

For practical usage, automatic insertion of the syscall by the compiler is necessary. Further study of the tradeoff between task preemption and the saved cache misses, and the tradeoff between DF sizes vs. the number of data fragments are needed.

References

1. N. P. Jouppi, D. W. Wall - Available instruction-level parallelism for superscalar and superpipelined machines – Proceeding of the 3rd ASPLOS conference, Apr 1989, pp. 272-282.
2. Y. Etsion, D. Tsafrir, D. Feitelson – Effects of Clock Resolution on the Scheduling of Interactive and Soft Real Time Processes – ACM SIGMETRICS, pp. 172-183, Jun 2003.
3. J. Torrellas, A. Tucker, A. Gupta - Evaluating the Performance of Cache Affinity Scheduling in Shared-Memory Multiprocessors, JPDC, Vol. 24, pp. 135-151, 1995.
4. E. Horowitz, S. Sahni – Exact and Approximate Algorithms for Scheduling Nonidentical Processors – J. ACM, vol. 23, no. 2, pp. 317-327, 1976.
5. M. Devarakonda, A. Mukherjee - Issues in Implementation of Cache-Affinity Scheduling – USENIX Technical Conference and Exhibition, pp. 345-357, 1992.
6. A. Gupta et al - The impact of operating system scheduling policies and synchronization methods of performance of parallel applications - ACM SIGMETRICS, Vol. 19, May 1991.
7. E. L. Lawler, C. U. Martel - Scheduling periodically occurring tasks on multiple processors - Information Processing Letters, vol. 7, pp. 9-12, Feb. 1981.
8. D. H. Craft - Resource management in a decentralized system - ACM SIGOPS Operating Systems Review, Vol. 17, Issue 5, Oct. 1983.
9. R. Bryant, B. Hartner – Java Technology, Threads and Scheduling in Linux, *Java Technology Update,* Volume IV, Issue 1 Jan 2000.
10. M. Kravetz et al - Enhancing Linux Scheduler Scalability - 5th ALS, Nov 2001.
11. S. Molloy, P. Honeyman – Scalable Linux Scheduling – CITI Technical Report, May 2001.
12. R. Vaswani et al - The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors – 13th ACM SOSP, pp. 26-40, Oct. 1991.
13. S. Yamamura et al -Speeding Up Kernel Scheduler by Reducing Cache Misses - Proceedings of the FREENIX Track 2002 USENIX Annual Technical Conference, pp. 275-285.
14. M. S. Squillante, E. D. Lazowska – Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling – IEEE TPDS archive, Volume 4(2), pp. 131-143, 1993.
15. R.W. Conway, W.L. Maxwell, L.Miller - Theory of Scheduling - Addison-Wesley, 1967.
16. Linux Benchmark Suite Home page - http://lbs.sourceforge.net/
17. G. J. Woeginger, Z. L. Yu - On the equal-subset-sum problem - Information Processing Letters, 42(6), pp. 299-302, 1992.