Implementation of Cache Fair Thread Scheduling for multi core processors using wait free data structures in cloud computing applications.

A.S.Radhamani, Research Scholar /
Department of Computer Science and Engineering, Manonmanium Sundaranar University,Tirunelveli.
asradhamani@gmail.com

E.Baburaj,Professor, Department of Computer Science and Engineering, Sun Collegeof Engineering and Technology, Nagercoil.alanchybabu@gmail.com

Abstract- As multi-core processors with tens or hundreds of cores begin to grow, system optimization issues once faced only by the High-Performance Computing (HPC). To satisfy the requirement, one can leverage multi-core architectures to parallelize traffic monitoring so as to progress information processing capabilities over traditional uni-processor architectures. In this paper an effective scheduling framework for multi-core processors that strike a balance between control over the system and an effective network traffic control mechanism for high-performance computing is proposed. In the proposed Cache Fair Thread Scheduling (CFTS), information supplied by the user to guide threads scheduling and also, where necessary, gives the programmer fine control over thread placement. Cloud computing has recently received considerable attention, as a promising approach for delivering network traffic services by improving the utilization of data centre resources. The primary goal of scheduling framework is to improve application throughput and overall system utilization in cloud applications. The resultant aim of the framework is to improve fairness so that each thread continues to make good forward progress. The experimental results show that the parallel CFTS could not only increase the processing rate, but also keep a well performance on stability which is important for cloud computing. This makes, it an effective network traffic control mechanism for cloud computing.

Key Words: *Cache Fair Thread Scheduling, multi-core, cloud computing*

I. INTRODUCTION

Explicit parallel architectures require specification of parallel task along with their interactions. For network traffic analysis it is a challenge for several reasons. First, packet capture applications are memory bound, but memory bandwidth does not seem to increase as fast as the number of core available [1]. Second, balancing the traffic among different processing units is challenging, as it is not possible to predict the nature of the incoming traffic. Exploiting the parallelism with general-purpose operating systems is even more difficult as they have not been designed for accelerating packet capture. During the last three decades, memory access has always been one of the worst cases of scalability and thus several solutions to this problem have been proposed in [2]. With the advent of Symmetric Multiprocessor Systems (SMP), multiple processors are connected to the same memory bus, hereby causing processors to compete for the same memory bandwidth. Integrating the memory controller inside the main processor is another approach for increasing the memory bandwidth. The main advantage of this architecture is fairly obvious: multiple memory modules can be attached to each processor, thus increasing bandwidth. In shared memory multiprocessors, such as SMP and NUMA (Non Uniform Memory Access), a cache coherence protocol [2] must be used in order to guarantee synchronization among processors. A multi-core processor is a processing system composed of two or more individual processors, called cores, integrated onto a single chip package. As a result, the inter-core bandwidth of multi-core processors can be many times greater than the one of SMP systems.

For traffic monitoring and control applications, the most efficient approach to optimize the bandwidth utilization is to reduce the number of packet copies. The proposed work shows that improving the cache locality of packet capture software through packet reordering allows the overall performance to be significantly improved by CFTS. In multi-core and multi-processor architectures, memory bandwidth can be wasted in many ways, including improper scheduling, wrong balancing of Interrupt ReQuests (IRQ) and subtle mechanisms such as false sharing [3]**.** For these reasons, squeezing performance out of those architectures require additional effort. Even though there is a lot of ongoing research in this area, most of the existing schedulers are unaware of architectural differences between cores, therefore the scheduling does not guarantee the best memory bandwidth utilization. This may happen when two threads using the same data set are scheduled on different processors, or on the same multi-core processors having separate cache domains. Therefore an effective software scheduler to better distribute the workload among threads can substantially increase the scalability is achieved by CFTS. Cloud Computing refers to both the

978-1-4673-0126-8/11/$26.00 ©2011 IEEE

applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services and the datacenter hardware and software is called a Cloud [5]. In the cloud, all the resources including infrastructure, platform and software are delivered as services, which are available to customers in a pay-peruse model. In this way, cloud computing gains advantages such as cost savings, high availability, and easy scalability [17]. Cloud must guarantee all resources as a service and provide them to users by means of customized System Level Architectures (SLAs). However, in the cloud, there might be tens of thousands or even more users accessing resource simultaneously, which give an extremely high pressure on the cloud. When all users are requiring service from the cloud, the out traffic from data center will be tremendous and the network bandwidth must be managed effectively to serve all users [16]. So, it is necessary to control the data streams and make a better utilization of free network resources in the cloud.

In this paper, a network traffic monitoring and control for multi core processors based on cloud is proposed. Also it monitors the application behavior at run-time, analyze the collected information, and optimize multi-core architectures for cloud computing resources. The traditional operations on traffic packet structures are modified to reduce the strong dependency in the sequential code. Then wait-free data structures are applied selectively to make multi-core parallelization easier and manageable. Based on this, the parallel network traffic controller can run in a 1- way 2-stage pipelined fashion on a multi-core processor, which not only increases the processing speed significantly, but also performs well in stability. The remainder of this paper is organized as follows. After related work in Section 2, Section 3 gives description of the problem for multi core systems in cloud applications using Cache Fair Thread Scheduling algorithm and Section 4 gives a result followed by conclusion.

II. RELATED WORK

As cloud computing is a relatively new concept, it is still at the early stage of research. Most of the published works focus on general description of cloud, such as its definition, advantages, challenges, and future [4]-[6]. In detail, security is a very popular and important research field in cloud computing. Some researches focus on the data confidentiality and integrity in cloud computing, and some analyze security problems in cloud computing from different points of view, such as network, servers, storage, system management and application layer [7]. Besides security, another hot topic in cloud computing is virtualization technologies. A security Private Virtual Infrastructure [PVI] and the architecture that cryptographically secures each virtual machine are proposed in [8]. A virtual machine image management system is introduced in [9], and a real-time protection solution for virtual machine is given in [10]. A Run-Time Monitor (RTM) which is a system software to monitor the characteristics of applications at run-time, analyze the collected information, and optimize resources on a cloud node which consists of multi-core processors are described in [18]. To characterize scientific and transactional applications in Cloud infrastructures - IaaS, identifying the best virtual machine configuration in terms of the optimal processor allocation for executing parallel and distributed applications are proposed in [19]. The effect of heterogeneous data on the scheduling mechanisms of the cloud technologies and a comparison of performance of the cloud technologies under virtual and nonvirtual hardware platforms are given in [20]. The number of cores which fit on a single chip is growing at an exponential rate while off-chip main memory bandwidth is growing at a linear rate at best. This core count to off-chip bandwidth disparity causes per-core memory bandwidth to decrease as process technology advances. An analytic model to study the tradeoffs of utilizing increased chip area for larger caches versus more cores is introduced in [21]. In this study on constructing many core architectures well suited for the emerging application space of cloud computing where many independent applications are consolidated onto a single chip is described.

To make CFTS capable for cloud computing, parallelization technology on multi-core processor is required. A research trend on multi-core parallelization is wait-free data structures. A general introduction to the wait free data structures is given in [11].

III. PROBLEM STATEMENT

This section first describes CFTS in detail by comparing it with static scheduling algorithm (deadline monotonic) for multi core systems when running cloud applications. The scheduling primitives must support a wide variety of parallelization requirements. Moreover, some applications need different scheduling strategies for different program phases. In deadline monotonic, the time that a program takes to run will not depend only on its computation requirements but also on the ones

of its co-runners. Therefore, the scheduler not only must select the task to be launched (e.g., a critical task) but also the appropriate core. The core must be selected according to the computational requirements of the tasks already running in each core. Therefore it is measured as time consuming for cloud applications. Also, as the number of tasks (traffic) increases, the static scheduling system employed by a traditional OS will no longer guarantee optimal execution of tasks. Therefore proposed CFTS reduces the effects of unequal CPU cache sharing that occur on these processors and cause unfair CPU sharing, priority inversion, and inadequate CPU accounting. CFTS attempts to minimize the effect of thread level data dependencies and maintain efficient execution of all threads in the processor without excessive overhead for scheduling computation. The proposed thread scheduling performs in parallel with CPU operation by utilizing resources and thereby minimizing the amount of time spent in the OS. In order to schedule the CFTS threads effectively, it must have information regarding the current state of all threads currently executing in the processor. To do gain this knowledge wait free data structures are implemented to store threads and maintain information about their current status. Therefore on parallelizing CFTS by applying wait-free design principles can be used for the allocation and management of shared network resources among different classes of traffic streams with a wide range of performance requirements and traffic characteristics in high-speed packet-switched network architectures. Since the proposed CFTS maximizes the through put, these metrics can then be used for efficient bandwidth management and traffic control in order to achieve high utilization of network resources while maintaining the desired level of service for cloud computing by CFTS scheduling in multi core systems.

A. *Description of Cache Fair Thread scheduling algorithm*

CFTS is suitable for cloud computing for its idea of bandwidth borrowing. It can not only control the bandwidth of all users, guaranteeing that all users could be given different levels of basic service by their payment, but also make more effective usage of free resource and make a better user experience. In the cloud, there could be different kinds of leaf users, e.g. 2Mbps, 5Mbps, regarding different service levels. Because CFTS is a dynamic traffic control mechanism, classes can be added or removed dynamically. This makes CFTS scalable enough for cloud computing.

On real hardware, it is possible to run only a single task at once, so while that one task runs, the other tasks that are waiting for the CPU are at a disadvantage - the current task gets an unfair amount of CPU time. In CFTS this fairness imbalance is expressed and tracked via the per-task p->wait_runtime (nanosec-unit) value. "wait_runtime" is the amount of time the task should now run on the CPU for it to become completely fair and balanced. CFTS's task picking logic is based on this p->wait_runtime value and it is thus very simple: it always tries to run the task with the largest p->wait_runtime value. So CFTS always tries to split up CPU time between runnable tasks as close to 'ideal multitasking hardware' as possible. This algorithm redistributes CPU time to threads to account for unequal cache sharing: if a thread's performance decreases due to unequal cache sharing it gets more time, and vice versa. The challenge in implementing this algorithm is determining how a thread's performance is affected by unequal cache sharing using limited information from the hardware. The cache-fair scheduling algorithm does not establish a new CPU sharing policy but helps *enforce* existing policies. The key part of our algorithm is correctly computing the adjustment to the thread's CPU quantum [13]. The given four-steps are used to compute the cache-fair scheduling algorithm adjustment.

1. Determine a thread's *fair L2 cache miss rate* – a miss rate that the thread would experience under equal cache sharing.
2. Compute the thread's *fair CPI rate* – the cycles per instruction under the fair cache miss rate.
3. Estimate the *fair number of instructions* – the number of instructions the thread would have completed under the existing scheduling policy if it ran at its fair CPI rate (divide the number of cycles by the fair CPI). Then measure the actual number of instructions completed.
4. Estimate how many CPU cycles to give or take away to compensate for the difference between the actual and the fair number of instructions. Adjust the thread's CPU quantum accordingly.

The algorithm works in two phases:

1. Searching phase: The scheduler computes the fair L2 cache miss rate for each thread.

2. Calibration phase: A single calibration consists of computing the adjustment to the thread's CPU quantum and then selecting a thread from the best effort class whose CPU quantum is adjusted to offset the adjustment to the cache-fair thread's quantum. Calibrations are repeated periodically.

The challenge in implementing this algorithm is that in order to correctly compute adjustments to the CPU quanta and need to determine a thread's fair CPI ratio using only limited information from hardware counters [14]. This

algorithm reduces L2 contention by avoiding the simultaneous scheduling of problematic threads while still ensuring real-time constraints.

B. Description of Static Algorithm (Deadline Monotonic)

To meet hard deadlines implies constraints upon the way in which system resources are allocated at runtime. This includes both physical and logical resources. Conventionally, resource allocation is performed by scheduling algorithms whose purpose is to interleave the executions of processes in the system to achieve a pre-determined goal. For hard real-time systems the obvious goal is that no deadline is missed. One scheduling method that has been proposed for hard real-time systems is a type of deadline monotonic algorithm [15] .This is a static priority based algorithm for periodic processes in which the priority of a process is related to its period. With this algorithm, several useful properties, including a schedulability test that is sufficient and necessary the constraints that it imposes on the process system are severe: processes must be periodic, independent and have deadline equal to period. The processes to be scheduled are characterized by the following relationship:

Computation time < deadline < period

Based on this each core is characterized by:

1) The frequency of each core, *fj*, given in cycles per unit time. With DVS, *fj* can vary from *fj min* to *fj max*, where 0 < *fj min* < *fj max*. From frequency it is easy to obtain the speed of the core, *Sj*, which is simply the inverse of the frequency.

2) The specific architecture of a core, *A*(*corej*), includes the type the core, its speed in GHz, I/O, local cache and/or memory in Bytes.

Tasks: Consider a parallel application, *T* = {*t*1, *t*2, …, *tn*}, where *ti* is a task. Each task is characterized by:

1) The computational cycles, *ci*, that it needs to complete. (The assumption here is that the *ci* is known *a priori*.)

2) The specific core architecture type, *A*(*ti*), that it needs to complete its execution.

3) The deadline, *di*, before each task has to complete its execution.

The application, *T*, also has a deadline, *D*, which is met if and only if the deadlines of all its tasks are met. Here, the deadline can be larger than the minimum execution time and represents the time that the user is willing to tolerate because of the performance-energy trade-offs. The number of computational cycles required by *ti* to execute on *corej* is a finite positive number, denoted by *cij*. The execution time of *ti* under a constant speed *Sij*, given in cycles per second is ,

$$tij = cij/Sij.$$

C. Description of Wait free data structures

A wait-free data structure is a *lock-free* data structure with the additional property that every thread accessing the data structure can make complete its operation within a bounded number of steps, regardless of the behaviour of other threads. Each thread is guaranteed to be progressing itself or a cooperative thread [12]. This property means that high-priority threads accessing the data structure never have to wait for low-priority threads to complete their operations on the data structure, and every thread will always be able to make progress when it is scheduled to run by the OS. For real-time or semi-real-time systems this can be an essential property, as the indefinite wait-periods of blocking or non-wait-free lock-free data structures do not allow their use within time-limited operations. A wait-free data structure has the maximum potential for true concurrent access, without the possibility of busy waits.

IV.RESULTS AND DISCUSSION

In the proposed work the time that it takes to complete its work segments in the Cache Fair Thread Scheduler with Wait Free data structures (CFTS-WF) and Cache Fair Thread Scheduler (CFTS) schedules are compared. This quantity is referred to as *completion time.* When running with a static scheduler, the difference between completion times is larger, but when running with the cache fair scheduler, it is significantly smaller. Figure 1 demonstrates normalized completion times with the static scheduler and Figure 2 demonstrates normalized completion times with the Cache Fair Thread scheduler.

Static Scheduler

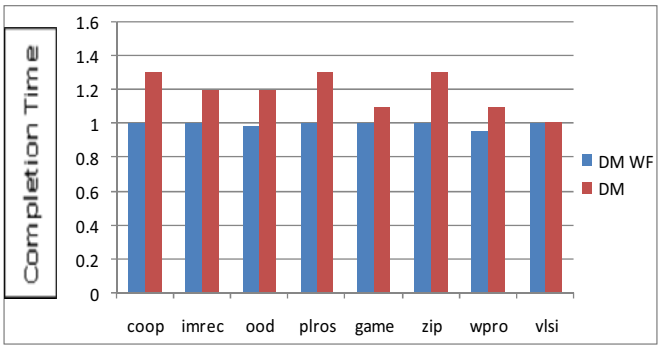

Figure: 1 Performance Variability with Static scheduler

Cache Fair Thread scheduler

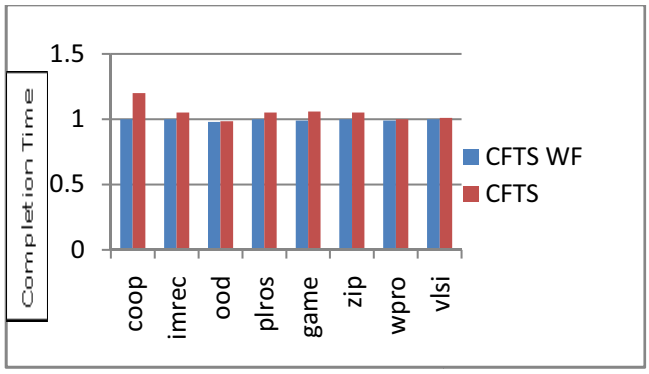

Figure: 2 Performance variability with Cache fair Thread Scheduler

Hence the CFTS seek to maximize the use of concurrency by mapping independent tasks on different threads, so that to minimize the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become executable, and it should seek to minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process.

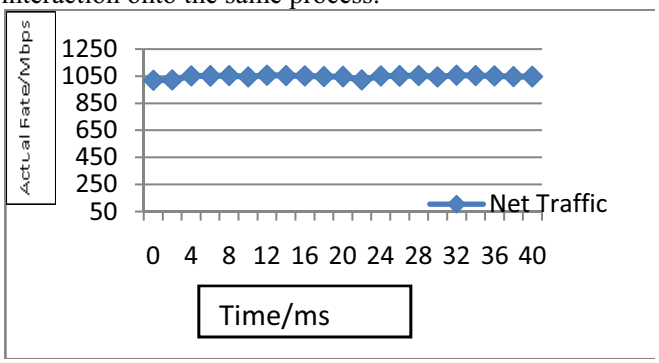

Figure: 3 Output traffic rate of total traffic

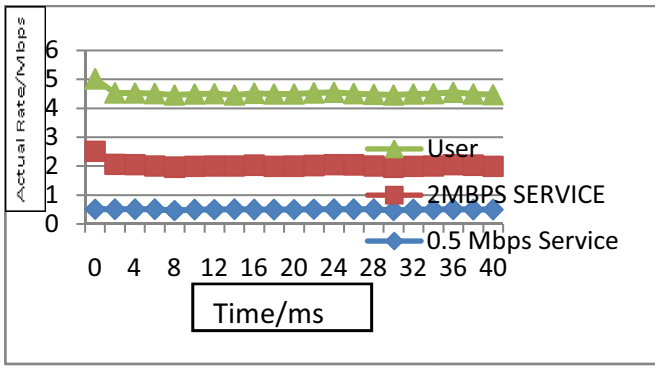

Figure: 4 Output traffic rate of a selected user

It is again designed to test whether the wait –free based CFTS could be competent for cloud scenario described in section 3 under extremely high traffic pressure. Figure 3 and Figure 4 shows the output rate sampling at a certain time interval for both the total CFTS and a random selected user. To make the results more accurate, different time interval for total traffic and user traffic is used, because their rates are at different levels. It is understood that the wait-free FIFO based CFTS performs traffic control quite stable, and the resulting rate lines are nearly smooth, indicating the traffic rates are accurately retained. The stability is important for cloud computing because it might face tens of thousands users at the same time.

V. CONCLUSION

A significant body of the performance modeling research literature has focused on various aspects of the parallel computer scheduling problem and the allocation of computing resources among the parallel jobs submitted for execution. Several classes of scheduling strategies have been proposed for such computing environments, each differing in the way the parallel resources are shared among the jobs. This includes the class of space-sharing strategies that share the processors in space by partitioning them among different parallel jobs, the class of time-sharing strategies that share the processors by rotating them among a set of jobs in time, and the class of scheduling strategies that combine both space-sharing and time-sharing. In this paper, a wait free based parallel CFTS is implemented for effective and stable traffic control in the cloud. Based on the algorithms on accessing data structures and the usage of wait free FIFO, the parallel CFTS can run a pipelined fashion. The experimental analysis and evaluation results both indicate that parallel CFTS is more suitable for cloud computing, due to its excellent performance on both line rate and stability. In future the improvement of CFTS WF for higher line rate may be implemented. Moreover, parallel network application based on multi-core processor is cheaper and more scalable than special hardware and explore its more effective usage in cloud computing.

REFERENCES

1. K. Asanovic and others, The landscape of parallel computing research: A view from Berkley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkley, December (2006).

2. T. Tartalja and V. Milutinovich, The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions, ISBN: 978-0-8186-7096-1, (1996).
3. C. Leiserson and I. Mirman, How to Survive the Multi-core Software Revolution, Cilk Arts, (2009).
4. R. Buyya, "Market-Oriented Cloud Computing: Vision, Hype, and Reality of Delivering Computing as the 5th Utility", Proc. of 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09), Shanghai, China, May, 2009, pp. 1, doi: 10.1109/CCGRID.2009.97.
5. M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing", Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Report No. UCB/EECS-2009-28, CA, USA, 2009.
6. J. Heiser and M. Nicolett, "Accessing the Security Risks of Cloud Computing", Gartner Inc., Stanford, CT,2008,http://www.gartner.com/.
7. M. Yildiz, J. Abawajy, T. Ercan, and A. Bernoth, "A Layered security Approach for Cloud Computing Infrastructure", Proc. of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks, 2009, pp.763-767, doi: 10.1109/I-SPAN.2009.157.
8. F.J. Krautheim, "Private Virtual Infrastructure for Cloud Computing", in HotCloud'09, San Diego, CA, USA, June, 2009.
9. J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment", Proc. Of the ACM workshop on Cloud computing security 2009, Chicago, IL,US, 2009, pp.91-96, doi: 10.1145/1655008.1655021.
10. M. Christodorescu, R. Sailer. D. L. Schales, D. Sgandurra and D. Zamboni, "Cloud Security Is Not (Just) Virtualization Security", Proc. of the ACM workshop on Cloud computing security 2009, Chicago, IL,US, 2009, pp.97-102, doi: 10.1145/1655008.1655022.
11. K. Fraser and T. Harris, "Concurrent programming without locks", ACM Transactions on Computer Systems, vol. 25 (2), May 2007.
12. J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lockfree queue", Proc. of PPoPP'08, New York, NY, USA, February 2008, pp.43-52.
13. Alexandra Fedorova, Margo Seltzer and Michael D. Smith,Harvard University, Sun Microsystems "Cache-Fair Thread Scheduling forMulticore Processors".
14. S. Kim, D. Chandra and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture, In Intl. Conference on Parallell Architectures and Compilation Techniques (PACT), 2004.
15. Ishfaq Ahmad, Sanjay Ranka, Sanjay Ranka, "Using Game Theory for Scheduling Tasks on Multi- Core Processors for Simultaneous Optimization of Performance and Energy", 2008
16. Zheng Li, Nenghai Yu, Zhuo Hao," A Novel Parallel Traffic Control Mechanism for Cloud Computing, 2nd IEEE International Conference on Cloud Computing Technology and Science,2010
17. Lizhe Wang, Jie Tao, Gregor von Laszewski, Holger Marten," Multicores in Cloud Computing: Research Challenges for Applications, Journal of computers, vol. 5, no. 6, june 2010
18. Mikyung Kang , Dong-In Kang, Stephen P. Crago, Gyung-Leen Park and Junghoon Lee , Design and Development of a Run-Time Monitor for Multi-Core Architectures in Cloud Computing , Sensors 2011, 11, 3595-3610
19. Denis R. Ogura, Edson T. Midorikawa, "Characterization of Scientific and Transactional Applications under Multi-core Architectures on Cloud Computing Environment," IEEE International Conference on Computational Science and Engineering, pp. 314-320, 2010
20. Ekanayake, J.; Gunarathne, T.; Qiu, J," Cloud Technologies for Bioinformatics Applications" , IEEE Transactions on Parallel and Distributed Systems, Volume: 22,pp 998 – 1011.
21. Agarwal, Anant; Miller, Jason; Beckmann, Nathan; Wentzlaff, David," Core Count vs Cache Size for Manycore Architectures in the Cloud", CSAIL Technical Reports, Volume 6568/2011, pp.39-50