A Novel Approach to Multiagent based Scheduling for Multicore Architecture

G.Muneeswari
Research Scholar
R.M.K Engineering College
Anna University, Chennai
munravi76@gmail.com

A.Sobitha Ahila
Research Scholar
R.M.K Engineering College
Anna University, Chennai
ssa.cse@rmkec.ac.in

Dr.K.L.Shunmuganathan
Professor & Head, Department of CSE
R.M.K Engineering College
TamilNadu, India
kls_nathan@yahoo.com

Abstract**:** *In a Multicore architecture, each package consists of large number of processors. This increase in processor core brings new evolution in parallel computing. Besides enormous performance enhancement, this multicore package injects lot of challenges and opportunities on the operating system scheduling point of view. 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. In this paper we combine the AMAS theory of multiagent system with the scheduler of operating system to develop a new process scheduling algorithm for multicore architecture. This multiagent based scheduling algorithm promises in minimizing the average waiting time of the processes in the centralized queue and also reduces the task of the scheduler. We actually modified and simulated the linux 2.6.11 kernel process scheduler to incorporate the multiagent system concept. The comparison is made for different number of cores with multiple combinations of process and the results are shown for average waiting time Vs number of cores in the centralized queue.*

Keywords**: *multicore, multiagent, centralized queue, average waiting time, scheduling, processor agent, middle agent, dispatcher.***

1. Introduction

Multicore architectures, which include several processors on a single chip, 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 idle 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. One technique that mitigates this limitation is to intelligently schedule jobs onto these processors with the help of software approach like multiagents.

The Paper is organized as follows. Section 2 reviews related work. In Section 3 we introduce the multiagent system interface with multicore architecture. This describes Middle Agent system implementation, process scheduler organization and process dispatcher organization. In section 4 we discuss the evaluation and results and section 5 presents future enhancements with multicores. Finally, section 6 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

DOI: 10.5176_2010-2283_1.2.58

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.

The Cache-Fair Thread Scheduling [6] 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 [7] 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 [8] 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 [9] 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 [10] 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.

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 [11] discusses several different mechanisms for generating self-organisation in multi-agent systems [12]. 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 [13]. 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.

3. Multicore Architecture with Multiagent System

Every processor in the multicore architecture (Fig.1) 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.

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.

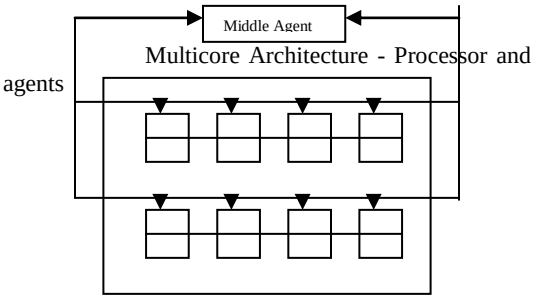

Figure 1. Multicore architecture with multiagent system

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	Multicore architecture and multiprocessor systems	PA registers with MA, MA assigns process to the appropriate processor via dispatcher	Getting processor state information from PSIB, Getting task from scheduler

3.1 The Process Scheduler Organization

Shared memory multicore system consists of a ready queue where all the processes that are ready for execution will be available. cpu scheduling is remarkably similar to other types of scheduling that have been studied for years. In this paper we take a model of the timesharing system, the criteria focused on providing an equitable share of the processor per unit time to each user or process is to minimize the average waiting time. The criteria for selecting the scheduling strategy will depend on the goals of the OS. These goals may emphasize priorities of the processes, fairness, overall resource utilization, maximized throughput and average waiting time. Scheduling algorithms for modern operating systems ultimately use internal properties.

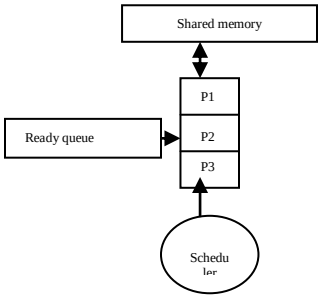

Figure 2. Shared memory with a ready queue

In our approach (Fig.2), the scheduler selects any one of the processes from the ready queue according to the priority based RR scheduling algorithm.

3.2 The Process Dispatcher Organization

After getting the Processor Agent name, process name and burst time from the MA the dispatcher just forwards the information to the Processor Agent. The PA that receives the information from the dispatcher will update its PSIB. The process will be allocated to the cpu by the dispatcher by performing context switch from itself to the selected process.

In the version Linux scheduler, the dispatcher is a kernal function, schedule(). This function gets called from other system functions, as well as after every system call and normal interrupt. Each time the dispatcher is called, it performs periodic work, inspects the set of tasks in the TASK_RUNNING state, chooses one to execute according to the scheduling policy, and then dispatches the task to run on the CPU until an interrupt occurs. The policy is a variant of RR scheduling. It uses the conventional time slicing mechanisms to place the upper bound on the amount of time a task can use the cpu continuously if other tasks are waiting to use it. A dynamic priority is computed on the basis of the value assigned to the task by the nice() or setpriority() system calls, and by the amount of time that a process has been waiting for the cpu to become available. The counter field in the task descriptor becomes the key component in determining the dynamic priority of the task.

3.3 Middle Agent System Implementation

The central common Middle Agent (Fig.3) maintains two tables. Initially all the PA must send a request to the MA to register with it (one time only the registration is made). Middle Agent is the central heart of the scheduling process. It communicates with the scheduler for getting the process to be scheduled on large number of processor. It also interacts with the dispatcher whose function is to assign the process to the different cores.

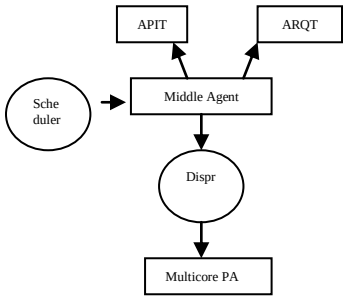

Figure 3. Middle Agent system communication

3.3.1 Agent Processor Information Table (APIT)

When the PA sends a register request to middle agent (MA), the relevant information is stored in the Agent Processor Information Table (Refer Table.2). Initially all the entries for the processor state will be idle. Once the scheduler selects the first set of processes based on RR scheduling algorithm, it then contacts with MA to give the process name and the burst time and the state of the process will be changed to busy by the MA in the following table. (So initially the assignment will be FIFO order (i.e agent registration will be in FIFO only). After updating the

table MA sends the corresponding PA name and process name to the Dispatcher. The dispatcher is finally responsible for allocating the processes to the respective processor via processor agents and the corresponding table is also updated. The Agent processor information table can be maintained as part of the operating system process management part of the scheduler. Linked list representation is a preferred data structure used for the arrangement of processes on the stipulated table.

Table 2. APIT- Agent Processor Information Table

Agent Name	Processor Id	Processor State	Process name	Burst Time
PA1	PR1	Busy	P1	10ps
. . .	. . .	. . .	. . .	. . .
PAn	PRn	Busy	Pn	200ps

3.3.2 Agent Request Table (ARQT)

Whenever the processor completes the first set of tasks, the agents of the processor PA immediately send a process request message to the MA. The MA after receiving the request message from the PA stores the information in ARQT-Agent Request Table, which can be implemented as a queue (Refer Table.3). Before storing the information MA has to check APIT to see whether the requested agent has already registered with the MA. Initially the process name will be 0 because we received only the request message from the PA. The activities will be repeated again. When the scheduler is ready it sends the job to the MA and the MA stores the process name and burst time in the following table. It then sends the corresponding PA and process name along with burst time to the dispatcher.

Table 3. ARQT- Agent Request Table

4. 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. 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 and tested against 25, 50, 75, 100, 125, 150, 175, 200, 225, 250 cores. In Fig.4, the average waiting time of 1000 processes is obtained for the selected number of cores. We discovered that the average waiting time decreases slowly with the increase of the number of cores.

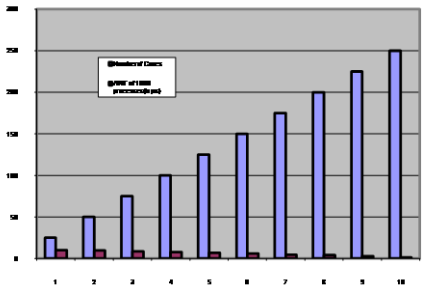

Figure 4. Number of cores vs average waiting time for 1000 processes

5. Future Enhancements

Although the results from the linux kernal version 2.6.11analysis in the previous section are encouraging, there are many open questions. Even though the

Agent Name	Processor Id	Process Name	Burst Time
PA1	PR1	P1	10ps
. . .	. . .	. . .	. . .
PAn	PRn	Pn	200ps

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.

6. 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 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.

References

[1] Managing contention for shared resources on multicore processors, Communications of the ACM Volume 53, Pages: 49-57 Issue 2 February 2010

[2] Addressing shared resource contention in multicore processors via scheduling, Sergey Zhuravley, Blagoduroy, Alexandra Fedorova, 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, 2010

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

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

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

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

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

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

[9] Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters. Stephen Ziemba, Gautam Upadhyaya, and Vijay S. Pai2004

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

[11] Applications of Self-Organising Multi-Agent Systems: An Initial Framework for Comparison Carole Bernon IRIT, INRIA2006

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

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

AUTHORS PROFILE

G.Muneeswari received her B.E and M.E degrees in Computer Science and Engineering from Madras University and Anna University in 1998 and 2004, respectively, and pursuing Ph.D from Anna University, Chennai ,India. Currently, she is an Assistant Professor in the Department of Computer Science and Engineering at R.M.K Engineering College, Chennai, India. Her Research interests include Multicore Architecture, Parallel and distributed computing and Artificial Intelligence.

Dr.K.L.Shanmuganathan B.E, M.E., M.S., Ph.D working as Professor & Head, Department of Computer Science & Engineering, RMK Engineering College, Chennai, TamilNadu, India. He has more than 15 publications in National and International Journals. He has more than 18 years of teaching experience and his areas of specializations are Artificial Intelligence, Networks, Multiagent Systems, DBMS.

A.Sobitha Ahila received her B.E degree in Electronics and Communication Engineering and M.E degree in Computer Science and Engineering from Madurai Kamaraj University, Bharathidasan

university in 1997 and 2001, respectively, and pursuing Ph.D from Anna University, Chennai ,India. Currently, she is an Assistant Professor in the Department of Computer Science and Engineering at R.M.K Engineering College, Chennai, India. Her Research interests include Network Security, Multicore Architecture and Artificial Intelligence.