# Reducing Shared Cache Misses via dynamic Grouping and Scheduling on Multicores

Wael Amr Hossam El Din
Computer Department ,Faculty of Engineering,
Cairo University,
Giza,Egypt
wael.amrhossam@gmail.com

Hany Mohamed ElSayed
Communication Department ,Faculty of Engineering,
Cairo University,
Giza,Egypt
helsayed@ieee.org

Ihab ElSayed Talkhan
Computer Department,
Faculty of Engineering,
Cairo University,
Giza,Egypt
italkhan@aucegypt.edu

***Abstract*—Multicore technology enables the system to perform more tasks with higher overall system performance. However, this performance can't be exploited well due to the high miss rate in the second level shared cache among the cores which represents one of the multicore's challenges.**

**This paper addresses the dynamic co-scheduling of tasks in multicore real-time systems. The focus is on the basic idea of the megatask technique for grouping the tasks that may affect the shared cache miss rate ,and the Pfair scheduling that is then used for reducing the concurrency within the grouped tasks while ensuring the real time constrains. Consequently the shared cache miss rate is reduced.The dynamic co-scheduling is proposed through the combination of the symbiotic technique with the megatask technique for co-scheduling the tasks based on the collected information using two schemes. The first scheme is measuring the temporal working set size of each running task at run time, while the second scheme is collecting the shared cache miss rate of each running task at run time.**

**Experiments show that the proposed dynamic co-scheduling can decrease the shared cache miss rate compared to the static one by 52%.This indicates that the dynamic co-scheduling is important to achieve high performance with shared cache memory for running high workloads like multimedia applications that require real-time response and continuous-media data types.**

***Keywords*—*Shared Cache Miss Rate; Dynamic Scheduling; Multicore; Symbiosis;***

## I. Introduction

Processor industry has moved towards the multicore technology since the delivered performance of single cores can not meet the needed requirements for running different applications like web servers, multimedia programs and databases. Multicore technology is introduced to increase the required performance and power efficiency. However, there are challenges for this technology, one of which is that the cores share a second level L2 cache. Therefore, with increasing the workload managing the shared cache space becomes essential to avoid higher miss rate which degrades the system performance. The cost of memory access has reached roughly 300 processor cycles in 2006 and has been increasing at the rate of 50% per year [1]. Decreasing the shared cache miss rate is the focus of the research in this paper.

Chip Multiprocessor (can also be named a Multicore processor) refers to a single chip that integrates two or more processors in an area that would have originally been filled with a single large uniprocessor. This solves the power consumption problem when adding more transistors to the uniprocessor and switching it at higher and higher frequencies.

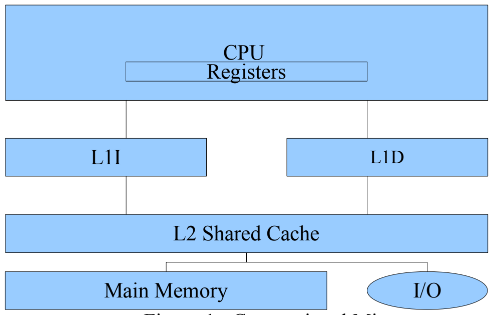

Figure 1 : Conventional Microprocessor

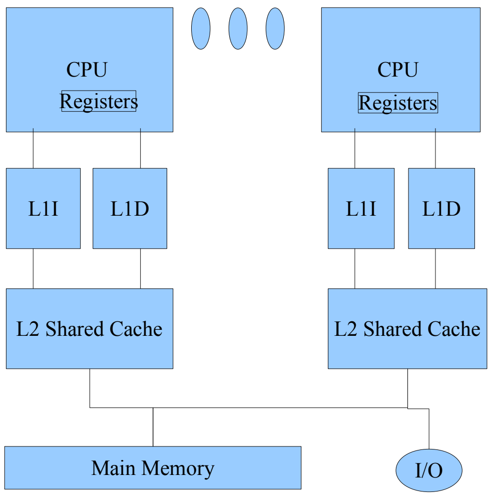

Figure 2 : Simple Chip Multiprocessor

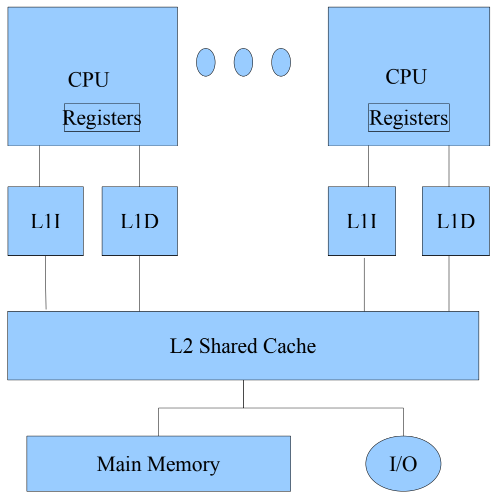

Figure 3 : Shared Cache Chip Multiprocessor

Figures 1-3 show the difference between the uniprocessor and the multicore systems. Figure 1 shows the conventional microprocessor architectures on which the other architectures are based. In figure 2 there are N cores that share only main memory and I/O, but in figure 3 there are N cores that have separate L1 cache memories and shared L2 cache. There are benefits for the shared cache architecture. For example, it can provide high bandwidth, low latency connection for the cores to communicate the shared data with each other[2]. Throughout this paper we will consider the shared cache chip multiprocessor architecture shown in figure 3.

This paper addresses the reduction of the L2 shared cache miss rate while ensuring the appropriate satisfaction of real time constraints of the tasks. For this purpose, we present two schemes for re-packing tasks in groups.Each task has a utilization and working set size (WSS)[3].The utilization indicates the core share that each task requires.The working set is defined as the collection of information referenced by the task during the task interval time ,while the working set size (WSS) is defined as the number of pages in the working-set.The temporal working set size (TWSS) is defined as the WSS every certain number of clock cycles.The first scheme is based on re-calculating the TWSS of each running task at run time. Then, it re-packs the tasks in groups. Each group has a number of tasks such that their total temporal WSS is less than or equal certain threshold. The second scheme is based on using the on-line counters for misses of each task at run time. Then it re-packs the tasks such that the miss rates are equally distributed on the groups. By this way, we can avoid the situation in which any group has a much higher miss rate than other groups. After that, each group in these two schemes is assigned a certain number of cores such that the maximum combined TWSS of all executing tasks is bounded at a value less than the capacity of the second level L2 shared cache. This will reduce concurrency within each group, eliminate the L2 shared cache thrashing, and reduce its miss rate. Finally the Pfair scheduling algorithm is used for selecting some tasks within each group for running.

The rest of the paper is organized as follows. In section II, we review the related work. In section III, we give details of the grouping and scheduling techniques that the proposed methods are based on. In section IV, we present the simulation methodology. In section V, we show the simulation results. Finally, we present the conclusions in section VI.

## II. Literature Review

Different scheduling algorithms for multicore systems have been introduced in previous researches. Fedorova *et al.* introduced using the instruction mix as a heuristic for the scheduling decisions [4]. Then, in [5] they showed that the miss rate for the second level (L2) shared cache can have the greatest negative impact on processor performance, consequently they introduced the balance-set principle for grouping all the runnable threads, such that the combined working set of each group fits in the cache. After that they introduced in [1],[6] the non-work-conserving , the target-miss-rate, and the cache fair algorithms for reducing miss rate of the L2 shared cache. These algorithms are based on using analytical performance models and online performance monitoring. Although they showed different techniques for resolving the L2 shared cache miss rate, they did not consider real time scheduling.

Real time scheduling was introduced by Anderson *et al.* [7] on which we build our proposed method. They introduced the concept of a megatask that simply represents a set of tasks to be treated as a single scheduling entity. They proposed a scheme for incorporating the megatask concept into a Pfair scheduled system. In [8] they proposed heuristics and other methods like the spread-cognizant method [9] to support both encouragement and discouragement of the co-scheduling of groups of tasks simultaneously while ensuring the satisfaction of real-time constraints. On the other hand, Anderson’s work was achieved under static co-scheduling, while we consider dynamic co-scheduling.

There are other papers that introduced scheduling algorithms that aim at increasing the system throughput of the multicore platform without considering the L2 shared cache miss rate or considering the real time scheduling. For example,Zhang *et al.* [10] proposed a hot-page coloring approach for the L2 shared cache partitioning. Cong *et al.*[11] proposed algorithms for reconfigurable resource allocation and job scheduling for achieving high performance. Azimi *et al.* [12] proposed a cache partition mechanism for partitioning the L2 shared cache among the applications based on guiding the allocation of its physical pages. Yang et al. [13] proposed a cache-aware scheduling policy which improves cache performance by considering data reuse, memory footprint of co-scheduled tasks, and coherency misses. Wang et al. [14] presented a hybrid approach for partitioning the multicores into clusters that share the L2 cache, then the tasks which access a common region of memory are statically assigned to the same cluster.

## III. Grouping & Scheduling Tasks Among Multicore Platform

The main two steps of our proposed approach is grouping and scheduling. For grouping, we are interested in combining between the two task-grouping techniques which are the symbiotic techniques and the megatasking. For scheduling, we consider applying the Pfair scheduling algorithm for ensuring the satisfaction of real time constraints for tasks.

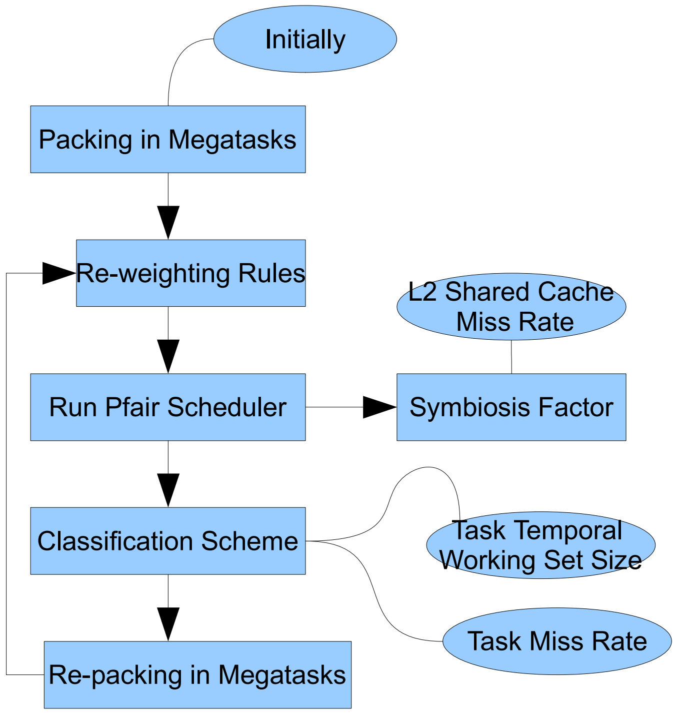

Figure 4 : The Proposed Dynamic Co-scheduling Technique

Figure 4 shows the main sequence for the proposed dynamic co-scheduling technique. It is an iterative method based on packing the tasks in groups, then running the scheduler for certain number of clock cycles. Then, getting some statistical information about the running tasks, re-grouping of tasks based on the obtained statistical information is made .These steps will be explained more in the following subsections.

## *A) Grouping*

***Megatasks***

At the initialization phase,the tasks are chosen randomly and grouped into megatasks[7]. Each megatask has a utilization that is equal to the total utilizations of its tasks. This utilization can be also termed the cumulative weight that is used to allocate one or more processors time in discrete quanta. Let Ɣ be a megatask. Its cumulative weight can then be expressed as:

$$W_{sum}=\sum_{T\in\gamma} wt(T) \quad (1)$$

where T is the component task within the Ɣ, and wt(T) is the utilization(i.e. weight) of each task.

***Packing Strategy***

As mentioned before, the tasks are grouped into megatasks such that one megatask is created at a time. The megatask is closed when the ratio of the total TWSS of the packed tasks to the size of the L2 shared cache is equal to or greater than certain integer, then another new megatask is created and so on until all the tasks are grouped.

***Re-weighting Rules***

After creating the megatasks,each megatask cumulative weight $W_{Sum}$ should be inflated.This inflation is referred as the megatask scheduling weight $W_{Sch}$ .It is computed for each megatask Ɣ using the re-weighting rules[7] in order to guarantee that its grouped-tasks meet their deadlines.This is done by assigning each megatask a number of cores indicated by the calculated $W_{Sch}$ as shown:

$$W_{Sch}=W_{Sum}+\Delta f \quad (2)$$

Where $\Delta f$ is the inflation value and can be calculated in [7].

***Symbiosis Factor***

After assigning each megatask with a certain number of cores based on its $W_{Sch}$ ,then the Pfair scheduler starts running and at run time ,the symbiosis factor is calculated. Symbiosis is a co-scheduling technique whose concept is derived from the meditation of the nature in which close and often long-term interaction between two or more different biological species is established so that they can rely on and benefit from each other.

Similarly, symbiosis is applied on the scheduling of tasks, hence symbiosis is a factor that indicates the performance of tasks that are co-scheduled and compete in hardware resources every cycle [15]. This factor may be based on system performance, system utilization, energy delay product, cores energy, cores power, average normalized turnaround time (time between submitting a job to the system and its completion) ,cache sensitivity and cache intensity, or average shared cache miss rate[15-18]. Throughout this paper, we will consider the symbiosis factor as the miss rate for the second level L2 shared cache.

It is found that computing the symbiosis factor is based on two main techniques which are sampling and probabilistic modeling.

**Sampling** This technique is known as SOS (Sample, Optimize, Symbiosis) [15, 17, 19]. It is based on producing a profile for all the possible combinations for scheduling the tasks, then taking the schedule decision based on the highest symbiosis factor.

**Probabilistic Modeling** The main drawback from the sampling technique is the large overheads resulted from profiling the different combinations of tasks to have the information necessary for scheduling. Hence the probabilistic job symbiosis modeling [18] is used to eliminate this drawback through predicting the symbiosis factor for the co-scheduled tasks without the need for the “Sampling Phase”. This technique is designed based on the following main steps:

1. The cycle stack [18,20] is calculated for each task.It is consists of three components:

Base cycle count: number of times the processor dispatches instructions for the task.

Miss event cycle count: number of times the processor consumes cycles handling miss events.

Waiting cycle count: number of times the processor dispatches instructions for another task and therefore can not make progress for the given task.

2. The probabilities for base cycle count,miss event cycle count and waiting cycle count for each task are calculated.This is done through normalizing each cycle count (i.e. Base ,miss ,or waiting cycles count) to their overall sum (i.e. Base cycle count + Miss cycle count +Waiting cycle count ).

As a consequence of the advantages of using probabilistic symbiotic modeling rather than using sampling, we use its concept to calculate the miss cycle count for each task without including the base and waiting cycles counts. Then we use this miss cycle count in calculating the task miss rate as shown:

$$\text{Task Miss Rate} = \frac{\text{Task Miss Counter}}{\text{L2 Miss Counter} + \text{L2 Hit Counter}} \quad (3)$$

where “Task Miss Counter” is the total misses in the L2 shared cache for a task,”L2 Miss Counter” is the total misses for all the running tasks ,and ”L2 Hit Counter” is the total hit for all the running tasks.

***Classification Scheme***

The scheduler takes decision to re-pack the system tasks into new megatasks every T clock cycles. This decision should be taken based on the classification scheme and the obtained information about the tasks. The classification scheme reflects how the threads affect each other when they are competing for shared resources. Consequently, it enables the scheduler to predict the performance effects of co-scheduling any group of threads in a shared cache system.

There are many different classification schemes like animalistic taxonomy, SDC, and pain. The most suitable classification schemes in our case of decreasing the miss rates among the L2 shared cache were proposed in [21,22]. These papers propose the classification schemes based on the collected information at run time. We use one classification scheme based on miss rate and propose another one based on temporal working set size.

Tasks can be classified based on the miss rate which plays a key role in the performance. The performance degradation is exacerbated by the tasks that have high miss rate due to memory controller contention, memory bus contention, and prefetching hardware contention. Hence the miss rate of each task can be obtained online using hardware counters, then the scheduler identifies the high miss rate applications and separates them into different groups, such that no one group will have a much higher total miss rate than any other group. Other metrics rather than miss rate can be also used, such as cache access rate and IPC, but the miss rate has been proved to give the best results. Hence, the miss rate classification scheme is a suitable scheme for our work. Besides that ,we propose another new classification scheme which is based on the TWSS of each task that is calculated every T clock cycles at run time.

***Dynamic Grouping***

Finally,the re-packing of the tasks in new megatasks should be done based on the classification scheme:

- In case of classification based on TWSS: the criteria of packing is exactly the same at the initialization phase.
- In case of classification based on miss rate (MR), the criteria of packing is based on that proposed in [21], in which the scheduling algorithm Distributed Intensity Online (DIO) takes the decision based on the miss rate classification. DIO uses performance counters at run time to get the miss rates of tasks (according to equation (3)). Hence, DIO observes the miss rates periodically not more frequently than once every one billion cycles in order to account for phase changes of tasks with low overhead resulted from the migrations. Then the scheduler assigns the tasks across the initially created megatasks such that the miss rates are distributed as evenly as possible according to the miss rate (MR) classification scheme.

Then , the $W_{Sch}$ for each megatask is computed using the re-weighting rules.

***Condition of Qualified Megatask***

The total number of cores ,that are assigned to the megatasks, should not exceed the number of the system cores.

## *B) Scheduling*

***Pfair Scheduler***

The second main phase is that at run time, the Pfair scheduling algorithm is used to serve the tasks within each megatask under assumption that every core is single-threaded (i.e. can only serve one data address request and one instruction address request). The most efficient Pfair scheduling algorithm is an algorithm called $PD^2$ .

Pfair scheduling can be used to schedule a periodic task system τ in which the tasks are assigned with the processor time in discrete time units that is represented with the time interval [t, t + 1), where t is a nonnegative integer, as slot t. The sequence of these scheduling decisions over time defines a schedule[23].

Each task T of the task system τ is assigned a rational weight wt(T) ∈ (0, 1] that denotes the processor share it requires. For a periodic task T ,

$wt(T) = \frac{T_e}{T_p}$ (4)where $T_e$ and $T_p$ are the (integral) execution cost and period of T .

**Tasks' Division** Each task T in τ is divided into an infinite sequence of quantum-length subtasks, $T_1, T_2, \cdots, T_i$ where each subtask $T_i$ has an associated release $r(T_i)$ and deadline $d(T_i)$ , defined as follows (for proof see [24]):

$$r(Ti) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \quad (5)$$

$$d(Ti) = \left\lceil \frac{i}{wt(T)} \right\rceil \quad (6)$$

**Tie-breaking Rules** The Pfair scheduler $PD^2$ has two tie-breaking rules which are used for breaking between the sub-tasks that have the same deadlines.

**First tie-break:The successor bit b(T)**

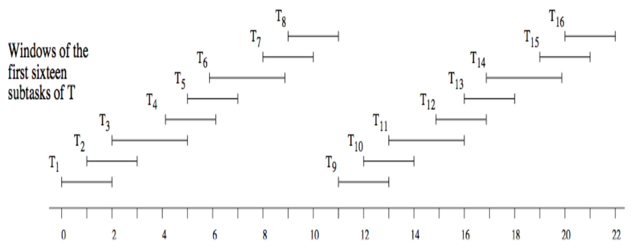

Figure 5 : Windows of the first 16 subtasks of Task T

As shown in the above figure 5,successive windows of a sub-task are either disjoint or overlap by one slot.

For example :

• The deadline of $T_1$ is 2 while the release time of $T_2$ is 1.

• The deadline of $T_2$ is 3 while the release time of $T_3$ is 2.

• The deadline of $T_3$ is 5 while the release time of $T_4$ is 4, and so on.

In other words,Formally let a sub-task $T_{i+1}$ and its predecessor $T_i$ ,so the release time of $T_{i+1}$ will be equal to either :

• deadline of $T_i$ ,or

• deadline of $T_i - 1$

From this point, they have defined a bit $b(T_i)$ that distinguishes between these two possibilities:

- $b(T_i)=1$ if release time of $T_{i+1}$ = deadline of $T_i$ -1
- $b(T_i)=0$ if release time of $T_{i+1}$ = deadline of $T_i$

**Second tie break:The group deadline D(T)**

Consider a sequence $T_i, \ldots, T_j$ of subtasks such that $b(T_k)=1$ and $|windowLength(T_{K+1})|=2$ for all i ≤ k < j.

For introducing the group deadline,as shown in figure 5,scheduling $T_i$ in its last slot forces the other subtasks in this sequence to be scheduled in their last slots.

For example, scheduling $T_3$ in slot 4 forces $T_4$ and $T_5$ to be scheduled in slots 5 and 6, respectively. So the group deadline of a subtask $T_i$ , denoted $D(T_i)$ , is the earliest time by which such a "cascade" must end.

Formally, it is the earliest time t, where $t \geq deadline(T_i)$ , such that either:

- $t = deadline(T_k)$ and $b(T_k)=0$ ,or
- $t+1 = deadline(T_k)$ and $|windowLength(T_k)|=3$ for some subtask $T_K$ .

For example, in the above Figure, $D(T_3)=d(T_6)-1=8$ and $D(T_7)=d(T_8)=11$ .

Now after defining the successor bit b(T) and the group deadline D,the next step is showing the $PD^2$ priority rules.

**The $PD^2$ Priority Definition** The $PD^2$ Priority is based on the successor bit b(T),the group deadline D ,and the deadline of each subtask d(T) as will be shown.

Under $PD^2$ , subtask $T_i$ priority is at least that of subtask $U_j$ , denoted $T_i \preccurlyeq U_j$ , if one of the following rules is satisfied:

I. $d(T_i) < d(U_j)$ .

II. $d(T_i) = d(U_j)$ and $b(T_i) > b(U_j)$ .

III. $d(T_i) = d(U_j)$ , $b(T_i) = b(U_j) = 1$ ,and $D(T_i) \geq D(U_j)$ .

## IV. Simulation Methodology

In this section we are going to show the stages of building the simulation environment. The cache simulator is based on trace-driven model and is written in C++. It models the private cache among each core, the shared cache, the main memory and the memory requests. Also It models the Modified Exclusive Shared Invalid (MESI) cache coherence protocol.

***Design Phases***

***First phase: Memory Trace Collection:*** It's a memory-access trace file based on using the "Pin" dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures [25]. The "Pin" contains a tool that can be modified for printing the address of every instruction and data that are executed within the running application. The running application is represented by SPECjvm2008 benchmarks [26] that contains 38 workloads intended to represent a diverse set of common types of computation for real-world applications including text/character processing, numerical computations, and bitwise computation. Consequently we run each workload alone with the pin tool to capture all its memory accesses, then these accesses are dumped into two trace files, one for the instructions addresses and the other for the data addresses. These trace files contain entries, where each entry has a cache type (data or instruction), an address and an access type (read or write).

***Second phase: Build The Proposed Scheduler:***This is the implementation of the Pfair for scheduling the workloads at run time and megatask grouping of the tasks. It includes four configurations that will be run for every test case in section V.

*First Configuration: Pfair without grouping:*

- There is no grouping for tasks.
- Each task is stored in its own queue.
- Each task is assigned one core at the initialization phase.

*Second Configuration: Static Megatask based on Working Set Size (WSS):*

- There is only static grouping for tasks such that each task is packed in a megatask at the initialization phase.
- The criteria for closing the megatask and creating a new one is that the ratio of the total TWSS of the packed tasks to the size of the L2 shared cache is equal to or greater than certain threshold.
- Each megatask is represented by a queue.
- Each megatask has its assigned number of cores based on its re-weighting rules.

*Third Configuration: Dynamic Megatask based on Working Set Size (WSS):*

- The initial steps are exactly like the second configuration.
- Every certain number of clock cycles or certain number of instructions (e.g. once every ten million cycles to avoid overheads due to re-scheduling), all the megatasks are re-created based on the TWSS of each task.

*Fourth Configuration: Dynamic Megatask based on Miss Rate:*

- The same as the third configuration but the only difference is in the last step as every certain number of clock cycles or certain number of instructions (e.g. once every ten million cycles),all the megatasks are re-created based on the shared cache miss rate (MR) of each task where tasks are distributed on the megatasks such that the total miss rate (MR) of all the tasks is equalized across all the megatasks.

***Third Phase: Cache Simulator:*** writing a cache simulator that models the architecture in figure 3 in which each core has a private cache and there is a shared cache among the cores. Besides that it is responsible for implementing the cache coherence protocol known as “Modified Exclusive Shared Invalid” (MESI).

*Operational Scenario*

In the proposed simulator, we assume single threaded core, so each core has a separate application. The scheduler serves these cores in a round robin manner. When there is a memory request, the cache simulator checks the cache type and operation type, then it sends it to the private cache L1 Data or L1 Instruction. If there is a hit, then it replies with data after the private cache latency cycles, otherwise it sends the request to the L2 shared cache, then if there is a hit, then it replies with data after the L2 latency cycles, otherwise it sends the request to the main memory, so it returns data after the main memory latency cycles.

***Fourth Phase: Test Cases:*** These are the test cases that are represented by the mixes of different scenarios of real execution. For example we can consider the following types of mixes:

- Total WSS for the workloads that is lower than the L2 shared cache size.
- Total WSS for the workloads that is greater than the L2 shared cache size.
- Total WSS for the workloads that is equal to the L2 shared cache size.

## V. Simulation Results

We compared the four configurations mentioned before: Pfair without grouping, Static megatask based on WSS at initialization phase, the newly proposed Dynamic Megatask based on WSS, and the newly proposed Dynamic megatask based on MR.

Table I shows the used configuration values for the main memory and the first level L1 private cache.Each workload in the SPECjvm2008 has a WSS of 2MB.These parameters are used in all the scenarios.

TABLE I. L1 Cache and Main Memory Parameters

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| Main Memory Latency | 200 |
| L1 Data Cache Size | 32KB |
| L1 Data Line Size | 64 bytes |
| L1 Data Associativity | 4 |
| L1 Data Latency | 2 |
| L1 Instruction Cache Size | 32KB |
| L1 Instruction Line Size | 64 bytes |
| L1 Instruction Associativity | 2 |
| L1 Instruction Latency | 1 |
| ***Task Parameters*** | |
| Working Set Size (WSS) | 2 MB |

The simulation results show the resulted Shared Cache L2 Miss Rate for each configuration in which total miss rate for the shared cache that is calculated as

$$\frac{\textit{shared cache L2 Total Misses}}{\textit{shared cache L2 Total Misses} + \textit{shared cache L2 Total Hits}} \quad (15)$$

## A) Scenario 1

Table II shows that there are 8 workloads of total WSS 16 MB and the number of cores is 16 with L2 shared cache of size 1MB.

TABLE II. PARAMETERS

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 8 workloads of total WSS 16 MB |
| L2 Cache Size | 1MB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 11 |
| Cores | 16 |

The graph in Figure 6 shows that the dynamic megatask outperforms the static one and the Pfair with no grouping. Also the total miss rates in both the dynamic megatask based on running tasks MR and based on WSS are near to each other.As this scenario represents the workloads of total WSS during a certain number of clock cycles that is equal or slightly greater than the shared cache L2 size.This leads to that the shared cache L2 miss rates of the four configurations are near to each other.The total miss rate tends to decrease with time as the workloads tend to finish and reach its end.

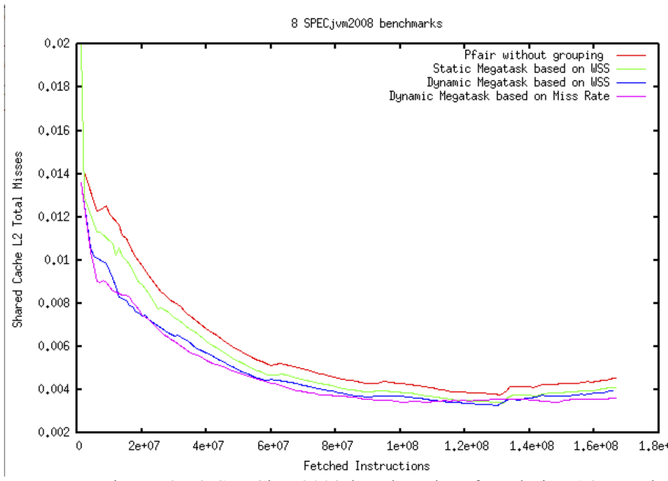

Figure 6 : 8 SPECjvm2008 benchmarks of total size 16 MB that share L2 cache of size 1 MB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

## B) Scenario 2

Table III shows that there are 4 workloads of total WSS 8 MB and the number of cores is 4 with L2 shared cache of size 8MB.

TABLE III. PARAMETERS

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 4 workloads of total WSS 8 MB |
| L2 Cache Size | 8MB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 7 |
| Cores | 4 |

Figure 7 shows that when the total WSS for the running workloads during a certain number of clock cycles fits the shared cache L2, the miss rate for the shared cache L2 becomes approximately the same for the static megatask and pfair without grouping while the dynamic megatask based on tasks WSS and based on tasks MR is slightly better than static megatask and pfair without grouping and tends to be the same when the system tasks tend to finish.

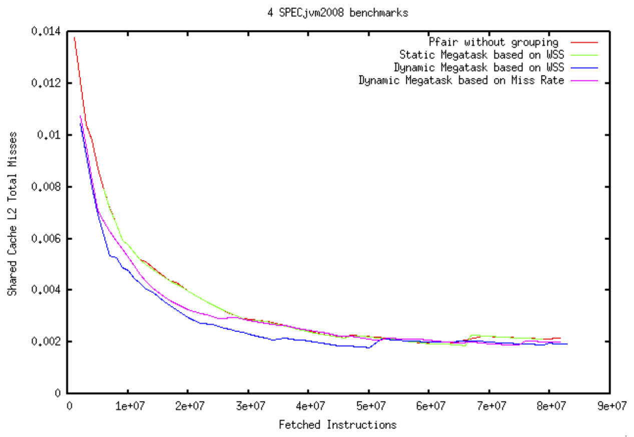

Figure 7 : 4 SPECjvm2008 benchmarks of total size 8 MB that share L2 cache of size 8 MB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

## C) Scenario 3

Table IV shows that there are 5 workloads of total WSS 10 MB and the number of cores is 4 with L2 shared cache of size 20MB.

TABLE IV. PARAMETERS

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 5 workloads of total WSS 10 MB |
| L2 Cache Size | 20MB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 20 |
| Cores | 4 |

Figure 8 shows that when the shared cache L2 is large enough, the shared cache L2 miss rate coincides in all the configurations.This case is the ideal case that rarely occurs.

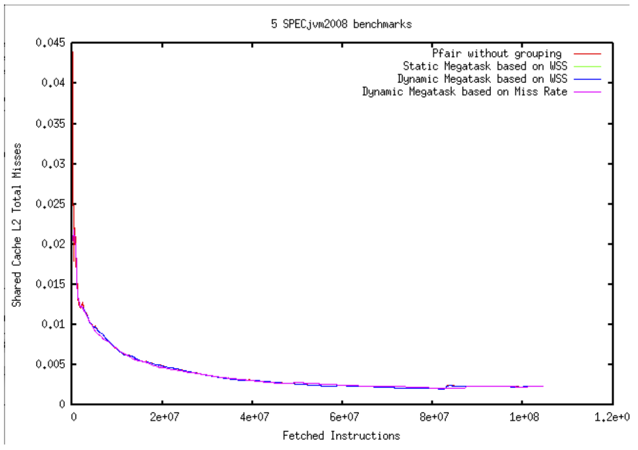

Figure 8 : 5 SPECjvm2008 benchmarks of total size 10 MB that share L2 cache of size 20 MB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

### D) Scenario 4

Table V shows that there are 5 workloads of total WSS 10 MB and the number of cores is 4 with L2 shared cache of size 1MB.

TABLE V. Parameters

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 5 workloads of total WSS 10 MB |
| L2 Cache Size | 1MB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 11 |
| Cores | 4 |

The graph in figure 9 shows a slight difference in the total miss rates as the shared cache L2 still fits the total TWSS for the running workloads.But the dynamic megatask based on the WSS and MR is still the winner.

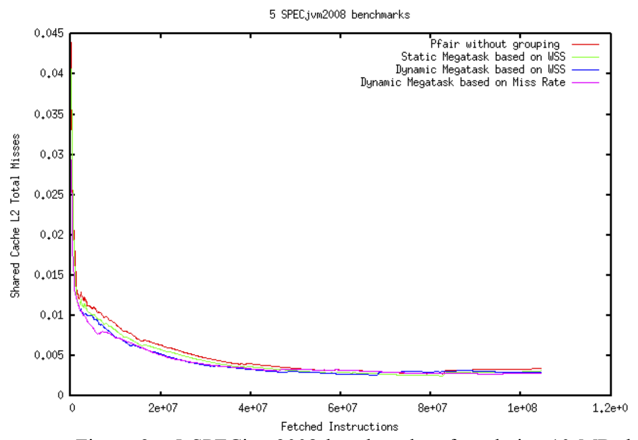

Figure 9 : 5 SPECjvm2008 benchmarks of total size 10 MB that share L2 cache of size 1 MB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

### E) Scenario 5

Table VI shows that there are 16 workloads of total WSS 32 MB and the number of cores is 4 with L2 shared cache of size 1MB.

TABLE VI. Parameters

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 16 workloads of total WSS 32 MB |
| L2 Cache Size | 1MB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 11 |
| Cores | 4 |

In this scenario ,the workloads are increased while the shared cache L2 size is kept the same.This represents the case in which the total TWSS of the running workloads is always greater than shared cache L2,thus the graph in figure 10 shows that the dynamic megatask in general has a dramatic change in the shared cache L2 miss rate rather than that in the static megatask and Pfair with no grouping.The dynamic megatask based on MR slightly outperforms that is based on WSS.Hence the proposed technique is appropriate for the high processing workloads like graphics and audio applications.

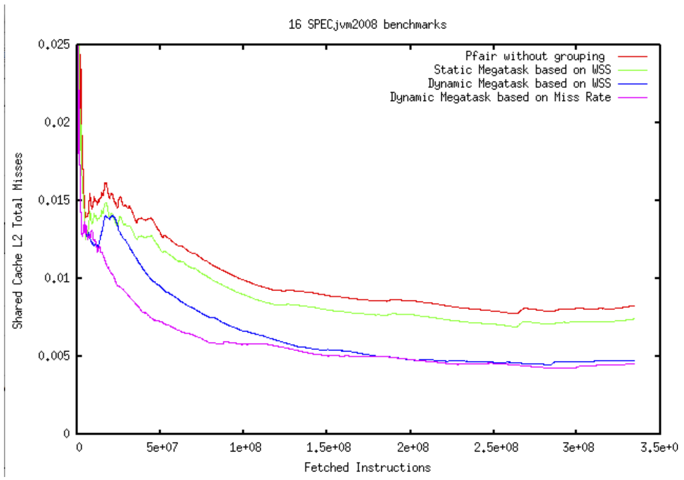

Figure 10 : 16 SPECjvm2008 benchmarks of total size 32 MB that share L2 cache of size 1 MB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

### F) Scenario 6

Table VII shows that there are 12 workloads of total WSS 24 MB and the number of cores is 48 with L2 shared cache of size 512MB.

TABLE VII. Parameters

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 12 workloads of total WSS 24 MB |
| L2 Cache Size | 512KB |

| | |
|---|---|
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 11 |
| Cores | 48 |

In this scenario the shared cache L2 size is very small compared to the total WSS of the running workloads, thus the graph in figure 11 shows that the static megatask outperforms the pfair with no grouping in decreasing the shared cache L2 miss rate ,but the dynamic megatask is still the better than that the static one.This indicates that the static megatask succeeds in reducing the concurrency within the running workloads as in the static megatask ,while the dynamic one succeeds in monitoring the symbiosis factor every certain number of clock cycles ,then re-scheduling based on the two classifier schemes MR and WSS.

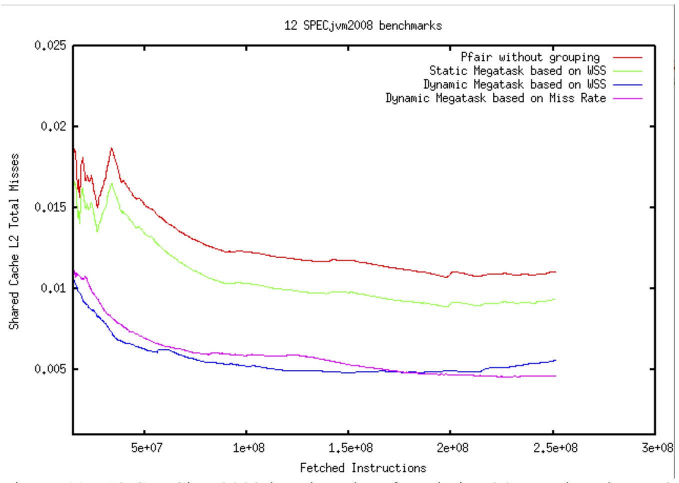

Figure 11 : 12 SPECjvm2008 benchmarks of total size 24 MB that share L2 cache of size 512 KB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

## *G) Scenario 7*

Table VIII shows that there are 10 workloads of total WSS 20 MB and the number of cores is 40 with L2 shared cache of size 512MB.

TABLE VIII. PARAMETERS

| Parameters | Values |
|---|---|
| ***Simulated Hardware Parameters*** | |
| SPECjvm2008 benchmarks | 10 workloads of total WSS 20 MB |
| L2 Cache Size | 512KB |
| L2 Cache Line Size | 64 bytes |
| L2 Associativity | 16 |
| L2 Cache Latency | 11 |
| Cores | 40 |

This scenario is the same like the previous one except that the total WSS of the workloads is slightly decreased, thus the graph in figure 12 shows that the dynamic megatask is still the better in decreasing the shared cache L2 miss rate. In all graphs the two dynamic megatask configurations are always close to each other in decreasing the L2 shared cache.Hence it is interesting to try other classifier schemes that depends on the workloads requirements and the shared cache L2 characteristics.This can be considered in the future work.

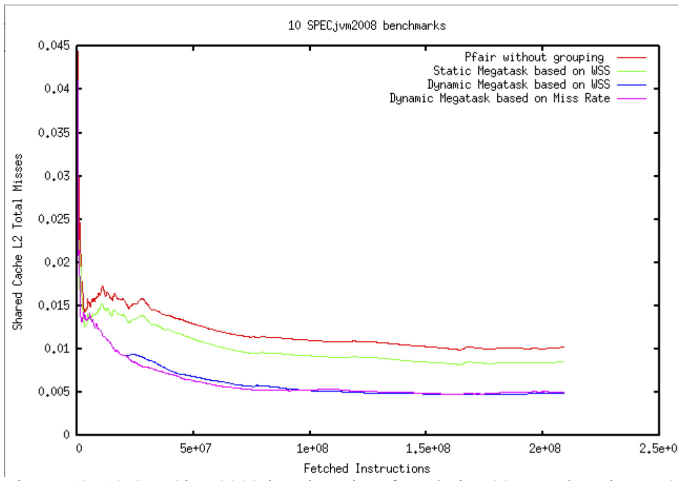

Figure 12: 10 SPECjvm2008 benchmarks of total size 20 MB that share L2 cache of size 512 KB, X-axis represents the fetched instructions and Y-axis represents the shared cache L2 Miss Rate.

Table IX shows the percentage decrease of the miss rates for the shared cache L2 in the above scenarios for the static and dynamic megatask configurations with respect to Pfair without grouping based on the following equation

$$\frac{100*(Average\ Miss\ Rate\,(Y) - Average\ Miss\ Rate\,(X))}{Average\ Miss\ Rate\,(Y)} \quad (16)$$

where Y represents the Pfair without grouping and X represents one of the other three configurations: static megatask , dynamic megatask based on TWSS, or dynamic megatask based on MR.

TABLE IX. IMPROVED SHARED CACHE L2 MISS RATE

| Scenario No. | Static Megatask | Dynamic Megatask based on WSS | Dynamic Megatask based on MR |
|---|---|---|---|
| 1 | 5.47% | 16.88% | 16.19% |
| 2 | 19.46% | 23.79% | 12.49% |
| 3 | 0.15% | 0.15% | 0.73% |
| 4 | 9.05% | 13.27% | 13.54% |
| 5 | 10.64% | 34.53% | 41.98% |
| 6 | 15.40% | 52.70% | 51.10% |
| 7 | 15.17% | 47.82% | 48.30% |

## VI. CONCLUSIONS

This paper has aimed at increasing the system throughput while ensuring the real-time constraints.It tackles the dynamic grouping technique that is based on mixing between the idea of megatask and symbiosis techniques. The symbiosis techniques is used to predict a factor for each task which is either the temporal working set size or the miss rate. The megatask is used in grouping tasks based on the classification scheme according to the symbiosis factor and calculating the re-weighting rules to ensure that the tasks meet their deadlines.The Pfair scheduling is used at run time for serving bounded number of tasks within each megatask group, hence reducing the concurrency of tasks execution within each megatask which leads to reducing the second level L2 shared cache misses.The simulation results show that the dynamic grouping technique outperforms the Pfair without grouping and the static megatask. This is especially true when the shared cache size is relatively small compared to the tasks requirements such as video coding and multimedia applications.

These results suggest some points for future work. For example, as we assume that each core has single thread, this work can be extended to multi-threaded cores.The challenge key is how to distribute threads across the cores[27].Besides that, timing analysis on multicore platform can be studied. Also our work can be extended to check the overheads for tasks migration and the impact of re-scheduling.This may suggest using another techniques for determining the tasks migration threshold as in [28].

Future work is also required to evaluate these techniques to handle multi-threaded applications.In addition to that, it is interesting to use other classification schemes that is based on cache properties like cache intensity and cache sensitivity.Research is required for proposing heuristics-based co-scheduling by machine learning.

## VII. Acknowledgment

The authors thank Islam M. Atta for his help in completing the presented work.

## References

1. Alexandra Fedorova.“Operating system scheduling for chip multithreaded processors”,A thesis to the Division Of Engineering And Applied Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy , Harvard University Cambridge, Massachusetts September, 2006.

2. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang, “The case for a single chip multi-processor,” Proc. 7th Int’l Conf. Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, 1996, pp. 2-11.

3. Peter J. Denning,”The working set model for program behavior ”,1st ACM Symposium on Operation Systems Principles,1968.

4. A. Fedorova, C. Small, D. Nussbaum and Margo Seltzer.”Chip multithreading systems need a new operating system scheduler”.In Proceedings of 11th ACM SIGOPS European Workshop, Leuven, Belgium, September 2004.

5. A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum.”Performance of multithreaded chip multiprocessors and implications for operating system design”. In Proceedings of the USENIX 2005 Annual Technical Conf., 2005.

6. A. Fedorova,M. Seltzer and M. D. Smith.”A non-work-conserving operating system scheduler for SMT processors”. In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), in conjunction with ISCA-33, June 2006.

7. James H.Anderson,John M.Calandrino, and UmaMaheswari C.Devi. “Real-Time scheduling on multicore platforms”.In Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium,pp. 179-190,April 2006.

8. James H. Anderson, John M. Calandrino, and UmaMaheswari C.Devi. “Parallel task scheduling on multicore platforms”. In Real-Time Systems Symposium, 2006, RTSS '06, 27th IEEE International, pp. 89-100, December 2006.

9. James H. Anderson, John M. Calandrino, and UmaMaheswari C.Devi. “Cache-Aware real-time scheduling on multicore platforms:heuristics and a case study”. In Proceedings of the 20th Euromicro Conference on Real-Time Systems, 2008, ECRTS '08, pp. 299-308, 2-4 July 2008.

10. R. Azimi, D. Tam, L. Soares, and M. Stumm.”Managing shared L2 Caches on multicore systems in software”.In Workshop on the Interaction between Operating Systems and Computer Architecture, Held in junction with 2007 International Symposium on Computer Architecture (ISCA-34), San Diego, CA, USA, June 2007.

11. J. Cong, K. Gururaj, and G. Han. “Synthesis of reconfigurable high-performance multicore systems”.In proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, February 2009.

12. R. Azimi, D. Tam, L. Soares, and M. Stumm. “Enhancing operating system support for multicore processors by using hardware performance monitoring”. In SIGOPS Operating Systems Review (OSR), Special Issue on the Interaction Among the OS, Compilers, and Multicore Processors, April 2009.

13. Teng-Feng Yang,Chung-Hsiang Lin,Chia-Lin Yang.”Cache-aware task scheduling on multi-core architecture”.In the proceedings of VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on April 2010.

14. Yan Wang,Lida Huang,Renfa Li,Rui Li. “A Shared cache-aware hybrid real-time scheduling on multicore platform with hierarchical cache”.In the proceeding Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on December, 2011.

15. A. Snavely,D. Tullsen,and ,and G. Voelker.Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In the Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems,University of California at San Diego.

16. Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen. “Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors”.In the Proceedings of the international parallel and distributed processing Symposium 2006,University of California, San Diego.

17. Rohit Jain. “Soft real-time scheduling on a simultaneous multithreaded processor “.Thesis Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2002.

18. Stijn Eyerman,Lieven Eeckhout. “Probabilistic job symbiosis modeling for SMT processor scheduling”.In the Proceedings of the fifteenth edition of ASPLOS 2010,ELIS Department, Ghent University, Belgium.

19. A. Snavely,N. Mitchell,L. Carter,J. Ferrante. “Exploration in symbiosis on two multithreaded architectures” , Research at IBM in the year 1999.

20. S.Eyerman,L.Eeckhout.“Per-Thread cycle accounting in SMT processors”. In the Proceedings of ASPLOS 2009,ELIS Department,Ghent University.

21. S. Zhuravlev,S. Blagodurov,A. Fedorova.”Addressing shared resource contention in multicore processors via scheduling”.In the Proceedings of ASPLOS 2010,School of Computing Science, Simon Fraser University.

22. Yuejian Xie,Gabriel H. Loh. “Dynamic classification of program memory behaviors in CMPs”.In the Proceedings of 2008 CMP-MSI.

23. J. Anderson and A. Srinivasan. “Mixed Pfair/ERfair scheduling of asynchronous periodic tasks”. Journal of Comp. and Sys. Sciences, 68, 2004.

24. J. Anderson and A. Srinivasan. “A new look at pair priorities”. Technical Report TR00-023, University of North Carolina at Chapel Hill, Sept. 2000.

25. “Pin2.13UserGuide”.

26. “SPECjvm2008 User's Guide”.

27. Stijn Eyerman,Lieven Eeckhout.”The Benefit of SMT in the Multi-Core Era:Flexibility towards Degrees of Thread-Level Parallelism”.In the Proceedings of 19th international conference ASPLOS ,March 2014,NY,USA.

28. Bagher Salami,Mohammadreza Baharani,Hamid Noori ."Proactive Task Migration with a Self-Adjusting Migration Threshold for Dynamic Thermal Management of Multi-Core Processors",The Journal of Supercomputing, 2014 – Springer.