European Journal of Scientific Research
ISSN 1450-216X Vol.55 No.3 (2011), pp.419-429

http://www.eurojournals.com/ejsr.htm

A Novel Hard-Soft Processor Affinity Scheduling for Multicore Architecture using Multiagents

G. Muneeswari
Research Scholar, Computer Science and Engineering Department
RMK Engineering College, Anna University-Chennai, Tamilnadu, India
E-mail: munravi76@gmail.com

K. L. Shunmuganathan
Professor & Head
Computer Science and Engineering Department
RMK Engineering College, Anna University-Chennai, Tamilnadu, India
E-mail: kls_nathan@yahoo.com

Abstract

Multicore architecture otherwise called as SoC consists of large number of processors packed together on a single chip uses hyper threading technology. This increase in processor core brings new advancements in simultaneous and parallel computing. Apart from enormous performance enhancement, this multicore platform adds lot of challenges to the critical task execution on the processor cores, which is a crucial part from the operating system scheduling point of view. In this paper we combine the AMAS theory of multiagent system with the affinity based processor scheduling and this will be best suited for critical task execution on multicore platforms. This hard-soft processor affinity scheduling algorithm promises in minimizing the average waiting time of the non critical tasks in the centralized queue and avoids the context switching of critical tasks. That is we assign hard affinity for critical tasks and soft affinity for non critical tasks. The algorithm is applicable for the system that consists of both critical and non critical tasks in the ready queue. We actually modified and simulated the linux 2.6.11 kernel process scheduler to incorporate this scheduling concept. Our result shows the maximum cpu utilization for the non critical tasks and high throughput for the critical tasks.

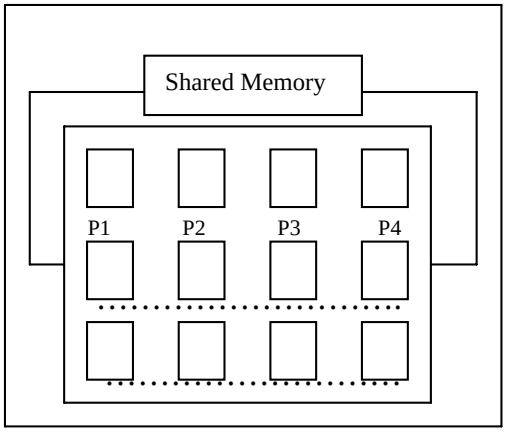
Keywords: Hard Affinity, Soft Affinity, Scheduler, Middle Agent, Processor Agent, Multicore Architecture, Scheduling, Agent Control Block

1. Introduction

Multicore architectures, which include several processors on a single chip [12], are being widely touted as a solution to serial execution problems currently limiting single-core designs. In most proposed multicore platforms, different cores share the common memory. High performance on multicore processors requires that schedulers be reinvented. Traditional schedulers focus on keeping execution units busy by assigning each core a thread to run. Schedulers ought to focus, however, on high utilization of the execution of cores, to reduce the idleness of processors. Multi-core processors do, however, present a new challenge that will need to be met if they are to live up to expectations. Since multiple cores are most efficiently used (and cost effective) when each is executing one process,

organizations will likely want to run one job per core. But many of today's multi-core processors share the front side bus as well as the last level of cache. Because of this, it's entirely possible for one memory-intensive job to saturate the shared memory bus resulting in degraded performance for all the jobs running on that processor. And as the number of cores per processor and the number of threaded applications increase, the performance of more and more applications will be limited by the processor's memory bandwidth. Schedulers in today's operating systems have the primary goal of keeping all cores busy executing some runnable process which need not be a critical processes.

Figure 1: General Multicore System Architecture



One technique that mitigates this limitation is to intelligently schedule jobs of both critical and non critical in nature onto these processors with the help of hard and soft affinities and intelligent approach like multiagents. Multi-Agent Systems (MAS) have attracted much attention as means of developing applications where it is beneficial to define function through many autonomous elements. Mechanisms of selforganisation are useful because agents can be organised into configurations for useful application without imposing external centralized controls. The paper [10] discusses several different mechanisms for generating self-organisation in multi-agent systems [11]. A theory has been proposed (called AMAS for Adaptive Multi-Agent Systems) in which cooperation is the engine thanks to which the system self-organizes for adapting to changes coming from its environment. Cooperation in this context is defined by three meta-rules: (1) perceived signals are understood without ambiguity, (2) received information is useful for the agent's reasoning, and (3) reasoning leads to useful actions toward other agents. Interactions between agents of the system depend only on the local view they have and their ability to cooperate with each other.

The Paper is organized as follows. Section 2 reviews related work on scheduling. In Section 3 we introduce the multiagent system interface with multicore architecture. In section 4 we describe the processor scheduling which consists of hard affinity scheduling and round robin based soft affinity scheduling. In section 5 we discuss the evaluation and results and section 6 presents future enhancements with multicores. Finally, section 7 concludes the paper.

2. Background and Related Work

The research on contention for shared resources [1] significantly impedes the efficient operation of multicore systems has provided new methods for mitigating contention via scheduling algorithms. Addressing shared resource contention in multicore processors via scheduling [2] investigate how and to what extent contention for shared resource can be mitigated via thread scheduling. The research on the design and implementation of a cache-aware multicore real-time scheduler [3] discusses the memory limitations for real time systems. The paper on AMPS [4] presents, an operating system

scheduler that efficiently supports both SMP-and NUMA-style performance-asymmetric architectures. AMPS contains three components: asymmetry-aware load balancing, faster-core-first scheduling, and NUMA-aware migration. In Partitioned Fixed-Priority Preemptive Scheduling [5], the problem of scheduling periodic real-time tasks on multicore processors is considered. Specifically, they focus on the partitioned (static binding) approach, which statically allocates each task to one processing core. In the traditional multi processor system, the critical load balancing task is performed through hardware. In [6] The cooperative load balancing in distributed systems is achieved through processor interaction. dynamic load balancing algorithm [7] deals with many important issues: load estimation, load levels comparison, performance indices, system stability, amount of information exchanged among nodes, job resource requirements estimation, job's selection for transfer, remote nodes selection. In ACO algorithm [8] for load balancing in distributed systems will be presented. This algorithm is fully distributed in which information is dynamically updated at each ant movement. The real-time scheduling on multicore platforms [9] is a well-studied problem in the literature. The scheduling algorithms developed for these problems are classified as partitioned (static binding) and global (dynamic binding) approaches, with each category having its own merits and de-merits. So far we have analyzed some of the multicore scheduling approaches. Now we briefly describe the self-organization of multiagents, which plays a vital role in our multicore scheduling algorithm.

The Cache-Fair Thread Scheduling [14] algorithm reduces the effects of unequal cpu cache sharing that occur on the many core processors and cause unfair cpu sharing, priority inversion, and inadequate cpu accounting. The multiprocessor scheduling to minimize flow time with resource augmentation algorithm [15] just allocates each incoming job to a random machine algorithm which is constant competitive for minimizing flow time with arbitrarily small resource augmentation. In parallel task scheduling [16] mechanism, it was addressed that the opposite issue of whether tasks can be encouraged to be co-scheduled. For example, they tried to co-schedule a set of tasks that share a common working were each 1/2 and perfect parallelism ensured.

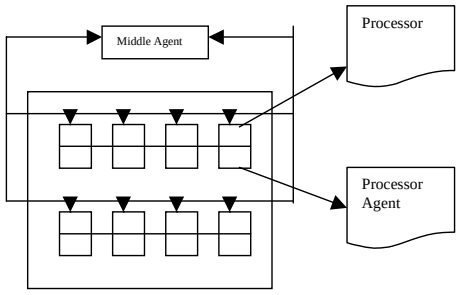
The effectiveness of multicore scheduling [17] is analyzed using performance counters and they proved the impact of scheduling decisions on dynamic task performance. Performance behavior is analyzed utilizing support workloads from SPECWeb 2005 on a multicore hardware platform with an Apache web server. The real-time scheduling on multicore platforms [18] is a well-studied problem in the literature. The scheduling algorithms developed for these problems are classified as partitioned (static binding) and global (dynamic binding) approaches, with each category having its own merits and de-merits. So far we have analyzed some of the multicore scheduling approaches. Now we briefly describe the self-organization of multiagents, which plays a vital role in our multicore scheduling algorithm.

The multiaget based paper [19] discusses several different mechanisms for generating self-organisation in multi-agent systems [20]. For several years the SMAC (for Cooperative MAS) team has studied self-organisation as a means to get rid of the complexity and openness of computing applications [21]. A new approach for multiagent based scheduling [12] for multicore architecture and load balancing using agent based scheduling [13] have improved cpu utilization and reduces average waiting time of the processes.

3. Multicore Architecture with Multiagent System

Every processor in the multicore architecture (Fig.2) has an agent called as Processor Agent (PA). The central Middle Agent (MA) will actually interact with the scheduler. It is common for all Processor Agents.

Figure 2: Multicore architecture with multiagent System



Every PA maintains the following information in PSIB (Processor Status Information Block). It is similar to the PCB (Process Control Block) of the traditional operating system. Processor Status may be considered as busy or idle (If it is assigned with the process then it will be busy otherwise idle) Process name can be P1or P2 etc., if it is busy. 0 if it is idle. Process Status could be ready or running or completed and the burst time is the execution time of the process.As we are combining the concept of multiagent system with multicore architecture, the processor characteristics are mentioned as a function of Performance measure, Environment, Actuators, Sensors (PEAS environment), which is described in table.1 given below. This describes the basic reflexive model of the agent system.

Table 1: Multicore in PEAS environment

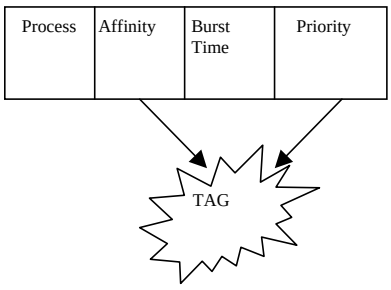
Agent Type	Performance Measure	Environment	Actuators	Sensors
Multicore Scheduling	Minimize the average waiting time of the processes and reduces the task of the scheduler	Multi core architecture and multi processor systems	PA registers with MA, MA assigns process to the appropriate processor via dispatcher	Getting processor state information from PSIB, Getting task from scheduler

We know that multiagent system is concerned with the development and analysis of optimization problems. The main objective of multiagent system is to invent some methodologies that make the developer to build complex systems that can be used to solve sophisticated problems. This is difficult for an individual agent to solve. Os scheduler implements the multiagent concept. Every agent maintains the linked list of processes.

4. Processor Scheduling

Before starting the process execution, the operating system scheduler selects the processes from the ready queue based on the first come first served order. Each process in the centralized queue has a tag indicating its priority (critical or non critical task) and preferred processor shown in fig.3. Critical tasks are assigned with priority 1 and non critical tasks are assigned with priority 0.At allocation time, each task is allocated to its processor in preference to others.

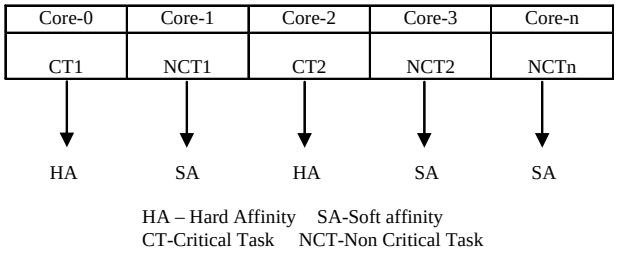
Figure 3: Ready queue process with the tag field



The scheduler after selecting M number of processes from the ready queue places in the middle agent. The middle agent is implemented as a queue data structure shown in fig.4. Middle agent holds

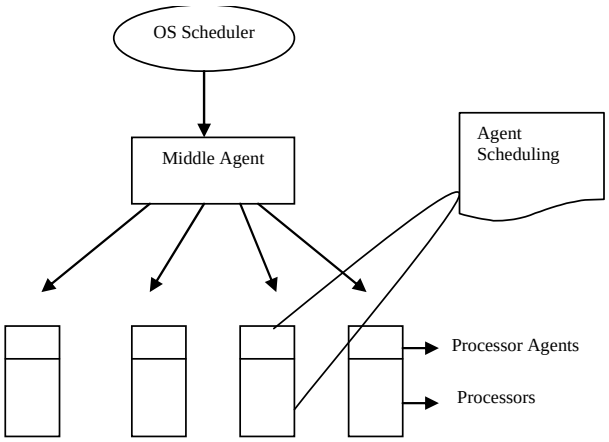
M tasks which is greater than N (Number of processor cores). Precisely the middle agent is acting as a storage space for faster scheduling. In fig.4, for example since CT1 is assigned with hard affinity, it should not be preempted after the time quantum expires. Most of the critical tasks are real time tasks and it is not desirable to context switch.

Figure 4: Middle agent queue implementation



Critical tasks should not be context switched. Processor affinity is maintained only for critical tasks. Actually we employ the basic concept of round robin scheduling along with that soft affinity based scheduling has been used. During the context switching time if it is not a critical task then it can be allocated to the idle processor to improve the overall efficiency (no resource contention). But if it is a critical task it should not be context switched and it has to be executed for its full burst time. If it is a critical task then agent will assign the process to the same processor. Otherwise it will assign the process to the idle processor.

Figure 5: Process Scheduling by OS scheduler, middle agent and individual agents

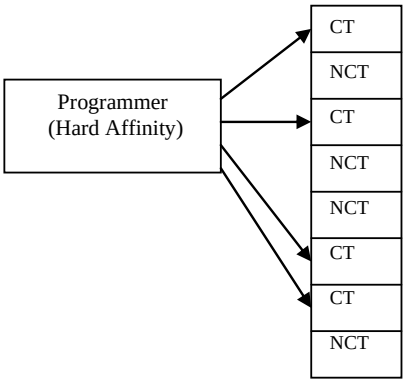


A brief explanation of overall scheduling is shown in fig.5. Initially all the jobs from the ready queue are selected by the os scheduler on fcfs basis and then all the selected processes are placed in the middle agent. The individual agent of every processor selects the job from the middle agent queue and assign it to the processor. This agent actually eliminates the job of the dispatcher.

4.1. Hard Affinity Based Scheduling for Critical Tasks

Scheduling processes to specific processors is called setting a processor affinity mask This affinity mask contains bits for each processor on the system, defining which processors a particular process can use. When the programmer set affinity for a process to a particular processor, all the processes inherit the affinity to the same processor. In fig.6, Programmer is setting hard affinity for real time critical tasks meaning that it should not be preempted from the processor to which it is assigned with the help of hard affinity. In the diagram, CT refers to the critical task and NCT refers to the non critical tasks.

Figure 6: Hard affinity assigned by the programmer / user



Threads restricted by a hard affinity mask will not run on processors that are not included in the affinity mask. Hard affinity used with Scheduling can improve performance of an multicore processor system substantially. However, hard affinity might cause the processors to have uneven loads. If processes that have had their affinity set to a specific processor are causing high CPU utilization on that processor while other processors on the system have excess processing capacity, the processes for which a hard affinity has been set might run slower because they cannot use the other processors.

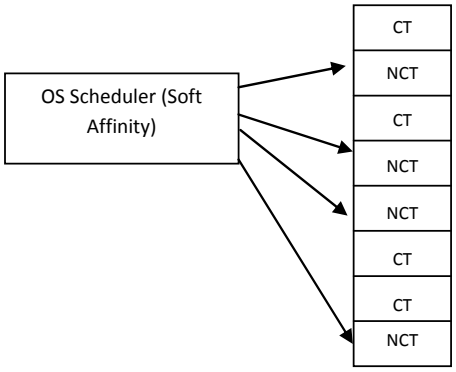
In our proposed algorithm the programmer can prescribe their own affinities and that will be termed to be hard affinities. The tag field of the critical tasks consists of high priority and affinity to the processor. Every processor has a dedicated processor agent and that is responsible for maintaining agent control block (ACB). This agent control block will be useful to identify the free idle processor for next scheduling.

In the case of critical tasks since it is not preempted it is not mandatory to establish an agent control block.

4.2. Round Robin based Soft Affinity Scheduling for Non Critical Tasks

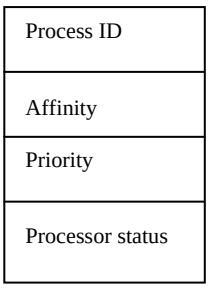
In the case of *soft* processor affinity, the scheduler automatically assigns which processor should service a process (fig.7). The soft affinity for a process is the last processor on which the process was run if it is free or the ideal processor of the process. The soft affinity processor scheduling algorithm enhances performance by improving the locality of reference. However, if the ideal or previous processor is busy, soft affinity allows the thread to run on other processors, allowing all processors to be used to capacity. Actually the default round robin scheduler will be used for the remaining set of non critical tasks.

Figure 7: Soft affinity assigned by the scheduler



We actually create a linked list of agent control block (ACB fig.8) for all the non critical tasks. It plays a vital role during context switching. The important components of ACB are process ID, affinity, priority, processor status. Processor status can be 0 if it is free otherwise it is set to 1.

Figure 8: Agent Control Block (ACB)



After the quantum expires for the non critical tasks, the processor agent checks the individual agent control block to identify whether it is a critical task or not. If it is a critical task then it will not be preempted. Otherwise it can be preempted and joins at the end of the middle agent queue. After some time if the context switched job is ready for execution then it can be allocated to the same processor if it is free. Otherwise the process can be allocated to the idle processor. The middle agent identifies the idle processor by scanning the agent control block of every agent starting from agent1.

5. Evaluation and Results

In this section, we present a performance analysis of our scheduling algorithm using a gcc compiler and linux kernal version 2.6.11. Multiagent simulation is executed with the help of Flame tool accompanied with MinGW C compiler, Xparser, Libmboard. The Kernel scheduler API for getting and setting the affinities are shown below:

int sched_getaffinity(pid_t pid, unsigned int len, unsigned long * mask);

This system call retrieves the current affinity mask of process ‘pid’ and stores it into space pointed to by ‘mask’.‘len’ is the system word size: sizeof(unsigned int long)

int sched_setaffinity(pid_t pid, unsigned int len, unsigned long * mask);

This system call sets the current affinity mask of process ‘pid’ to *mask ,‘len’ is the system word size: sizeof(unsigned int long)

The results show that there is a linear decrease in the average waiting time as we increase the number of cores. Our scheduling algorithm results in keeping the processor busy and reduces the average waiting time of the processes in the centralized queue. As an initial phase, our algorithm partitions every process into small sub tasks. Suppose a process, $P_{i,j}$ is being decomposed into k smaller sub tasks $P_{i,j,1}$ $P_{i,j,2}$ ..... $P_{i,j,k}$, where τ_{ijl} is the service time for P_{ijl} Each P_{ijl} is intended to be executed as uninterrupted processing by the original thread $P_{i,j}$, even though a preemptive scheduler will divide each τ_{ijl} into time quanta when it schedules P_{ijl} . Now the total service time for $P_{i,j}$ process can be written as

$\tau(P_{i,j})= \tau_{I,j,1+} \tau_{I,j,2 +} \ldots\ldots\ldots \tau_{i,j,k}$

In every core we calculate the waiting time of the process as previous process execution time. The execution time of the previous process is calculated as follows:

$P_{ET} = P_{BT} + \alpha_i + \beta_i + \delta_i + \gamma_i$

Where P_{ET} is the execution time of the process, P_{BT} is the burst time of the process, α_i is the scheduler selection time, β_i is the Processor Agent request time, δ_i is the Middle Agent response time, γi is the dispatcher updation time. The average waiting time of the process is calculated as the sum of all the process waiting time divided by the number of processes.

$P_{AWT}= \Sigma(i=1..n) P(i=1..n) / N$

Here when we say the process P it indicates the set of subtasks of the given process. For our simulation we have taken 1000 processes as a sample that consists of large number of critical tasks and few non critical tasks and this sample is tested against 25, 50, 75, 100, 125, 150, 175, 200, 225, 250 cores. Matlab tools are used for generating the number of tasks. By Performance analysis, we can see that the utilization of cpu increases tremendously for different set of processes keeping the number of

cores constant. The same simulation was executed for different number of cores also. We discovered that the average waiting time decreases slowly with the increase of the number of cores. The utilization of the cpu is maximum for our algorithm when compared to the traditional real time scheduling algorithms. Our experiment results are varied for the different loads and different cores.

- For each set of parameters, the experiment is repeated 100 times and the results shown are the averages from the 100 experiments.
- In fig.9, we explained the number of cores vs average waiting time for 1000 processes. In fig.10,we show the performance analysis of our algorithm against traditional round robin scheduling algorithm. In fig.11, we show the performance analysis of our algorithm against traditional shortest job first scheduling algorithm. In fig.12, we show the performance analysis of our algorithm against traditional EDF scheduling algorithm. Only for EDF algorithm our math lab tool generates only critical tasks. In fig.13, we show the summary of cpu utilization for all the algorithms. From the results we prove that the average waiting time of the processes decreases along with the tremendous increase in cpu utilization for our affinity based algorithm.

Figure 9: Number of cores vs average waiting time for 1000 processes

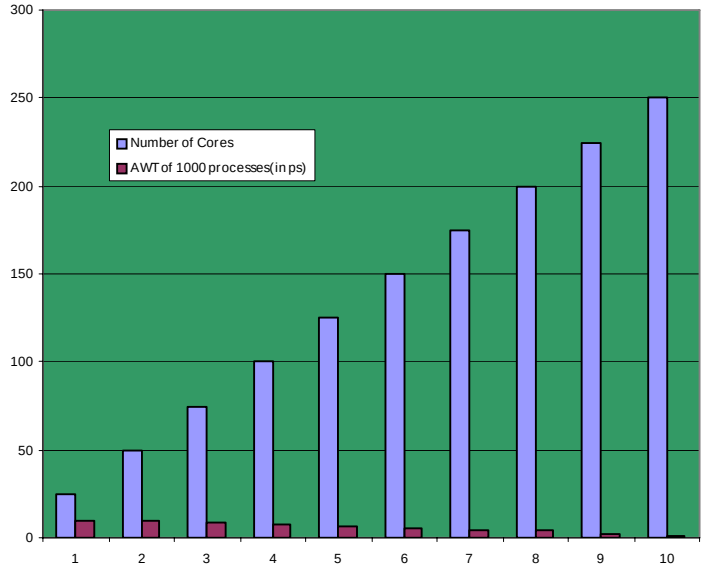


Figure 10: Performance analysis of Affinity and RR algorithms

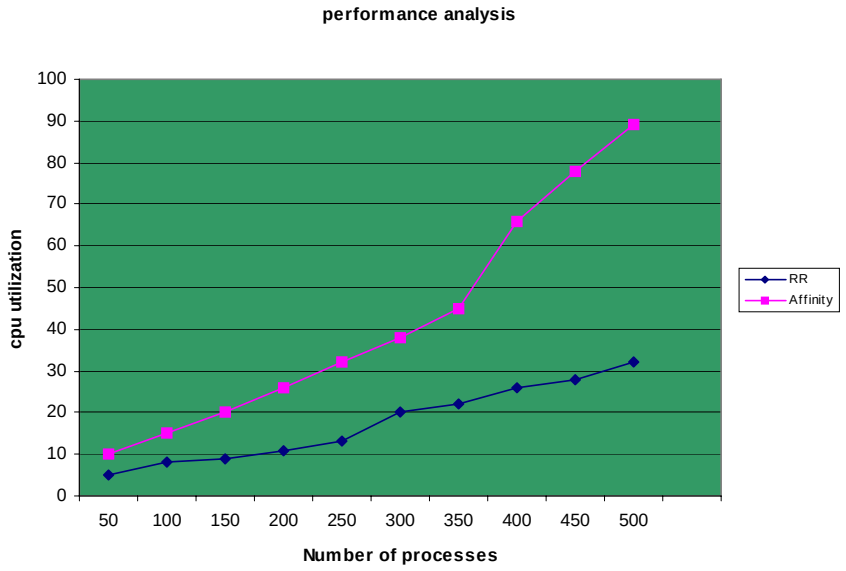


Figure 11: Performance analysis of Affinity and SJF algorithms

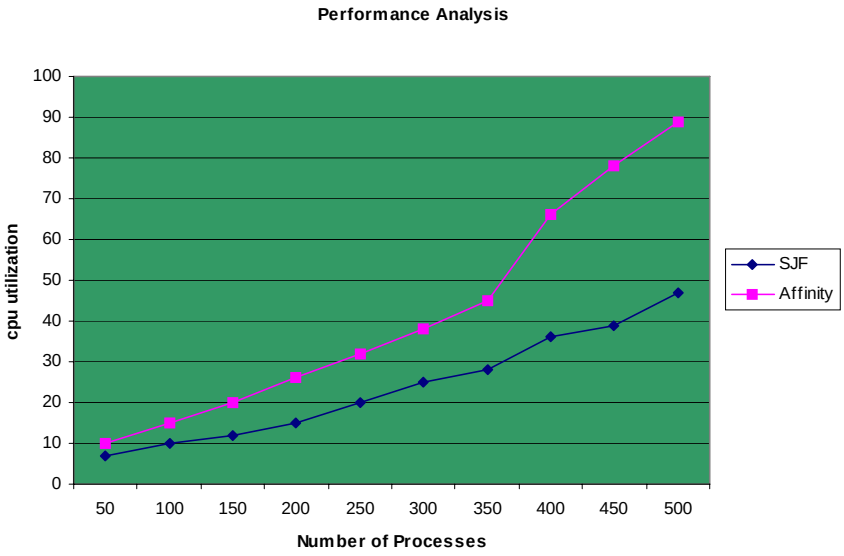


Figure 12: Performance analysis of Affinity and EDF algorithms

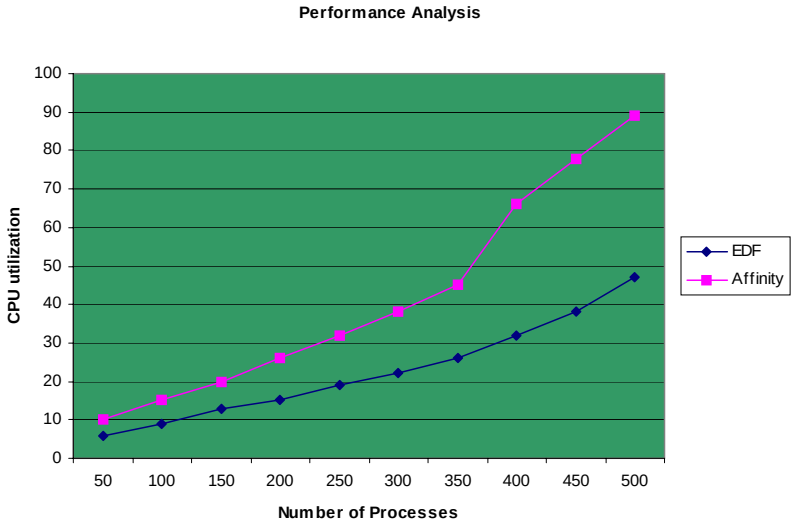
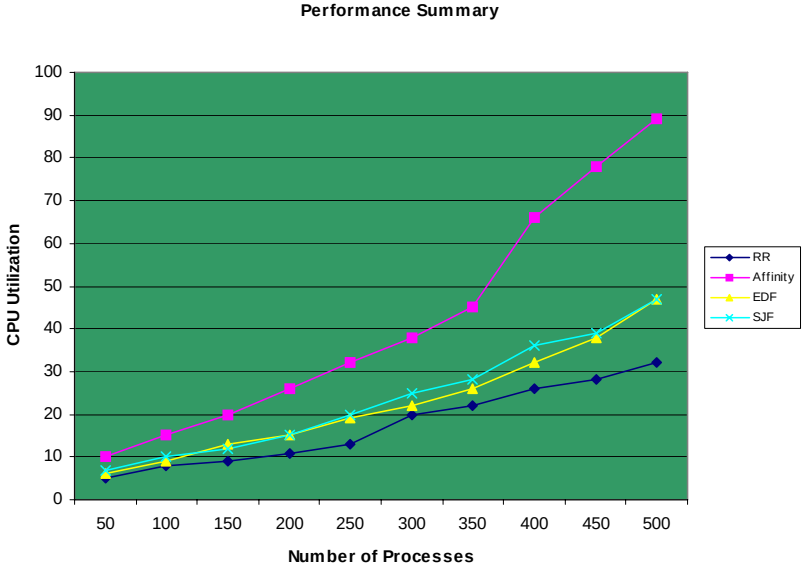


Figure 13: Summary of Performance analysis of all the algorithms



6. Future Enhancements

Although the results from the linux kernal version 2.6.11 analysis in the previous section are encouraging, there are many open questions. Even though the improvement (average waiting time reduction) possible with number of cores, for some workloads there is a limitation by the following properties of the hardware: the high off-chip memory bandwidth, the high cost to migrate a process, the small aggregate size of on-chip memory, and the limited ability of the software (agents) to control hardware caches. We expect future multicores to adjust some of these properties in favor of our multiagents based scheduling. Future multicores will likely have a larger ratio of compute cycles to off-chip memory bandwidth and can produce better results with our algorithm. Our scheduling method can be extended for the actual real-time systems implemented on multicore platforms that encourages individual threads of multithreaded real-time tasks to be scheduled together. When such threads are cooperative and share a common working set, this method enables more effective use of on-chip shared caches and other resources. AN efficient load sharing can be incorporated with the help of multiagents as they cooperate with each other and get the status information of every other processor in the multicore chip. This could also be extended for the distributed system that may deploy the multicore environment.

7. Conclusion

This paper has argued that multicore processors pose unique scheduling problems that require a multiagent based software approach that utilizes the large number processors very effectively. We actually eliminated the work of dispatcher with the help of processor agents itself. Each processor scheduling will be similar to the self scheduling employed in the traditional multiprocessor system. This is possible only with the help of processor agents assigned for every processor. We also proved that lot of drastic enhancements in the traditional scheduler that optimizes for CPU cycle utilization. We discovered that the average waiting time decreases slowly with the increase of the number of cores. As a conclusion our new novel approach eliminates the complexity of the hardware and improved the CPU utilization to the maximum level. The cpu utiliazation is actually maximum for the critical tasks and ideal processors are utilized well in the case of non critical tasks. Even though there is a cost of migrating the non critical tasks to some other processor efficient and maximum utilization of the cpu is our primary concern.

References

[1] Sergey Zhuravley, Blagoduroy, Alexandra Fedorova,2010. “Managing contention for shared resources on multicore processors”, Communications of the ACM Volume 53, Pages: 49-57 Issue 2 February.

[2] Sergey Zhuravley, Blagoduroy, Alexandra Fedorova, 2010. “Addressing shared resource contention in multicore processors via scheduling”, Architectural support for Programming Languages and Operating Systems, Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems Pages: 129-142.

[3] John M.Calandrino, James H. Anderson, 2009. “On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler”, 21st Euromicro Conference on Real-Time Systems July 01-July 03.

[4] Tong LiDan, BaumbergerDAvid A, KoufatyScott Hahn, 2007.“Efficient operating system scheduling for performance asymmetric multi-core architectures”, Conference on High Performance Networking and Computing Proceedings of the ACM/IEEE conference on Supercomputing.

[5] Karthik Lakshmanan, Ragunathan (Raj) Rajkumar, and John P. Lehoczky,2009. “Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors”, Proceedings of the 21st Euromicro Conference on Real-Time Systems Pages: 239-248.

[6] D. Grosu, A. T. Chronopoulos, M. Y. Leung, 2008. “Cooperative Load Balancing in Distributed Systems”, Concurrency and Computation,Practice and Experience. Vol. 20, No. 16, pp. 1953-1976, November

[7] Ali M. Alakeel, 2010. “Load Balancing in Distributed Computer Systems”, International Journal of Computer Science and Information Security Vol. 8 No. 4 July.

[8] Dahoud Ali, Mohamed A. Belal and Moh’d Belal Zoubi, 2010, “Load Balancing of Distributed Systems Based on Multiple Ant Colonies Optimization” , American Journal of Applied Sciences 7 (3): 433-438.

[9] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi, 2006. “Real-Time Scheduling on Multicore Platforms”, Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, Pages: 179 – 190.

[10] Carole Bernon, 2006, “Applications of Self-Organising Multi-Agent Systems: An Initial Framework for Comparison”, IRIT, INRIA.

[11] Di Marzo Serugendo G., Gleizes M-P. and Karageorgos, 2006. “Self-Organisation and Emergence in MAS: An Overview”, A INFORMATICA.

[12] G.Muneeswari, A.Sobitha Ahila, Dr.K.L.Shunmuganathan, 2011. “A Novel Approach to Multiagent Based Scheduling for Multicore Architecture”, GSTF journal on computing, Singapore vol1.No.2.

[13] G.Muneeswari, Dr.K.L.Shunmuganathan, 2011. “Improving CPU Performance and Equalizing Power Consumption for Multicore Processors in Agent Based Process Scheduling”, International conference on power electronics and instrumentation engineering, Springer-LNCS.

[14] Alexandra Fedorova, Margo Seltzer and Michael D. Smith, 2006. “Cache-Fair Thread Scheduling for Multicore Processors”, TR-17-06.

[15] Chandra Chekuri, 2004. “Multiprocessor Scheduling to Minimize Flow Time with Resource Augmentation”, STOC’04, June 13–15.

[16] James H. Anderson and John M. Calandrino, 2006. “Parallel Task Scheduling on Multicore Platforms”, ACM SIGBED.

[17] Stephen Ziemba, Gautam Upadhyaya, and Vijay S. Pai.,2004. “Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters”.

[18] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi, 2006. “Real-Time Scheduling on Multicore Platforms”, Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, Pages: 179 – 190.

[19] Carole Bernon, 2006. “Applications of Self-Organising Multi-Agent Systems: An Initial Framework for Comparison”, IRIT, INRIA.

[20] Di Marzo Serugendo G., Gleizes M-P. and Karageorgos A, 2006, “Self-Organisation and Emergence in MAS: An Overview”, INFORMATICA 30 2006 40-54.

[21] Gleizes M.P, Camp, V. and Glize P,1999. “A Theory of Emergent Computation Based on Cooperative Self-Organisation for Adaptive Artificial Systems”, 4th European Congress of Systems Science, Valencia. 623-630.