An Optimized and Efficient Multi Parametric Scheduling Approach for Multi-Core Systems

Sonia Mittal and Priyanka Sharma

***Abstract*—Multi-core processor technologies have become pervasive and mainstream. Several types of multi-core CPUs, including symmetric and asymmetric models, are emerging. A multi-core processor architecture may be defined as: on-chip clusters of heterogeneous functionality modules (processors), cooperating in the implementation of multiple concurrent applications. On these platforms, developing applications that truly take advantage of the power of multi-core capabilities is still a complex, error-prone, and challenging endeavor. Application code must be tuned to optimally fit the available resources. Operating System procedures must cover issues at lower abstraction layers, close to firmware, in order to enable features like optimal task/thread level scheduling depending upon the application requirements on the appropriate processor as per its characteristic. For Efficient scheduling of task or thread on multi core system the operating system scheduler must be aware about the underlying heterogeneity present in the system, also it must be aware about the characteristics of application (at static and at run time). Because as per the characteristic of the executing application the scheduler can take decision to schedule the task on available core so that optimal performance and good throughput can be achieved**.

***Index Terms*—Multicore scheduling, fine grained threads, cooperating thread, load balancing.**

I. Introduction

In Unix operating system the scheduler is known as Round Robin with multilevel feedback, meaning is that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds its time quantum, and feeds it back into one of several priority queues. Priority is not fixed it is dynamically changed, with initially there is some priority assigned to a process by the user later on the priority is calculated as per Equation 1.

$$\text{Priority} = (\text{'recent CPU usage'}/\text{constant}) + \text{Base priority} + \text{Nice value.} \quad (1)$$

Here, the constant is the value calculated by scheduler. The value is assumed to be 2 as per [1] to maintain the priority value into the specified limits.

In Unix SVR4, some changes are made in the scheduling algorithms used earlier UNIX systems. The new algorithm is designed to give highest preference to Real Time processes, next – highest preference to kernel mode process, and lowest preference to other user-mode process, referred to as time –shared process as shown in Table I.

In Unix SVR4 system, Pre-emptable static priority scheduler is implemented with insertion of preemption points. Because the basic kernel is not preemptive, now in between the processing steps safe places known as preemption points have been identified where kernel can safely interrupt processing and schedule a new process [1].

TABLE I: SVR4 Priority Class

Priority class	Global value	Scheduling
Real time	159 to 100	Highest
Kernel	99 to 60	Medium
Time shared	59 to 0	Low

In Multiuser system above scheduling technique does not differentiate between classes of users [1]. The Fair share scheduler (FSS) is implemented on a number of UNIX systems [1]. It includes the user group. FSS considers the execution history of a related group of process, along with the individual execution history of each process in making scheduling decision. The system divides the user community into a set of fair – share groups and allocates a fraction of the processor resource to each group.

David Choffnes et al. [2] had implemented Linux kernel scheduler called the Practical Fair-Share Scheduler (PFS). PFS is a fair-share process scheduler designed to support real-time workloads with soft (i.e., elastic) timeliness requirements. A novel aspect of PFS is its treatment of placement and migration in SMP or multi-core settings. PFS uses a strategy that maintains utilization without un-fairly penalizing processes.

Windows Operating system makes use of a priority – driven preemptive scheduler, threads with real-time priorities have precedence over other threads [3]. In Windows, priority of real time threads are fixed but priority of other threads are changed dynamically. Windows executive raise or lower the priority of I/O bound threads and processor bound threads respectively [3]. In multiprocessor system with *N* processors, the (*N*-1) highest priority threads are always active, running exclusively on the (*N*-1) extra processors. The remaining, lower priority threads share the single remaining processor.

This strategy is affected by the processor affinity attribute of a thread [4]. If a thread is ready to execute but the only available processors are not in its processor affinity set, then thread is forced to wait, and the executive schedules the next available thread.

Multithreaded applications are becoming pervasive due to the emergence of multi core processors. Previously, multithreading has been used primarily to extract concurrency between I/O and computation, but it can also be used to enable concurrent computation on multiprocessor systems. One of the main problems with threads, however, is

Manuscript received September 15, 2012; revised November 30, 2012.

Sonia Mittal is with the MCA, Computer Science & Engineering Dept., Nirma University, Ahmedabad, India (e-mail: sonia.mittal@nirmauni.ac.in). Priyanka Sharma is with the Computer Science and Engineering Department, Institute of Technology, Nirma University.

that their memory access behavior is completely invisible, which makes it challenging to schedule threads for optimal cache utilization and performance [5].

There are basically two ways of multithreading: coarse-grain and fine-grain. For applications with coarse grain multithreading, cache locality is not a very big problem.

In contrast, fine-grain multithreading involves an abundance of threads with frequent communication and short execution times, typically only 100 to 10,000 cycles [6]. Applications with fine grain multithreading have many frequently executed, independent regions of code that can be extracted for parallel execution.

Fine grained parallelism exhibit good performance due to their short execution time. At the same time, it is only suitable when the overheads of scheduling and communication are small, as is the case with multi core systems. Multithreading has the central drawback that is generally not showing cache reusability; threads have a relatively short duration and are context switched frequently without opportunity to leverage the data already in the cache [7]. Hence, the processor must frequently stall to re-populate thread context, and the wait is often substantial. In proposed scheduler it is considered.

In this paper a new approach is proposed in which on the basis of the characteristics of applications certain decisions regarding scheduling can be done. The characteristics of application can be defined by following parameters.

1) The application is compute intensive or I/O intensive.
2) The pattern of application is irregular that manipulate pointer-based data structures like trees and graphs or regular that deal with arrays and dense matrices.
3) The nature of application in terms of data requirement is static or dynamic.
4) Resource requirement and resource sharing.

Identification of I/O intensive task and compute intensive task is done by proposed scheduler is as:

The process working set is analyzed in order to find out the frequency of read( ) and write( ) operations. RTOC and WTOC are use to evaluate whether a task is compute Intensive or I/O intensive. The expression for the same as equation 2 and 3 respectively.

$$\text{RTOC} = \text{No. of read ( ) / no. of compute ( )} \quad (2)$$

$$\text{WTOC} = \text{No. of write ( )/ no. of compute ( )} \quad (3)$$

If RTOC or WTOC value is low from a certain threshold value then task is considered as compute intensive and if RTOC or WTOC value is high from a threshold value then task is considered I/O intensive. This approach is based on prediction.

Further the application which is a mix of small tasks is analyzed to find whether it is computed in parallel or not. Also, it needs to be identified whether application be divided into small and big tasks. i.e. threads. The threads will be scheduled to run concurrently.

II. Proposed Approach

Optimal performance can be exploited by making the process scheduler aware of the Multicore topologies & the task characteristics [8]. In this paper a new approach is proposed to control scheduling decisions based on characteristics of applications. If application is a mix of only small tasks then it can be divided into various coarse grain tasks and then as second level, further it can be divided into fine grained threads. This can further be scheduled on available core as per the characteristic of the thread and core matches to certain degree.

In some applications there is a mix of tasks in which some tasks cannot be executed parallel i.e. serial code so these tasks can be scheduled on faster core and the parallel portion of application can be run on slow cores.

In asymmetric multicore systems cores in the same processor can have different performance [9]. We consider asymmetric multicore systems because they exhibit good performance as compare to homogeneous multicore systems [10]. Some degree of performance asymmetry is beneficial. This is because all applications, whether multi-threaded or single-threaded, have serial portions, and providing a high-performance core helps speed these serial portions. Here we assumed that hardware performance monitors will provide the hardware characteristics to the run-time system.

A. Scheduling

On the basis of application type, at primary level mapping will be done as given in pseudo codes – Fig. 1, Fig. 2 and Fig. 3:

```
Schedular_Compute_Intensive_Task( )
  {
    If Task is pure Compute intensive and cannot be
     executed parallel
    then schedule the task on relatively faster cores.
     Perform Enqueue operation on Ready_queue( ) of
     Faster Core
   Else
    If task can be executed parallel
    then convert it into coarse grain thread and many fine
     grain threads and schedule coarse grain thread on
     faster core and fine grain threads on slow cores.
     Perform Enqueue operation on Ready_queue( ) of
     Faster Core
     Perform Enqueue operation on Ready_queue( ) of
     slow core
  }
```

Fig. 1. Pseudo code for scheduling compute intensive tasks.

```
Schedular_I/O_Intensive_Task( )
  {
    If Task is I/O intensive and its working set is big  (data
          base oriented)
    Then
       If only faster core is free
        Then divide task into coarse grain threads and select
        highest priority tasks among them and schedule it
        on faster cores and others task will be maintained in
        Global pool of tasks.
        (To achieve good efficiency the required data for
        this task is to be included into working set.)
         Perform enqueue operation on ready_queue( ) of
          faster core.
       Else Divided the task into small tasks i.e. fine grain
          threads and schedule them on relatively slow cores.
```

```
        Perform enqueue operation on ready_queue( ) of
        small cores.
}
```

Fig. 2. Pseudo code for scheduling I/O intensive tasks.

```
Schedular_mixed_task()
{
      If the task is a mix of Compute intensive
       and I/O intensive
       then
      Call schedular_Compute_Intensive_Task() for
       compute intensive workload
       and Call Schedular_I/O_Intensive_Task() for I/O
      intensive workload
}
```

Fig. 3. Pseudo code for scheduling mixed tasks.

B. Thread To Core Affinity and Cooperating Thread Scheduling

Round robin preemptive scheduling technique is used in proposed approach. If a high priority task is come into the system then low priority task will be preempted if it is running. When a thread is required to context switched i.e. a new thread is bring into the running state and previously running thread has to put in either ready queue or block queue, the new thread is scheduled such that maximum shared resource utilization is done. Here we consider cache memory. In many systems the local cache level 2 (*L*2) is shared among Core in the same package and level 1 (*L*1) is made private to Core. Generally multicore is coming with minimum 2 level of caches. The threads of a process share global data, and global data is kept in *L*2 cache, so the threads of same process can share common data. Threads are scheduled onto the cores of same package which share *L*2 cache [11].

Here in this approach thread affinity to processor core is considered. When a thread is required to migrate from a package i..e. the thread is pulled out from a processor and this thread is assigned to the core which has highest affinity point and core is idle. Otherwise another core is assigned first time with a base affinity point.

The generalized approach is followed to calculate the affinity point. Whenever Thread is executed on a core the affinity is increased by one unit. So as a result when a Thread is repeatedly executed on same core affinity becomes highest for that core as compared to other core. Thread is migrated to the core that is having second highest affinity point. A Thread to core, affinity matrix is maintained by the scheduler as shown in Table II. In this manner the maximum cache data is utilized among the cooperating threads. As results inter thread communication is decreased. And overall performance would be improved.

TABLE II: THREAD TO CORE AFFINITY MATRIX.

Core / Thread	Core1	Core 2	Core 3
*T*1	2	0	0
*T*2	2	0	1
*T*3	2	0	1
*T*4	1	1	0
*T*5	1	1	0

For example here *T*1, *T*2 ……*T*5 are threads. *T*1 and *T*2 belong to process *P*1. *T*3, *T*4 belong to process *P*2 and *T*5 belongs to process *P*3. Package1 contain Core 1 and Core 2. Package 2 contains Core 3.

When a core is migrated from the current package to other package then *T*2CA value would be decreased by certain factor because now it is not as much affiliated to this core as up to the previous schedule was done on this core. The reducing factor (RDF) is based on cache hit ratio to cache miss ratio of new threads which are scheduled to execute on this core. And it is given as below.

If RDF value is < 1 and *T*2CA > zero
Then
T2CA = T2CA – RDF
Else
No changes in T2CA value

Algorithm: Here in the proposed algorithm (figure : 4), the *T*2CA i.e. Thread to core affinity value for a thread is checked before scheduling to the core, if thread waiting time does not exceeds to threshold_wait_parent i.e. the waiting time for its parent core where its *T*2CA is maximum and its maximum cache hit ratio is guaranteed. So the thread will wait till threshold_wait_parent time so that it can take maximum benefit of the local cache *L*1 data which is private to this. But if thread waiting time is increased than threshold_wait_parent then it will check for second parameter i.e. second level cache *L*2 within the same package. So now the thread will wait till threshold_wait_pakage time so that it can take maximum benefit of the shared cache *L*2 data which is shared to this. But if waiting time exceeds then it will migrate from this package. Otherwise threads throughput will be poor if it excessively waiting on its turn to share resource, but here at this point by migrating the thread will balance the throughput. This strategy will give best result with least recently used (LRU) cache replacement policy.

When a thread has to migrate from a package it will check the T2CA value of cooperating thread. If T2CA value is less than zero then cooperating thread can be moved along with this thread to other nearest located package to improve inter thread communication.

Step 1: *A* Thread *T* with highest waiting time in ready_queue is searched.
Let us assume Thread *T* is found
Step 2: For Thread *T*, Thread to core affinity, *T*2CA values within a package for all core are searched.
Step 3: If highest *T*2CA value core is found free then schedule thread on this core.
Step 4: Otherwise If waiting time of a thread i.e. waiting_time < threshold_wait_parent
Then thread is put back into ready_queue
And next thread is searched.
Go to Step :1
Step 5: Else If waiting time of a thread i.e., waiting_time < threashold_wait_package
Then thread is put back into ready_queue and next thread is searched.
Go to Step: 1
Step 6: Else Migrate thread into other package which is nearest located to current package.

Fig. 4. Thread to core scheduling pseudo code.

C. Nearest Neighbor Affinity

The above approach explained in II B is suitable for SMP like systems but if the system is Non Uniform memory Access (NUMA) the main memory is distributed among cores, then one more information is required to know is Nearest Neighbor affinity [4]. If thread is bringing into other core which is having its own private memory, it is not sharing the memory with previous core where the thread was previously schedule to run. Then the underlying communication network is required to consider the core which is nearest to previous core is desirable core to schedule first. Because now this thread has its data in previous core's main memory so it is required to access from there. To minimize memory access time it is necessary to allocate the core which is topologically near to this core.

III. Related Work

In Reinventing Scheduling [12] for Multicore Systems multicore processors pose unique scheduling problems that require an approach that utilizes the large, but distributed on-chip memory well. So the approach is based on scheduling objects and operations to caches and cores, rather than a traditional scheduler that optimizes for CPU cycle utilization. Predictive Thread-to-Core Assignment on a Heterogeneous Multi-core Processor [13] has a technique which statically determines the approximate phase behavior in a program. This phase behavior and the exhibited execution characteristics of a small set of representative phases are then exploited at runtime to determine likely profitable thread-to-core assignments for later phases of the program.

Several studies (e.g. [14], [15]) suggest that operating system schedulers insufficiently deal with threads that allocate large parts of the shared level 2 cache and thus slow-up threads running on the other core that uses the same cache. The situation is unsatisfactory due to several reasons: First, it can lead to unpredictable execution times and throughput and second, scheduling priorities may loose their effectiveness because of threads running on cores with aggressive "co-runners" (i.e. threads running on another core in the same package). In the scheduling algorithm [14], the threads on a system are grouped into a best effort class and a cache-fair class. Best effort threads are penalized for the sake of performance stability of cache-fair threads, if necessary, but not vide-versa. However, it is taken care, that this does not result in inadequate discrimination of best effort threads. Fairness is enforced by allocating longer time shares to cache-fair threads that suffer from cache-intensive co-runners at the expense of these co-runners, if they are best effort threads.

Two scheduling algorithms proposed for asymmetric single-ISA multicore processors by Becchi et al. [5] and Kumar et al. [16]. Both of them assume a system with two core types ("fast" and "slow") and rely on continuous performance monitoring to determine optimal thread-to-core assignment. Becchi's *IPCdriven* algorithm periodically samples threads' instructions per cycle (IPC) on cores of both types to determine the relative benefit for each thread from running on the faster core. Those threads that have a higher fast-to-slow IPC ratio have a priority in running on the fast core, because they are able to achieve a relatively greater speedup there. Kumar's method uses a similar technique, except that the sampling method is made more robust by using more than one sample per core type per thread.

In addition, Kumar et al. [16] proposed an algorithm that tries to determine a globally optimal assignment by sampling performance of thread groups rather than making local thread-swapping decisions. Both these approaches promise significantly better performance than naïve heterogeneous-agnostic policies with any kind of heterogeneous workload, but they are both difficult to scale to many cores. According to Hass: A scheduler for Heterogeneous Multicore System [17] thread signature is constructed on the basis of micro architectural parameters of thread execution and this information is utilized in thread scheduling by the runtime system. A similar approach has been given in Using OS Observations to improve performance in Multicore Systems [7]. According to this article operating system can use data obtained from dynamic runtime observation of task behavior to ameliorate performance variability and more effectively exploit multicore processor resources.

History-aware resource-based dynamic scheduling for heterogeneous multi-core processors [18] introduces a history-aware, resource-based dynamic scheduler (HARD) for heterogeneous chip multi-processors (CMPs). HARD relies on recording application resource utilization and throughput to adaptively change cores for applications during runtime.

According to Bias Scheduling in Heterogeneous Multi-core Architectures [19], with cores that have different micro architectures and performance the key metrics are identifies that characterizes an application bias, namely the core type that best suits its resource needs. By dynamically monitoring application bias, the operating system is able to match threads to the core type that can maximize system throughput. KAPPI [20] implementation of runtime thread scheduling allows to group together tasks for a sequential execution in order to reduce scheduling overhead.

IV. Conclusion and Future Work

In this approach the scheduler efficiently utilizes the asymmetry of underlying processor. Performance Asymmetry-aware load balancing ensures that the load on each core is proportional to its computing power. Also depending up on the nature of the task compute intensive or I/O intensive the thread to core assignment is done. The scheduler uses intelligent locality-aware scheduling of fine-grain threads. i.e., thread to core affinity, by utilizing private cache L1 and shared on board cache L2. Hence inter thread communication is also minimized. To maintain the throughput and load balancing the thread is migrated to nearest neighbor. So that memory operation time i.e. latency can be minimized if underlying system is NUMA or Distributed memory system. In this paper mainly the cache memory and processor performance asymmetry is addressed through the proposed approach, the application data characteristics like irregular or regular and another parameter related to data requirement is static or dynamic will be considered in future work and implementation of the

approach will be included.

REFERENCES

[1] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986, pp. 248-252

[2] D. Choffnes, M. Astley, and M. J. Ward, "Migration policies for multi-core fair-share scheduling," *Newsletter ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 92-93, January 2008.

[3] W. Stallings, *Operating systems Internals and design Principles*, 6th ed. Pearson 2009, pp. 487- 490.

[4] F. N. Sibai, "Nearest neighbor affinity scheduling in heterogeneous multi-core architectures," *Journal of Computer Science and Technology*, vol. 8, no. 3, Oct. 2008.

[5] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. 3rd Conf. Computing Frontier*, pp. 29-40, 2006.

[6] M. D. Kruijf and G. Stockman, " Footprint-based scheduling," *CS736 Course Project, Fall 2007 Department of Computer Sciences, University of Wisconsin–Madison, CS736 Course Project*, Fall 2007.

[7] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS Observations to improve performance in Multicore systems," *IEEE MICRO*, vol. 28, no. 3, pp. 56-66, May-June 2008.

[8] S. Siddha, V. Pallipadi, and A. Mallick, "Process scheduling challenges in the era of multi-core processors," *Intel Technology Journal*, vol. 11, no. 4, 2007.

[9] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proc. of the 2007 ACM/IEEE Conference on Supercomputing Article* no. 53.

[10] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," *International Symposium on Computer Architecture*, 2005.

[11] A. Fedorova, D. Vengerov, and D. Doucette, "Operating system scheduling on heterogeneous multi core systems," *Workshop on Operating System Support for Heterogeneous Multicore Architecture, PACT'* 2007

[12] B.-W. Silas, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *Proc. 12th conf. Hot Topics in Operating Systems*, pp. 21, 2009.

[13] T. Sondag, V. Krishnamurthy, and H. Rajan, "Predictive thread-to-core assignment on a heterogeneous multi-core processor," in *Proc. 4th Workshop Programming Languages and Operating Systems*, no.7, 2007.

[14] A. Fedorova, M. Seltzer, and M. D. Smith, "Cache-fair thread scheduling for multicore processors," Technical Report TR-17-06, Harvard University, Oct. 2006

[15] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. International Conference Parallel Architectures and Compilation Techniques*, 2004.

[16] R. Kumar, *et al.*, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proc. 31st Annual International Symposium on Computer Architecture - München, Germany, ISCA '04. IEEE Computer Society,* Washington, DC, USA, vol. 64, June 19-23, 2004.

[17] D. Shelepov, A. Fedorova, S. Blagodurov, J. C. S. Alcaide, N. Perez, V. Kumar, and S. J. Z. F. Huang, "HASS: A scheduler for heterogeneous multicore systems," *Newsletter ACM SIGOPS Operating Systems Review*, vol. 4, Issue 2, pp. 66-75, April 2009.

[18] A. Z. Jooya, A. Baniasadi, and M. Analoui, "History-aware, resource-based dynamic scheduling for heterogeneous multi-core processors," *Computer and Digital Techniques, IET*, vol. 5, no. 4, July 2011, pp. 254-262.

[19] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. 5th European Conf. Computer Systems*, pp. 125-138, April 13-16, 2010.

[20] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. International Conference Parallel Symbolic Computation* 2007, pp. 15-23.

Sonia Mittal received her Master of Computer Applications degree from MBM Engineering College, JNV University in 1994 and Bachelor in Science from JNV University in 1990. Currently; she is associated as Assistant Professor in MCA, Computer Science & Engineering Dept., Nirma University, Ahmedabad, India. She has teaching experience around 14 years. Her Research interest area includes Parallel Processing and Multicore Computing.

Priyanka Sharma received her Masters in Computer Science and Engineering from Nirma University in 2007 and Bachelors Degree in Computer Engineering from Gujarat University in 1999. She has over 13 years of working experience including Industrial and Academia. She is pursuing her PhD in the area of QoS based routing of Multimedia Traffic over IP based networks. She has published around fourteen research papers in International Journal and Conferences. She is currently working as an Associate Professor with Computer Science and Engineering Department, Institute of Technology, Nirma University. She has guided more than 11 MTech Thesis so far. Her research area includes applications of Machine Learning in Wireless Communication and High Speed networks.