Performance Analysis of Homogeneous and Heterogeneous Multicore Processor Using Static and Dynamic Schedulers

A.S. Radhamani
Department of Computer Science and Engineering,
Manonmanium Sundaranar University, Tirunelveli, Tamil Nadu, India

Abstract: Now a days, there is a need in the design of multicore processor in order to improve system performance, and reducing power consumption. Homogenous cores are all exactly the same: equivalent frequencies, cache sizes, functions, etc. However, each core in a heterogeneous system may have a different function, frequency, memory model, etc. There is an apparent trade-off between processor complexity and customization. This study presents a comparison between homogeneous multicore processor and heterogeneous one. This study examines in detail the performance and power consumption by increasing the number of cores in Intel processor. In this study, four different scheduling algorithms for finding near-optimal thread to core assignments in a multicore processor is explored with and without wait free data structure is integrated with each scheduling paradigm. The results demonstrate that heterogeneous multicore architecture can provide significantly higher performance than a homogeneous chip multiprocessor. It does so by matching the various jobs of a diverse workload to the various cores.

Key words: Multi-core, homogeneous, heterogeneous, scheduling, wait free datastructure, performance, power consumption

INTRODUCTION

A multicore processor has multiple cores integrated on a single chip. A multicorearchitecture where every core is just an image of the other is called homogeneous multicore. Heterogeneous is a set of cores which may differ in area, performance, power dissipated etc. The various design issues in multicore architecture include resource sharing power consumption, performance, area of the cache, cache coherence etc. In order to harness the resources provided by a multicore architecture application must show a certain level of parallelism (Balakrishnan, 2005; Kumar *et al.*, 2005).

The increase in need for machines with higher performance, computational power and increase in complexity in the design of uniprocessor has been the inspiration for increase in interest in design of multicore architecture (Kumar *et al.*, 2007). In order to satisfy the high-performance and low-power requirements for advanced multicore architecture with greater flexibility, it is necessary to develop parallel processing on chips by taking advantage of the advances being made in semiconductor integration. Ongoing progress in processor designs has enabled servers to continue delivering increased performance which in turn helps fuel the powerful applications that support rapid business growth. However, increased performance incurs a corresponding increase in processor power consumption and heat is a consequence of power use.

At first glance, scheduling a multicore processor might not appear to present a substantially new problem for operating systems. There is a long history of OS scheduling for multithreaded microprocessors and traditional multichip multiprocessors. There has even been recent research into one aspect of scheduling that is unique to multicores, which is that processors often share L2 caches (Fedorova *et al.*, 2005, 2007).

We propose a new approach to scheduling on both homogeneous and heterogeneous multicore systems based on static and dynamic scheduling strategies. Currently, the operating systems do not have any proper scheduling algorithm to improve the performance and power competence on core platforms efficiently. In this study two scheduling algorithms for multi core processors namely deadline monotonic and cache fair thread scheduling are evaluated and finally, scheduling algorithm that takes the advantages of batching with an exclusively wait-free data structure is implemented. The goal of these scheduling algorithms are to maximize system utility, performance and fairness by assigning application to cores over a fixed, comparatively short period of time.

The primary goals of our scheduling framework are to improve application throughput and overall system utilization. A secondary goal of the framework is to reduce the power consumption as the number of core increases to make good forward progress. The goal of this design is to enable a system to run more tasks simultaneously and thereby achieve greater overall system performance. The following sections are arranged as follows.

Literature review: There are a number of challenges involved in designing a portable and easy-to-use, yet high performance scheduling interface for multicore platforms. For example, it will be essential to reduce shared cache misses since the on-die caches of many-core processors will be relatively small for at least the next several years, and memory latencies have grown to several hundred cycles (Radhamani and Baburaj, 2011). Similarly, choosing which threads run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core Since, Simultaneous-Multithreaded (SMT) cores share low-level hardware resources such as TLBs among all threads.

The two most major heterogeneity-aware scheduling algorithms were described by Becchi and Crowley (2006) and Kumar *et al.* (2004). The continuous monitoring of performance of each thread on each type of core to determine the best thread-to-core assignment for the system is focused by Becchi's algorithm. Kumar's algorithm relied similar approach as well as another technique where selected thread-to-core assignments were tried by the operating system and then the best-performing assignment was used there after.

A user-mode scheduler prototype that determines appropriate cores for threads using signatures was built and evaluated by Daniel and Alexandra in. They found that architectural signatures are good predictors of sensitivity to clock frequency and that the scheduler can improve performance using them. But, a major draw back of the signature-based scheduling is that it does not take into account dynamic phase behavior of the application (Sherwood *et al.*, 2002). While heterogeneity-aware scheduling algorithms were proposed in the past (Becchi and Crowley, 2006; Kumar *et al.*, 2003), they were targeted at small-scale multicore systems and assumed long-lived threads. Heterogeneous architectures are motivated by their potential to achieve a higher performance per watt than comparable homogeneous systems (Kumar *et al.*, 2003). Heterogeneity also can be beneficial in systems with multithreaded cores. Despite the additional scheduling complexity that simultaneous multithreading cores pose due to an explosion in the possible assignment permutations, effective assignment policies can be formulated that do derive significant benefit from heterogeneity (Kumar *et al.*, 2004). Building a performance heterogeneous core is desired because many simple cores together provide high parallel performance while complex cores help in providing high serial performance (Balakrishnan *et al.*, 2005). From Amdahl's law it can be concluded that the serial execution plays an important part in overall performance of the system (Hennessy and Patterson, 2006).

The ability to dynamically switch between different cores, and power down unused cores is the key in asymmetric chip multiprocessing. It was shown that a representative heterogeneous processor using two core types achieves as much as a 63% performance improvement over an equivalent-area homogenous processor (Kumar *et al.*, 2003). Heterogeneous multiprocessors achieve better coverage of a spectrum of load levels.

MATERIALS AND METHODS

Proposed scheduling approach: In this study, explanation of Static Scheduling algorithm (Deadline Monotonic DM) followed by dynamic algorithm (Cache Fair Thread Scheduling CFTS) for homogeneous and heterogeneous multi core system when running different workloads are compared. The given scheduling primitives support a wide variety of parallelization necessities. The proposed approach for scheduling both periodic and aperiodic tasks in a multi-core system consists of two parts. First, periodic tasks with hard deadlines are scheduled. Based on the chosen periodic task schedule, the latest start times for each task are determined as is the slack information for each resource. These steps are performed off-line. The second part, aperiodic task scheduling, is performed for each incoming aperiodic task is dynamically scheduled based on its computation time on each resource and the available slack on each resource.

In deadline monotonic, execution time depends on computation requirements. Hence, the scheduler has to find some task and also the suitable core for execution. In this algorithm core selection is done based on the computational necessities of the tasks. If some cores are busy with the already running tasks, then the current task should wait for some predefined time and hence, it is considered as a long procedure in high performance applications. The major drawback of static deadline monotonic scheduling is as the traffic increases it will not produce an optimum result. Therefore, to improve the performance and to manage real time tasks CFTS is proposed. In CFTS, a global queue of ready threads are
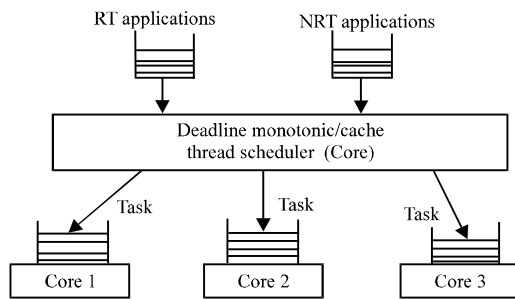
Fig. 1: Task partitioning and scheduling by the multicore processor

maintained and each core when idle, selects a thread from the queue thus, minimizes the effect of data dependencies and maintains efficient implementation of all the thread in the cores. In CFTS, scheduling, parallelization is maintained to manage the tasks and also to use the resources efficiently. For better CFTS scheduling, it must have the knowledge about all the threads currently executing in the core. To maintain information about the current status of the threads, wait free data structure is used to store the thread. Thus, parallelizing CFTS with wait free data structures can assist in handling different type of traffic streams and manages shared network resources. Therefore, the proposed CFTS-WF, utilizes the network resources and services of cloud to provide maximum through put and better band width management. Figure 1 shows the task partitioning and scheduling by the multicore processor.

Description of Static algorithm (deadline monotonic): Scheduling is the process of deciding how to assign resources between varieties of possible tasks. Also, it is a method by which threads, processes or data flow are given access to the system resources, e.g., processor time, communication bandwidth. This is usually done to balance a system effectively or to achieve a target quality of service. The need for scheduling algorithm arises from the requirement of most modern multicore systems to perform multitasking, execute more than one process at a time. Deadline monotonic is a static priority based scheduling where the priority of the process is concerned with the time of task or application. The main goal of deadline monotonic is, for real time systems no deadline to be missed. Each task has a different execution time on each resource. In essence, each task has been implemented for each possible resource type by compiling different software. The main constraint to be satisfied by this algorithm is tasks should be periodic, independent and have deadline equal. The tasks scheduled to be are described by the relationship: computation time<deadline<period. Tasks are given

priorities based on their deadlines, so that the task with shortest deadline is given highest priority. This method of priority assignment is optimal for periodic or sporadic tasks with the following restrictions:

- Deadlines of all tasks are equal or less than their minimum inter arrival time
- The Maximum Length Execution Times (MLET) of all tasks is equal or less than their deadline
- Tasks must not be dependent in order to avoid blocking
- No task voluntarily suspends itself
- At the critical instant all tasks must be executed simultaneously
- Overheads in switching from one task to another must be negligible
- All tasks have zero release jitter

If restriction 7 is elevated then "deadline minus jitter" monotonic priority assignment is optimal. If restriction 1 is elevated allowing deadlines greater than periods. If the number of computational cycles of a task is assumed as t_i on core c_{ij} by a constant speed S_{ij}, given in cycles per second then the execution time of t_{ij} is given by:

$$t_{ij} = c_{ij}/S_{ij}$$

In heavy overload conditions (real time applications), the deadline monotonic algorithm may lead to many tasks missing their deadlines in multicore systems. Focusing at this problem, a novel scheduling algorithm which considers all parameters, and provides priority to real time task is implemented using CFTS.

The description of Cache Fair Thread Scheduling algorithm: The CFTS algorithm focuses on fair cache allocation. In some multicore processors cache allocation is hidden from the operating system. In processors like Intel duo core, they allow cache lines measurement allotted to each co-runner, they provide no steps to implement fair cache sharing. If the OS schedulers pay no attention to fair cache allocation of cache, it tends to three severe problems that can make the OS ineffective. The first problem occurs, when one thread fails in opposing for enough cache freedom essential for further progress and is known as thread starvation. When, the lower priority leads the advanced priority thread, the second problem named priority inversion takes place. When the onward growth rate is extremely reliant on the thread mix in a co-schedule which leads to the third problem called inadequate CPU accounting which make it difficult to predict the forward progress, therefore system performance can be reduced.

The CFTS mechanism reduces the effects of uneven CPU cache distribution on threads routine, hence increases the system performance stability which is suitable for cloud applications. This algorithm calculates every thread CPI (Cycles Per Instructions) under balanced cache sharing and if any deviation occurs CFTS compensate it by regulating the CPU time quantum. On real hardware, when one task runs, the other tasks have to wait for the CPU which lead to unfair amount of CPU time for the existing task. This fairness imbalance in CFTS is expressed in terms of per task p≥"wait runtime" (nanosecond unit) which is the amount of time the task should run on the CPU for the better fairness and balanced condition. Based on this value of "wait runtime", the CFTS runs the task with the leading "wait runtime" the first. This algorithm determines how a thread performance is affected by unequal cache sharing and is the challenge in the implementation. The major part of this algorithm is it calculates the adjustment to the thread CPU quantum.

The CFTS allows us to maximize sharing through the L2 (last-level) caches while making use of the existing scheduling policies to achieve good load balancing. In general, our goal is to improve performance by minimizing contention for different cores and by maximizing the performance. In this scheme, the programmer identifies closely-related threads-threads that share data-and the runtime will do its best to schedule the threads close together so that they share data through nearby shared caches.

Cache miss ratio can be avoided by allocating the threads that belong to the same process in the same core. In order to avoid overhead, more than one process must not be allotted to the same core. Maintain load balancing between the cores.

Description of wait free data structures: A wait free data structure is a special case of lock free data structure with the unique characteristic that every thread can complete their operation within the restricted number of steps, not considering the behavior of other threads. Each thread is characterized to be succeeding itself (or) a supportive thread which makes the higher priority threads accessing the data structure never have to wait for the low priority threads to complete their operations. Wait free data structure uses concurrent access to cooperatively progress the data structure as a whole, which can be an essential property for real time applications.

Cache Fair Thread Scheduling with Wait Free data structure (CFTS WF): At present, there are different types of scheduling algorithms are available viz shortest job first, round robin, etc. each with its own merits and demerits. Scheduling algorithms to be of priority based since tasks of some category need an immediate attention while others can be listened later. All threads, accessing the wait free data structure can complete its process within a restricted number of steps not considering the behavior of other threads. Therefore, when batching CFTS scheduling with wait free data structure where all the threads are given a priority based on the either by the programmer (via system API) or the operating system. There are several scheduling queues are available each with its own priority. Initially the scheduler tries to schedule threads waiting with top priority queue then subsequently queue with low priority. Thus, no queue left unattended. Hence, CFTS WF schedules such that maximum throughput, minimum response time, minimum waiting time and utmost CPU consumption can be obtained.

RESULTS AND DISCUSSION

In this research initially, the time taken to complete seven different applications is calculated with four types of schedulers. This time is referred as completion time. Table 1 describes the different applications chosen for the simulation. Figure 1 and 2 shows the result of various schedulers where seven applications are running simultaneously on a system with variations in the cache and core sizes. Based on the cache size, number of cores and the use of threads in each of these schedulers for different applications the average completion time are recorded. The difference in completion time is larger for static deadline monotonic scheduler which is considerably smaller with cache fair scheduler. Also, both the schedulers are combined with the wait free data structure and the performance is computed. Among, all the schedulers CFTSWF provides minimum completion time by concurrent mapping of independent tasks on different threads by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become executable. The variability is significantly
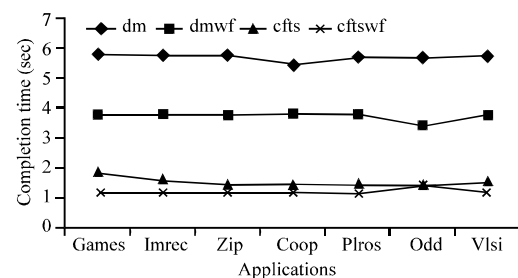

Fig. 2: Completion time with core-2 and cache-2 in different applications

Table 1: Applications

Name of the Application	Expansion	Lines of code	Memory requirement (MB)	Execution time (seconds)
Games	Game playing:	1000	4	10
Imrec	A compiler	1500	4	10
Zip	A compression utility	1200	4	10
Coop	Combinatorial optimization	2000	4	15
Plros	Place and route simulator	3000	8	18
Ood	Object-oriented database	2500	8	16
Vlsi	FPGA circuit placement and routing	2500	8	16
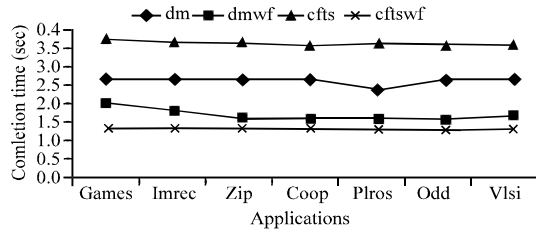
Fig. 3: Completion time with core-4 and cache-2 in different Applications
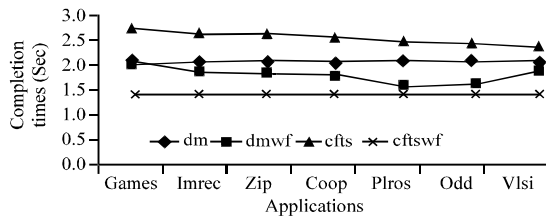
Fig. 4: Completion time with core-8 and cache-2 in different applications

Fig. 5: Task completion time with 32 core-cache 2
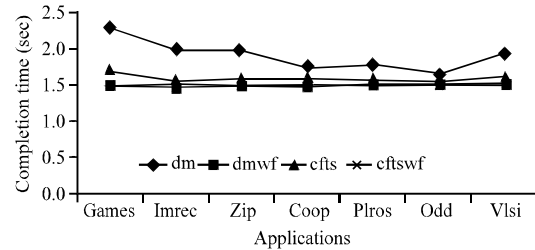
Fig. 6: Task completion time with 64 core-cache 2 in different applications

smaller. All the schedulers performance is tested with varying core sizes and cache sizes. Figure 2 shows the result of various schedulers where seven applications are running simultaneously on a system with the core si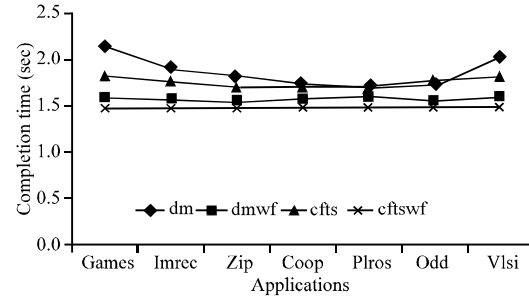ze 2 for a cache size of 2. From the graph, it is observed that the application Games, consumed high completion time with static schedulers than dynamic schedulers which is due the fair cache allocation of threads in CFTS and CFTSWF. For other applications no significant changes observed.

Figure 3 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 4 for a cache size of 2. From the graph, it is observed that the application "plros" consumed less completion time with static scheduler DMWF than DM. But with dynamic schedulers, the completion time is significantly reduced which is due the fair cache allocation of threads in CFTS and CFTSWF. For other applications no significant changes observed.

Figure 4 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 8 for a cache size of 2. From

the graph, it is observed that the application "plros" consumed less completion time with static scheduler DMWF than DM. But with dynamic schedulers, the completion time is significantly reduced which is due the fair cache allocation of threads in CFTS and CFTSWF. For other applications no significant changes observed.

Figure 5 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 32 for a cache size of 2. From the graph, it is observed that all applications consumed almost same completion time with static scheduler DMWF than DM as well as with dynamic schedulers CFTS and CFTSWF because the number of cores increases significantly.

Figure 6 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 64 for a cache size of 2. From the graph, it is observed that all applications consumed almost same completion time with static scheduler DMWF and dynamic schedulers CFTS and CFTSWF because the number of cores increases significantly.
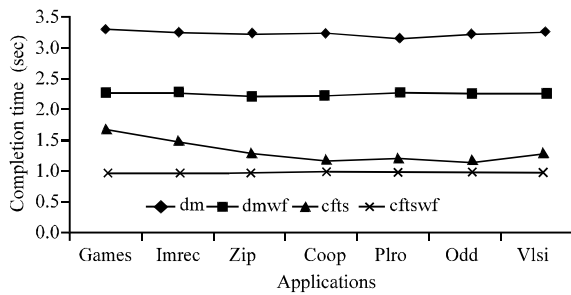
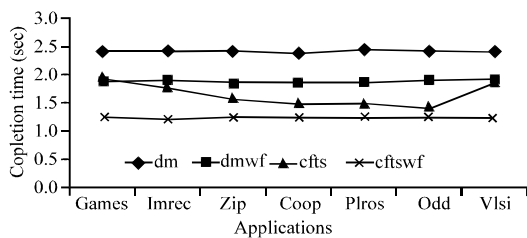Fig. 7: Completion time with 2 core-cache 4 in different applications

Fig. 8: Completion time withcore-4 and cache-4 in different applications
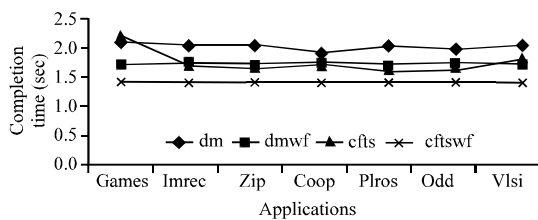

Fig. 9: Task completion time with 8 core-cache 4 in different applications

Figure 7 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 2 for a cache size of 4. From the graph it is observed that for all applications the completion time reduced twice than that of Fig. 2 as the cache is increased.

Figure 8 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 4 for a cache size of 4. From the graph it is observed that it takes less completion time both static schedulers DM, DMWF as well as with dynamic schedulers which is due the fair cache allocation of threads as the cache size increased.

Figure 9 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 8 for a cache size of 4. From the graph, it is observed that it takes less completion time
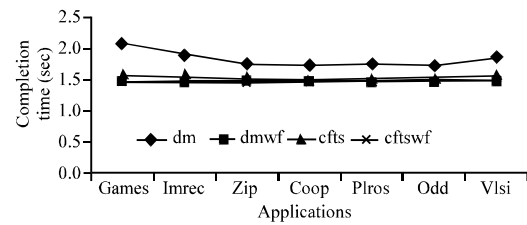

Fig. 10: Completion time with core-32 and Cache-4 in different applications
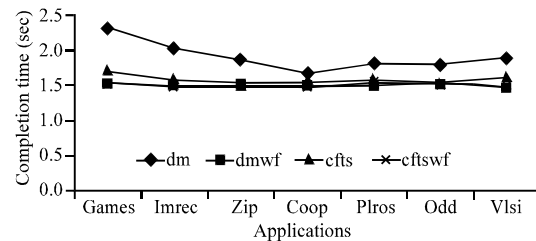

Fig. 11: Task completion time with 64 core-cache 4 in different applications

both static schedulers DM, DMWF as well as with dynamic schedulers CFTS, CFTSWF for all applications which is due the fair cache allocation of threads as the cache size increased except some applications like games, imrec and vlsi with DM as the cache requirement is high.

Figure 10 shows the result of various schedulers where seven applications are running simultaneously on a system with the core size 32 for a cache size of 4. From the graph, it is observed that it takes less completion time both static schedulers DM, DMWF as well as with dynamic schedulers CFTS, CFTSWF for all applications which is due the fair cache allocation of threads as the cache size increased also the number cores increases to 32, hence applications are distributed to many cores.

Figure 11 shows the result of various schedulers where seven applications are running simultaneously, on a system with the core size 64 for a cache size of 4. From the graph, it is observed that it takes less completion time both static schedulers DM, DMWF as well as with dynamic schedulers CFTS, CFTSWF for all applications which is due the fair cache allocation of threads as the cache size increased also the number cores increases to 64, hence applications are distributed to many cores.

Table 2 describes the average completion time with different types of schedulers. From the plots and table, it is found that for vlsi, the completion time was comparatively less than the other type of applications. For zip, the difference was reduced by a factor of five which ranges from 17-2%. One application ood that

Table: 2 Average Task completion Time for the various applications with different schedulers for different in core and cache sizes.

	Cache 2 (average) completion time (sec)					Cache 4 (average) completion time (sec)				
Algorith	Core 2	Core 4	Core 8	Core 32	Core 64	Core 2	Core 4	Core 8	Core 32	Core 64
DM	5.697163	3.644863	2.641438	1.792625	1.631738	3.277863	2.36811	2.054375	1.576663	1.615238
DMWF	3.756367	2.629525	2.069588	1.5837	1.490638	2.264238	1.88635	1.699675	1.466475	1.495488
CFTS	1.621463	1.837113	1.94430	1.969813	1.769644	1.473738	1.77955	1.895813	1.967576	2.00390
CFTSWF	1.130538	1.31475	1.411963	1.4779	1.493875	0.949738	1.250438	1.375857	1.46535	1.49445

experienced only small performance variability with the deadline monotonic scheduler, experienced a slightly higher variability with the CFTS-WF; this was due to variations of thread in the model. Also, the cache-fair thread scheduler with wait free data structures improved performance variability by at least a factor of three for the all applications. In particular, plros has longer average completion times with the CFTS as their cache requirements are high. But, for the same application the completion time is significantly reduced with CFTS-WF. Also, applications with low cache requirements such as zip and vlsi get shorter completion times with the CFTS-WF. Such effect on absolute performance is expected. The goal of the CFTS-WF is to reduce the effects of imbalanced cache sharing on performance. However, all applications experience improved performance stability and predictability with CFTS with wait free data structures. Also, it is observed that as the cache size increases the completion time of the various applications have been reduced considerably. From Fig. 2 and 3, it is clear that as the number of core increases there is a negligible increase in the completion time which is due to the time spent in the selection of the cores. The results indicate that for some applications, having a small fraction of the cache is sufficient to achieve performance close to the performance achieved with the entire cache.

Completion time performance with heterogeneous multicore systems: Although, most current multicore processors are homogeneous, microarchitects are now proposing heterogeneous core implementations including systems in which heterogeneity is introduced at runtime. Therefore, in this research by randomly varying core voltage and frequency of each core, experiments are conducted to test the different performance characteristics. Thus, it will be preferable for each core to have a different frequency rather than derate the entire chip to the lowest-common maximum frequency, particularly as core counts continue to increase. Performance characteristics such as completion time and power consumption is analysed by varying cache sizes with the above static (DM, DMWF) schedulers and dynamic schedulers (CFTS, CFTSWF). Figure 12 shows the completion time vs number of cores for the cache size 2mb. The simulations show a significant reduction in
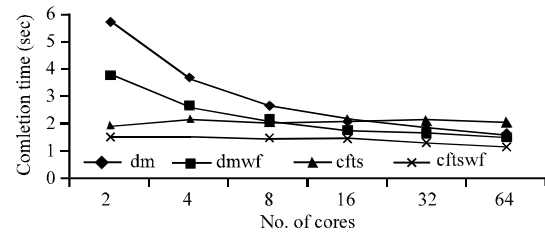

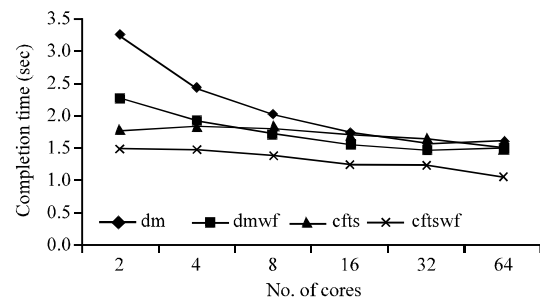Fig. 12: Completion time vs. Number of cores (cache size 2 MB)

Fig. 13: Completion time vs. number of cores (cache size 4 MB)

completion time going from two to four multicores but an insignificant reduction from four to eight multicores in both static and dynamic methods. Exceeding eight multicores causes a steep decline in completion time for static schedulers. After eight multicores, a uniform reduction is registered as more cores are added for dynamic scheduler is due to the fair allocation of threads. Also cores with the lowest frequency are mapped with to the large task set where as the highest frequency core are mapped with small task set. The experiment ends when each thread has run at least once, that is until the longest thread finishes. While, the longest thread is not finished, the other threads restart their executions as soon as they are finished.

Figure 13 shows the completion time vs number of cores for the cache size 4MB. As the above plot, the simulations show a significant reduction in completion time going from two to four multicores but an insignificant reduction from four to eight multicores in both static and dynamic methods. Exceeding eight multicores causes a steep decline in completion time for static schedulers.

After eight multicores, a uniform reduction is registered as more cores are added for dynamic scheduler is due to the fair allocation of threads. Most graphs show monotonically decreasing completion times as the cache size is increased as expected. However, there are a few exceptions. For instance, CFTS shows increasing completion time as the cache size is increased and core numbers from 2-4 due to some application could exhibit streaming behavior consisting of high L2 cache access frequency and no reuse frequency, leading to a high miss frequency.

The experimental results demonstrate that the completion time of the generated schedule is reduced by our approach due to the decrease of the L2 cache miss rate for both scheme (CFTS and CFTSWF), because the number of inter-thread cache interferences on a shared L2 cache is reduced. Therefore, the proposed algorithms are likely to generate successful schedules with stricter timing constraints for the task model studied, compared to the static scheduling approaches that are not aware of the inter thread interference using WCET and scheduling on multicore platforms. In general, the completion time of the dynamic algorithms is better than the static algorithms.

Power performance with heterogeneous multicore systems: Heterogeneous multi-core processors are attractive for power efficient computing because of their ability to meet varied resource requirements of diverse applications in a workload. However, one of the challenges of using a heterogeneous multi-core processor is to schedule different programs in a workload to matching cores that can deliver the most efficient power consumption. The dynamic power consumption of a core is a time varying function of the core speed s(t), voltage v(t), and the task allocated to it. Hence, dynamic power consumption is calculated by:

$$P_{core} = C_{active} \times P^{core\ active} + _{Cidle} * P^{core\ idle} \qquad (2)$$

$$P^{core\ active} = C_{dd} \times V^2 \times f \qquad (3)$$

Where:
C_{dd} = The switched circuit capacitance
V = The supply voltage
f = The clock frequency

Using the equations 2 and 3 multi-core processors can provide high power consumption since they can allow the clock frequency and supply voltage to be reduced together to dramatically reduce power dissipation during periods when full rate computation is not needed. From the following Fig. 14, it is observed that when a number of
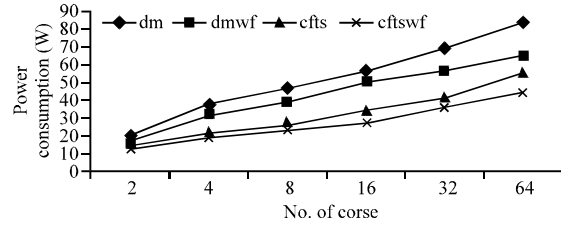

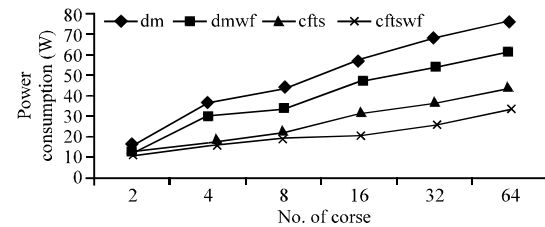Fig. 14: Power consumption vs. Number of cores (cache size 2 MB)

Fig. 15: Power consumption vs. Number of cores (cache size 4 MB)

core are less (2 and 4), the hardware is fully utilized. When, the number of core increases there is a possibility of cache miss, the processor waits, consumes power and performs no useful work. Therefore, with multithreading in CFTS and CFTSWF the processor launch a second thread that runs while the first is waiting and a third when the second thread waits, etc. then the hardware stays fully utilized all the time. In this way, multicore processor can improve performance with out impacting power consumption.

As the cache size increased, the performance are significant and is shown in Fig. 15. It is due to the fact that the possibility of cache miss is notably reduced and hence power consumption is below 40 watts for dynamic power consumption than the static as there is a steep increase in power is noticed as the number of core increases and reaches the maximum of 80 Watts which is double the amount of dynamic scheduling.

CONCLUSION

In the above study, experiments are conducted based on homogeneous and heterogeneous multicore processor architectures and the advantage of which is that the design allows the operating system to assign workloads to any of the processing units at any time. However, although design architectures in this category can deliver faster compute performance compared to a traditional single core processor, eventually performance gains become limited by power consumption. Performance of

heterogeneous multicore processors consisting of multiple cores with the same instruction set but different performance characteristics (e.g., clock speed, core voltage) are of great concern since, they are able to deliver higher performance per watt and area for programs with diverse architectural requirements than comparable homogeneous ones. And hence, with the same static schedulers (DM, DMWF) and dynamic schedulers (CFTS, CFTSWF), performance is analysed. For the threads running on these cores, their properties and resource demands may and often will be different. And hence performance improvement is realized when the scheduler map threads to cores in a way that will maximize the utilization of resources on each other With CFTSWF measuring threads efficiencies in utilizing fast cores is highly significant. We have shown that as the thread on all core types are the same and proved that the speedup factor of a thread is monotonically decreasing in CFTSWF and hence the completion time. Furthermore, an efficient power consumption analysis with the above scheduler in the heterogeneous environment is provided. The results show that high power consumption is achieved with heterogeneous processors, since they can allow the clock frequency and supply voltage to be reduced together to dramatically reduce power dissipation during periods when full rate computation is not needed. In our current scheduling framework, we target single many-core processors. We expect to extend our framework to multiple many-core processors in the future.

ACKNOWLEDGEMENTS

REFERENCES

Balakrishnan, S., R. Rajwar, M. Upton and K. Lai, 2005. The impact of performance asymmetry in emerging multicore architectures. Proceedings of the 32nd Annual International Symposium on ACM SIGARCH Computer Architecture News, June 4-8, 2005, IEEE Computer Society, Washington, USA., ISBN: 0-7695-2270-X, pp: 506-517.

Becchi, M. and P. Crowley, 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. Proceedings of the 3rd Conference on Computing Frontiers, May 3-5, 2006, ACM, New York, USA., ISBN: 1-59593-302-6, pp: 29-40.

Fedorova, A., M. Seltzer, C. Small and D. Nussbaum, 2005. Performance of multithreaded chip multiprocessors and implications for operating system design. Proceedings of the Annual Conference on USENIX Annual Technical Conference, April 10-15, USENIX Association Berkeley CA, USA., pp: 26-26.

Fedorova, A., M. Seltzer and M.D. Smith, 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, September 15-19, 2007, IEEE Computer Society, Washington, DC., USA., ISBN: 0-7695-2944-5, pp: 25-38.

Hennessy, J.L. and D.A. Patterson, 2006. Computer Architecture: A Quantitative Approach. The 4th Edn., Morgan Kaufmann, San Francisco, ISBN: 978-0-12-370490-0.

Kumar, R., K.I. Farkas, N.P. Jouppi, P. Ranganathan and D.M. Tullsen, 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003, MICRO-36, December 3-5, 2003, IEEE, USA., ISBN: 0-7695-2043-X, pp: 81-92.

Kumar, R., D.M. Tullsen, P. Ranganathan, N.P. Jouppi and K.I. Farkas, 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. ACM. SIGARCH. Comput. Archit. News, 32: 64-75.

Kumar, R., D.M. Tullsen, N.P. Jouppi and P. Ranganathan, 2005. Heterogeneous chip multiprocessors. Comput., 11: 32-38.

Kumar, S., C.J. Hughes and A. Nguyen, 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. ACM. SIGARCH. Comput. Archit. News, 35: 35-173.

Radhamani, A.S. and E. Baburaj, 2011. Implementation of Cache Fair Thread Scheduling for multi core processors using wait free data structures in cloud computing applications. Proceedings of the World Congress on Information and Communication Technologies, December 11-14, 2011, Mumbai, pp: 600-605.

Sherwood, T., E. Perelman, G. Hamerly and B. Calder, 2002. Automatically characterizing large scale program behavior. ACM. SIGOPS. Operating Syst. Rev., 36: 45-57.