International Journal of Electrical and Computer Engineering (IJECE)
Vol. 5, No. 4, August 2015, pp. 669~684
ISSN: 2088-8708

Performance Enhancement of Multicore Architecture

Medhat Awadalla, Hanan Konsowa
Electrical and computer engineering department, SQU, Oman
Communications, Electronics and Computers Department, Faculty of Engineering,
Helwan University, Cairo, Egypt

Article Info

Article history:

Received Mar 8, 2015
Revised Apr 21, 2015
Accepted May 10, 2015

Keyword:

Fetch policy
multi2sim
Multicore
Ordinary Least Square (OLS)
pipeline Processor

ABSTRACT

Multicore processors integrate several cores on a single chip. The fixed architecture of multicore platforms often fails to accommodate the inherent diverse requirements of different applications. The permanent need to enhance the performance of multicore architecture motivates the development of a dynamic architecture. To address this issue, this paper presents new algorithms for thread selection in fetch stage. Moreover, this paper presents three new fetch stage policies, EACH_LOOP_FETCH, INC-FETCH, and WZ-FETCH, based on Ordinary Least Square (OLS) regression statistic method. These new fetch policies differ on thread selection time which is represented by instructions' count and window size. Furthermore, the simulation multicore tool, is adapted to cope with multicore processor dynamic design by adding a dynamic feature in the policy of thread selection in fetch stage. SPLASH2, parallel scientific workloads, has been used to validate the proposed adaptation for multi2sim. Intensive simulated experiments have been conducted and the obtained results show that remarkable performance enhancements have been achieved in terms of execution time and number of instructions per second. produces less broadcast operations compared to the typical algorithm.

Corresponding Author:

Medhat Awadalla,
Departement of Electrical and Computer Engineering,
SQU, Oman
Email: medhatha@squ.edu.om

1. INTRODUCTION

Industry is embracing multicore by rapidly increasing the number of processing cores per chip. In 2005, both AMD and Intel offered dual-core x86 products; AMD shipped its first quad-core product in 2007. Meanwhile Sun shipped an 8-core, 32-threaded CMP in 2005. It is conceivable that the number of cores per chip will increase exponentially at the rate of Moore's Law, over the next decade [1]. In fact an Intel research project explores CMPs with eighty identical processor/cache cores integrated onto a single die, and Berkeley researchers suggest future CMPs could contain thousands of cores [2]. Multicore processor has been widely used to execute different applications. Multicore processor depends on multithreading architecture in many stages. The multithreading architectures, especially the Simultaneous Multithreading (SMT) architecture provides the microprocessor with a significant parallelism, which is the potential key for better performance and more throughputs. It is more promising to enhance processor utilization than superscalar or the multithreading due to its nature of a global instruction scheduling.

SMT architecture has been defined as fully shared system resources, i.e., computation resources and memory accessing resources, by several concurrently executing threads in the same core [1]. However, the resource sharing leads to uneven distribution among threads as well as performance unfairness, which might not be optimized until the well-designed scheme is carried out. Long-latency load might make the shared resources occupied by some threads inefficiently, and thus is one of the major obstacles towards higher

parallelism and better performance [3]. As a result, some instruction fetch policies are designed [4] to allocate the priorities in the fetch stage to alleviate the impact of long-latency load.

The rest of this paper is organized as follows. Section 2 explains a simulation framework and benchmark suite description. Section 3 gives the related work to the theme of the paper. Section 4 presents the developed methodology. Section 5 illustrates experiments and discussion. Section 6 concludes the paper.

Our framework consists of multicore simulation tool and a subset of benchmark programs used to evaluate the architectural enhancement using either one benchmark program executed in parallel way or multiple benchmark programs executed in sequential way [1], [6].

1.1 Multicore Simulation Description

Multi2Sim [5] is a simulation framework for heterogeneous computing including models for super-scalar, multithreaded, multicore, and graphics processors. Multi2sim uses three different models: A functional simulation engine also called simulator kernel; A detailed simulation; and An event-driven simulation.

Hereinafter we use two terms, context and threads. Figure 1 depicts the multicore components. Context will be used to denote a software entity, defined by status of virtual memory image and a logical register file. Thread will refer to a processor hardware entity which can be represented as physical register file, a set of physical memory pages, a set of pipeline queues, …etc.

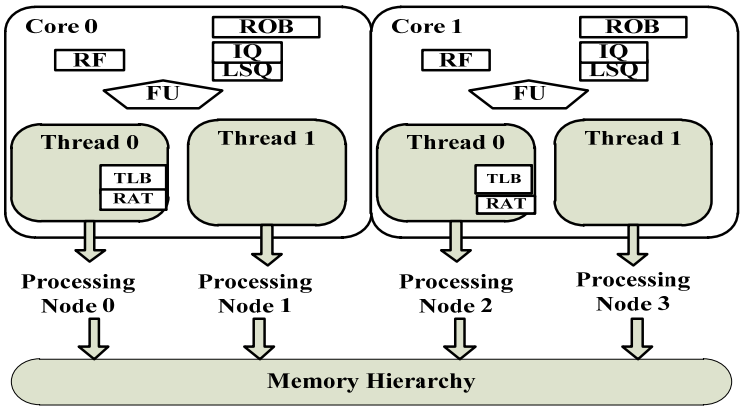

Figure 1. Multicore components

Our simulation tool supports a set of parameters that specify how stages are organized in multithreaded design. Stages can be shared among threads or private per thread except execution stage, which is shared by definition of Multithread. The pipeline model in multicore simulator is divided into five stages, fetch stage, decode/rename stage, issue stage, execution stage and commit stage as depicted in Figure 2.

The Fetch stage takes instructions from cache L1 and places it in instruction fetch queue (IFQ). The decode/rename stage takes instruction from an IFQ, decodes them, renames their registers and assigns them a slot in reorder buffer (ROB) and instruction queue (IQ).

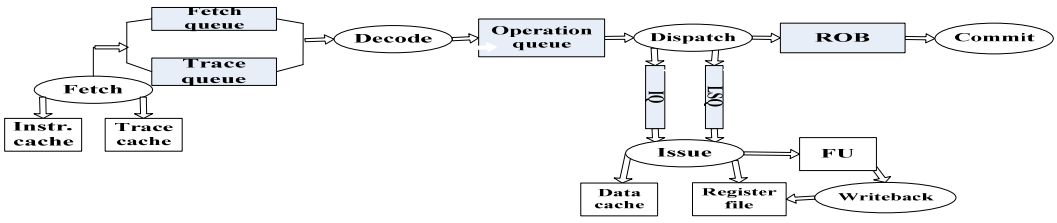

Figure 2. Processor pipelines.

Then, the issue stage consumes instructions from IQ and sends them to the corresponding functional unit. During the execution stage, the functional units operate and write their results back to register file. Finally, the commit stage retires instructions from ROB in program order.

Multicore processor switches among threads according to a thread selection policy. Processor can be classified as Coarse-grain multithreading (CGMT), Fine-grain multithreading (FGMT) and Simultaneous multithreading (SMT). FGMT processor switches thread on fixed schedule on every processor cycle, CGMT processor is characterized by thread switching induced by a long latency operation or thread quantum expiration (switch-on-event) and SMT processor is able to issue instructions from different threads in a single cycle.

The used tool has three fetch thread selection policy choices (three switch case "Shared", "TimeSlice" and "SwitchOnEvent") and the multithreading paradigms as shown in Table 1 [2].

Table 1. Classification of multithreading paradigms depending on Multi2Sim variables in a CPU Configuration file

Variable	Coarse-Grain MT	Fine-Grain MT	Simultaneous MT
FetchKind	SwitchOnEvent	TimeSlice	TimeSlice/Shared
DispatchKind	TimeSlice	TimeSlice	TimeSlice/Shared
IssueKind	TimeSlice	TimeSlice	Shared
CommitKind	TimeSlice	TimeSlice	TimeSlice/Shared

The variables to specify how pipeline stages divide their slots among threads are FetchKind, DispatchKind, IssueKind, and CommitKind. The values that these options can take are TimeSlice and Shared. The former means that a stage is devoted to a single thread in each cycle, alternating them in a round-robin fashion, while the latter means that multiple threads can be handled in a single cycle. The stage bandwidth always refers to the total number of slots devoted to threads.

The fetch stage can be additionally configured with long term thread switches, by assigning the value SwitchOnEvent for the FetchKind variable. In this case, instructions are fetched from one single thread either until a quantum expires or until the current thread issues a long-latency operation, such as a load instruction incurring a cache miss.

1.2 Benchmark Suite Description

SPLASH-2, is a workload that has 11 parallel scientific benchmarks, classified as kernels or applications [6]. All of them provide command-line arguments or configuration files to specify the input data size. Table 2 and Table 3 outline SPLASH-2 workload and the command-line arguments for some of these benchmarks. SPLASH2 benchmarks perform computations, synchronizations, communication, stressing processor cores, memory hierarchy, and interconnection networks. Thus, they are used for the evaluation of the baseline and proposed multicore architectures. Additionally, SPLASH2 benchmarks provide arguments to specify the number of contexts created at runtime, which allow the evaluation of systems with different number of cores. We focus on barnes, fmm and ocean. Each benchmark executes same number of worker threads as number of cores. Trace is collected for 500 million instructions.

Table 2. Overview of SPLASH-2 workloads

Program	Application Domain	Problem Size
Barnes	High-Performance Computing	65,536 particles
cholesky	High-Performance Computing	tk29.O
Fft	Signal Processing	4,194,304 data points
Fmm	High-Performance Computing	65,536 particles
Lu	High-Performance Computing	1024×1024 matrix, 64×64 blocks
Ocean	High-Performance Computing	514×514 grid
Radiosity	General	large room
Raytrace	Graphics	car
Volrend	Graphics	head
Water	High-Performance Computing	4096 molec

Table 3. Command-line arguments for the SPLASH-2 benchmarks

Benchmark	Arguments	Description
Fmm	$NTHREADS <input	System of N-body problem over a number of timesteps
Barnes	$NTHREADS <input	Hierarchical Barnes-Hut method for N-body problem
Ocean	-n258 –p$NTHREADS -e1e-07 -r20000 -t28800	Ocean currents simulation
LU	-p$NTHREADS –n2048 –b16	Lower/Upper matrix factorization
Radiosity	-batch –room –p$NTHREADS	Integer radix sort
Raytrace	$NTHREADS <input	Ray tracer
Waterrsp	$NTHREADS <input	Water molecule simulation

2. RELATED WORK

Much work to improve the performance of multicore architecture has been conducted including Tullsen [12], he explored a variety of SMT fetch policies that assign fetch priority to threads according to various criteria. ICOUNT was the best performing policy, in which the priority is assigned to a thread according to the number of instructions it has in decode, rename, and issue stages (issue queues) of the pipeline. Threads with the fewest number of instructions will be given the highest priority for fetch, the reason being that such threads may be making more forward process than others, to prevent one thread from clogging the issue queue, and to provide a mix of threads in the issue queue to increase parallelism. Two parameters characterize ICOUNT scheme. The first parameter dictates the maximum number of threads to fetch from each cycle, while the second denotes the maximum number of instructions per thread to fetch. More recent policies, implemented on top of ICOUNT, focus in this problem and add more control over issue queues as well as the physical registers. In [13], a load hit/miss predictor is used in a super-scalar processor to guide dispatching of instructions that the scheduler makes. This allows the scheduler to dispatch dependent instructions at the time they require data. The authors propose several hit/miss predictors that are adaptations of well-known branch miss predictors. The authors suggest adding a load miss predictor in SMT processor in order to detect L2 misses. This predictor would guide the instruction fetch, switching between threads when any of them is predicted to miss in L2. Data Gating (DG) [14] and D-CacheWarn (DWarn) [13] are other fetch policies that are based on L1 Data Cache (D-Cache) miss, Assume L1 DCach misses clearly indicate future L2 cache miss. Consequently, DG prevents the thread with unsolved L1 D-Cache miss rate from fetching instructions, such that it is less likely to introduce long latency load to the system. Similarly DWarn takes actions when L1 D-Cache miss happens too. It reduces the priority of those threads with unsolved L1 D-Cache miss without gating them completely. By adjusting priority, it is expected to achieve the balance between minimal inefficient occupancy and optimized fetch width utilization. Lichen Weng and Chen Liu [4], [5] advocated that L1 and L2 cache misses are closely associated during execution, but the relationship is not as simple as that L1 D-Cache misses leads to L2 cache miss. In fact, the relationship between L1 and L2 cache misses can hardly be described by a simple model perfectly. They used a statistical model, Ordinary Least Square (OLS) regression, to represent the relationship between L1 and L2 cache misses.

Another vital factor increases processor throughput in Simultaneous Multithreading (SMT) is the resource management. Here, we briefly present previous works on dynamic resource allocation in multiprocessor, multithreaded and multicore platforms. Although several proposals that addressed the management of a single micro-architectural resource exist in the literature, proposals to manage multiple interacting resources on multicore chips at runtime are much scarce. Static resource partitioning techniques have been suggested, but are not as effective as dynamically controlling the resource usage of each thread since program phases are not fixed all the time. Static resource partitioning [15], [16] evenly splits critical resources among all threads, thus preventing resource monopolization by a single thread. However, this method can cause resources to remain idle when one thread has no need for them, even if other threads could benefit from additional resources. Martınez [17] introduced a machine learning approach to multicore resource management. It produces self-optimizing on-chip hardware agents capable of learning, planning, and continuously adapting to change the workload demands. These results are more efficient and flexible to manage the critical hardware resources at runtime. A history-aware, resource based dynamic (or simply HARD) scheduler for heterogeneous CMPs has been developed [18]. HARD relies on recording application resource utilization and throughput to adaptively change cores for applications during runtime. This technique is used to achieve both performance and power improvements. HARD is a dynamic scheduler for heterogeneous multicore systems. It uses past thread assignments to find the best matching core for every core, saves power by downgrading applications with low resource utilization to weaker cores and improves

performance by upgrading demanding application to stronger cores. Adaptive Resource Partitioning Algorithm (ARPA) [19] that dynamically assigns resources to each thread according to thread behavior changes is proposed. It analyzes the resource usage efficiency of each thread in a time period and assigns more resources to threads which can use them in a more efficient way. The purpose of ARPA is to improve the efficiency of resource utilization, thereby improving overall instruction throughput.

3. THE PROPOSED METHOD

In the following subsections, the contribution of this paper will be presented.

6.1. Updating Multicore Simulator to Enhance Cache Prediction Accuracy

The used simulation multicore tool, multi2sim is adapted to cope with dynamic design of the multicore architecture by adding a new feature for thread selection in fetch stage and to improve the prediction accuracy. The used tool has three thread selection policy choices (three switch case “Shared”, “TimeSlice” and “SwitchOnEvent”). In practice, the fetch policy indirectly controls the usage of all processor resources. So, it can be leveraged to avoid the starvation of the concurrent threads, e.g. monopolization of instruction queues by a thread due to long-latency load misses in the last cache levels. Branch mispredictions and memory-level parallelism can also be taken into consideration by the fetch policy. In this paper fetch stage as a one of multicore resource is selected to implement a dynamic design for multicore.

Since TimeSlice fetch kind is the default choice in Multi2Sim, it should be updated to implement OLS regression feature [11] and be able to forecast L2 data cache misses from historical L1 data cache misses and actual L2 data cache misses for previous runs. To implement the above mentioned update of fetch stage, for each thread, a new array with two dimensions to store L1 data cache misses and the corresponding actual L2 data cache misses as shown in Figure 3 is used. This array represents the samples window size parameter. The storing operation of samples is accomplished at discrete time. This time is measured by number of instructions. An instruction counter is used to determine the time for storing the samples. The function to calculate the future L2 data cache miss is exist at thread level and called regression engine [22].

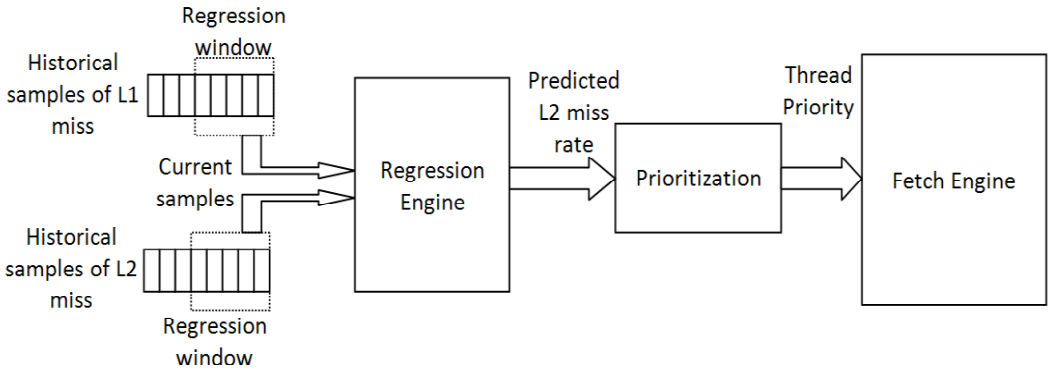

Figure 3. Fetching according to predicted L2 D-Cache Misses

In this study we change instruction count parameter which represents the time of storing the samples and address its effect on prediction accuracy of L2 data cache. The performance metric, predication accuracy, is used to measure L2 data cache miss compared by actual L2 data cache miss.

6.2. The Proposed Dynamic Fetch Policies Based on Data Cache Misses

In this section, three different fetch policies (EACH_LOOP_Fetch, INC_Fetch and WS_Fetch) are introduced. These policies depend on the data cache misses values and Ordinary Least Square (OLS) regression statistic method. At real time, the relation between L1 and L2 cache misses are updated dynamically by calculating OLS regression coefficients. The differences among these three new policies rely on the calculation time of OLS regression coefficients. The time for calculating "OLS regression coefficients" can be one of the following:

1. EACH_LOOP_Fetch: At each loop if the system can access the cache of level one.

2. INC_Fetch: At each loop if the system can access the cache and instruction counter has certain value which is defined as sampling period.this means after It is measured by Misses Per Kilo Instructions (MPKI).
3. WS_Fetch: At each loop if the system can access the cache, instruction counter is equal to sampling period and location of store reaches the maximum window size.

The calculation of OLS regression coefficients is developed through implementing the software function called Calc_ols_reg_coeff () in the simulation tool. The developed functions for the mentioned fetch policies are shown figure 4, figure 5 and figure 6.

Method one "EACH_LOOP_Fetch"

```
start ws_store1()          //window size stor
 {
            wsl[current_cache][loc]=miss[current_cache];
             calc_ols_reg_coeff();
             calc_future_L2();
            loc++;
            if(loc==ws)                          //window size
             {
               zero_ws();
               loc=0;
              }
}
```

Figure 4. "EACH_LOOP_Fetch" function in simulation tool

Method two "INC_Fetch"

```
start ws_store2()  //window size store
{
 calc_future_L2();
 ins_count++;
if(ins_count==500)  //Instruction count
      {
            wsl[current_cache][loc]=miss[current_cache]
            ins_count=0;
            loc++;
            calc_ols_reg_coeff();
           if(loc==ws)
                {
               zero_ws();
                loc=0;
                }
       }
}
```

Figure 5. "EACH_LOOP_Fetch" function in simulation tool

Method three "WS_Fetch"

```
start ws_store3()  //window size store
{
    calc_future_L2();
     ins_count++;
    if(ins_count==500) //Instruction count
     {
          wsl[current_cache][loc]=miss[current_cache]
          ins_count=0;
          loc++;
        if(loc==ws)  //window size
            {
             calc_ols_reg_coeff();
             zero_ws();
             loc=0;
            }
    }
}
```

Figure 6. "WS_Fetch" "EACH_LOOP_Fetch" function in simulation tool

4. DYNAMIC FETCH POLICIES BASED ON TRANSACTION METRICS WITH VARIABLE FETCH QUEUE SIZE

The simulation tool supports a set of parameters that specify how stages are organized in multithreaded design. Stages can be shared among threads or private per thread except execution stage, which is shared by definition of multithread. In this section, the shared allocated resources are used. We focus on fetch queue as a shared resource. Our problem is how to distribute the fetch queue for multicore among the threads by a dynamic way? The size of fetch queue is changed dynamically among the threads during the run time. The deviation from the baseline in allocated fetch queue resource is done according to transaction metric. This transaction metric [18] [23] is modified. It is defined as the ratio between the number of instructions in commit queue to the sum of the number of instructions in micro operation code (opcode) queue, fetch queue and order buffer.

$$\text{Transaction metric} = \frac{\text{Number of instructions in commit queue (output m/c)}}{\text{Number of instructions in (opcode +fetch +reorder buffers)(input m/c)}} \quad (1)$$

It is also used to differentiate between the threads, it can be defined as an evaluation metric. This fetch policy is named as Maximum Transaction policy, MTRANS, which increments the allocated resources for a specific thread. This thread must have maximum value of transaction metric. So, it should have more resources to speed up the execution of its instructions.

The implementation of this fetch policy is done by dividing fetch queue size equally for all threads. The calculation of transaction metric is performed periodically after some cycles called epoch for each thread. After each epoch, an additional resource is granted gradually in steps. This means that after each epoch the redistribution operation is done by deduction the fetch queue entries from each thread (processing node) and add these values to the active thread.

The developed pseudo code for MTRANS is shown in figure 7. In this algorithm, MTRANS, the active thread is the thread that has the maximum transaction value. The number of instructions migrated from one thread to another is called step. For example, If the multicore processor have two cores, each core has two threads and the fetch queue size is 64, so each thread will take 16 entries in the fetch queue. Assume the epoch is 1000 instruction and the step is 2. After 1000 instructions, the transaction metric is calculated based on equation (1). We take 2 instruction entries from fetch queue assigned for three threads and add them to the fetch queue size of the active thread (say thread 2). In this case, the size of fetch queue for thread1, thread 3 and thread 4 will be 14 while for thread 2 will be 20.

```
MTRANS
// select thread with maximum transaction regardless data misses
get_thrd_max_Transacton()
{
for(i=0;i<4;i++) // The used multicore model consists of 2
            // cores, each one has 2 threads.
   {
     if(thrd_max_trans==i)continue;
     q_sz[i]= q_sz[i]-step _q;
    if(q_sz[i]<=q_sz_min)
    q_sz[i]=q_sz_min;
    q_sz[thrd_max_trans]=
    queue_sz[thrd_max_]+step_q;
    if(q_sz[thrd_max_ trans]>=q_sz_max)
    queue_sz [thrd_max_ transaction]=  queue_sz _max;
    break;
            }
}
```

Figure 7. MTRANS function

5. DYNAMIC FETCH POLICIES BASED ON TRANSACTION METRICS AND DATA CACHE MISS WITH VARIABLE FETCH QUEUE SIZE

Two new thread selection algorithms for fetch policy have been proposed in this section. The first algorithm is Maximum Transaction No Max Misses algorithm (MTNMM) which is used to increment the allocated resource i.e. (increasing the size of fetch queue to accommodate more instructions) for specific thread. This thread must have maximum value of transaction metric and must not have the maximum misses of L1 data cache. In this case, the thread will gain more resources to speed up the execution of its instructions. However, if it does not have the maximum misses for L1 data cache, then it will have only the assigned size of fetch queue, the default assigned fetch queue size (baseline). This means that at each epoch, the transaction metric is recalculated to select the active thread and determine the size of fetch queue.

The concept of the second algorithm MICRO_COUNT is the same as MTNMM, however the selected thread must not have minimum opcode in micro opcode queue. The increasing in resource allocation will be granted to the active thread as if it has not the minimum number of opcode in micro_opcode queue. The pseudo code these two algorithms are stated in figure 8 and figure 9 while flowcharts are depicted in Figure 10 and Figure 11.

```
// MTNMM
//select thread with maximum transaction and it has not maximum data cache misses in L1.
//get_thrd_max_Trans()
get_thrd_max_Transaction()
{
 get_thrd_max_miss();
 for(i=0;i<4;i++){
 if(thrd_max_trans==i)          continue;
//exclude the thread with maximum data cache misse
         if(thrd_max_trans==thrd_max_misses)    continue;
q_sz[i]=q_sz[i]-step _q;
if(q_sz [i]<= q_sz_min)
        q_sz[i]=q_sz_min;
 q_sz [thrd_max_trans]  = q_sz[thrd_max_trans]+step_q;
 if(q_sz [thrd_max_ trans]>= q_sz _max)
 q_sz[thrd_max_trans]=q_sz _max;
 break;
}
}
```

Figure 8. MTNMM function

```
//MICRO_COUNT
//select thread that has maximum transaction and minimum opcode in //micro opcode queue.

get_thrd_min_opcode() // select the thread of minimum opcode
{
    uopq_count(core,thread); //get micro opcode count for each thread

    min_micr=0;frst=0;
    for(i=0;i<4;i++){
    if(frst==0){min_micr=micr_count[0];thrd_min_micr=0;frst=1;}
                if(micr_count[i]==0) continue;
                if(micr_count[i]<min_micr){thrd_min_micr=i;}
                 }
}

get_thrd_max_Transaction() //get_thrd_max_Trans()
{
for(i=0;i<4;i++){
        if(thrd_max_trans==i)continue;
//exclude the thread with minimum opcode in micro opcode queue.

        if(thrd_min_micr == thrd_max_trans)continue;
         q_sz[i]=q_sz[i]-step _q;
         if(q_sz [i]<= q_sz_min)
         q_sz[i]=q_sz_min;
         q_sz [thrd_max_trans] =q_sz[thrd_max_trans]+step_q;
         if(q_sz [thrd_max_ trans]>= q_sz _max)
         q_sz[thrd_max_trans]=q_sz _max;
         break;
            }
}
```

Figure 9. MICRO_COUNT function

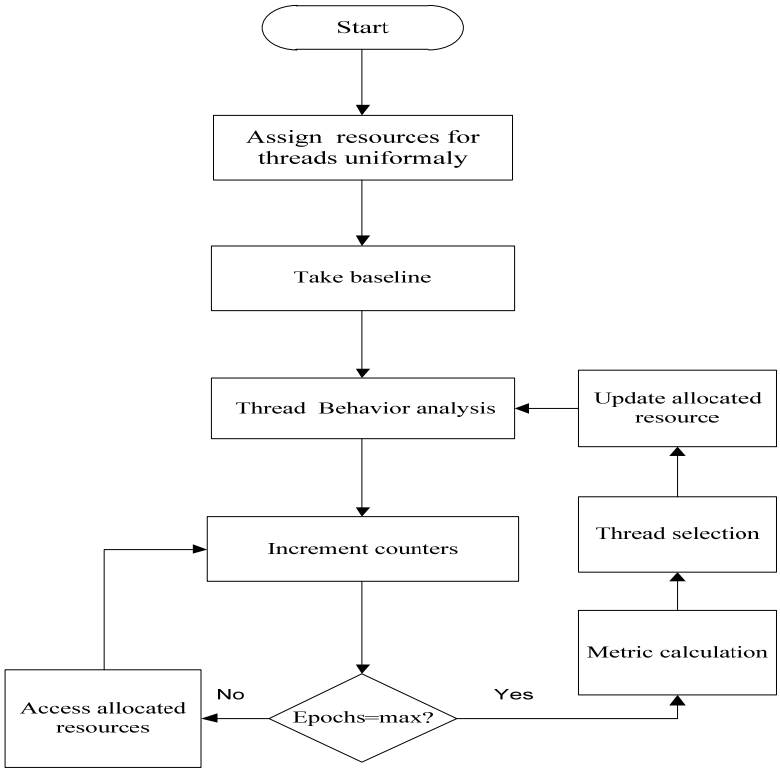

Figure 10. Dynamic resources allocation flowchart for MTRANS algorithm

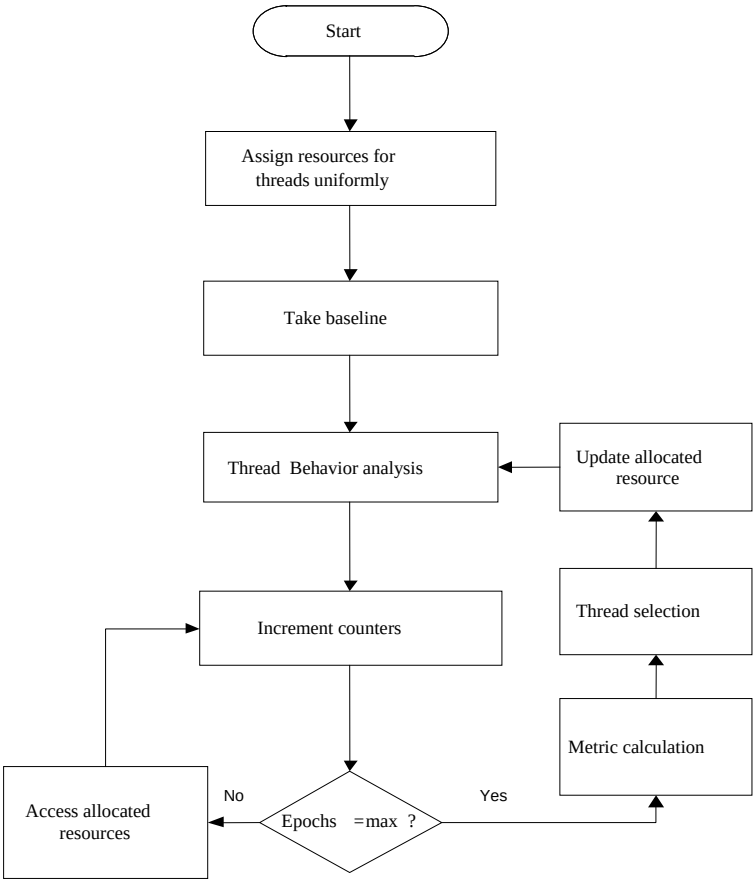

Figure 11. Dynamic resources allocation flowchart for MTNMM algorithm

6. SIMULATION RESULTS

A lot of simulated experiments have been conducted using the modified simulation tool, Multi2Sim, to check the validity of the proposed approaches.

6.1. The Modified Simulation Tool

Multi2Sim is modified to incorporate more options and implement OSL. Figure 12 and Figure 13 illustrate the effect of changing the sampling time on the relation between actual data cache misses at level one (L1) and the predicted value of data cache misses in data cache level two (L2). It is actual reading value of data cache misses in DL2 minus the predicted value of data cache misses in DL2 for each windows array size. The experiments have been repeated for different values of window array size and instruction count. We tried to find the values of these parameters at which the highest prediction accuracy by changing one parameter of them and fix the other and vice versa. As shown in Figure 13, the highest prediction accuracy is occurred at instruction count (sampling time) = 1000 and windows array size equal 7.

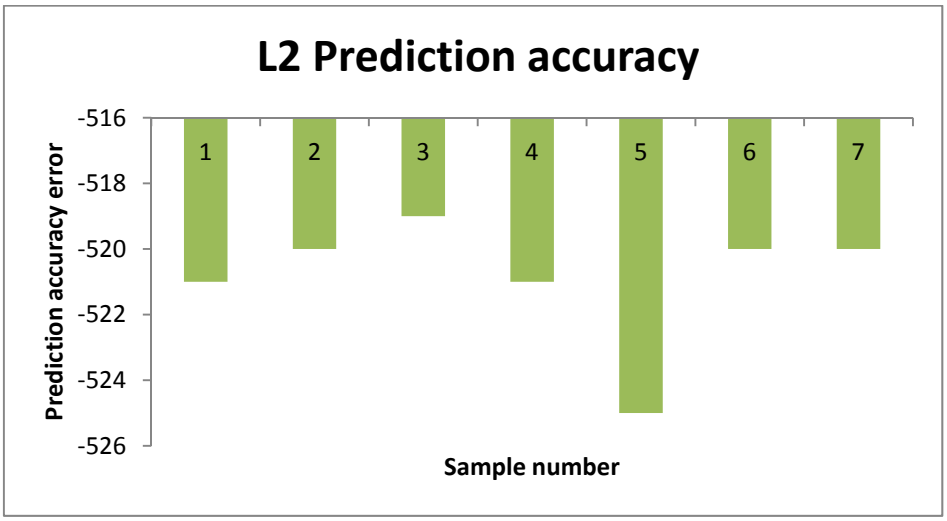

Figure 12. Prediction accuracy for Ocean program from SPLASH benchmark suite (Window array size =7 and instruction count=500)

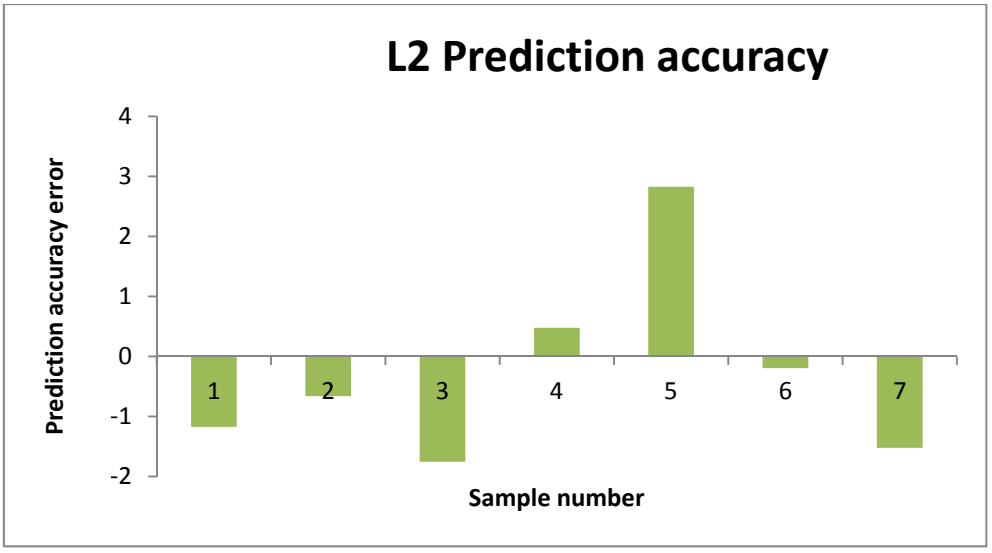

Figure 13. Prediction accuracy for Ocean program from SPLASH benchmark suite (Window array size =7 and instruction count=1000)

6.2. Dynamic Fetch Policies Based on Data Cache Miss

To investigate the dynamic fetch policies based on data cache miss, three applications (LU, Sort, and Ocean) have been run using the three new proposed fetch policies. Different performance metrics have been evaluated such as the execution time, the number of instructions executed per second and number of cycles per second. The achieved results are depicted in figure 14, figure 15 and figure 16. As obvious from the achieved results, the minimum execution time, the best IPS and highest cycles executed per second are occurred during WS_FETCH fetch policy for all benchmarks.

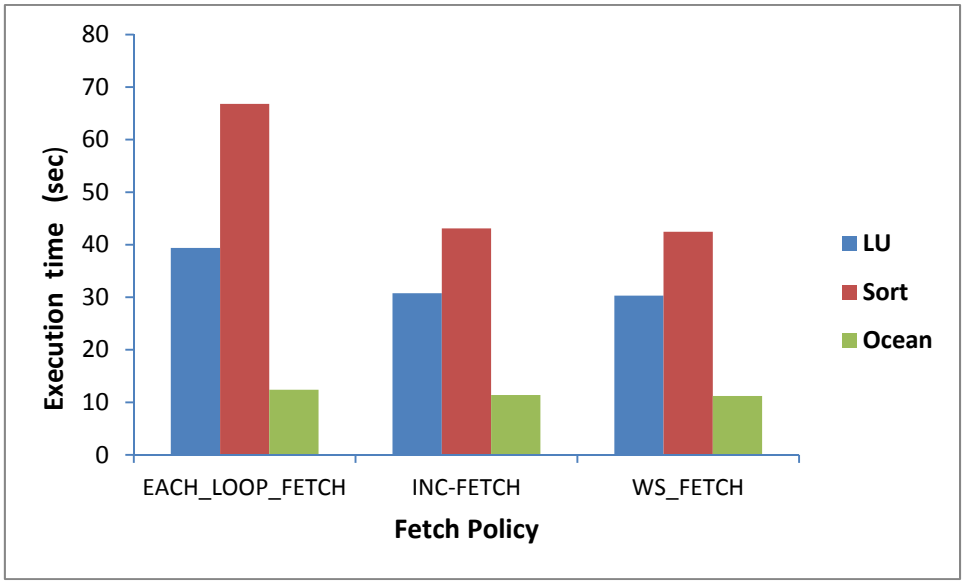

Figure 14. Execution time metric using different fetch policies for three SPLACH benchmarks

Furthermore the best results of IPS and cycles executed per second are occurred, shown in Figure 15 and Figure 16, when WS_FETCH fetch policy is used for all benchmarks when the time between the OLS coefficient is increase.

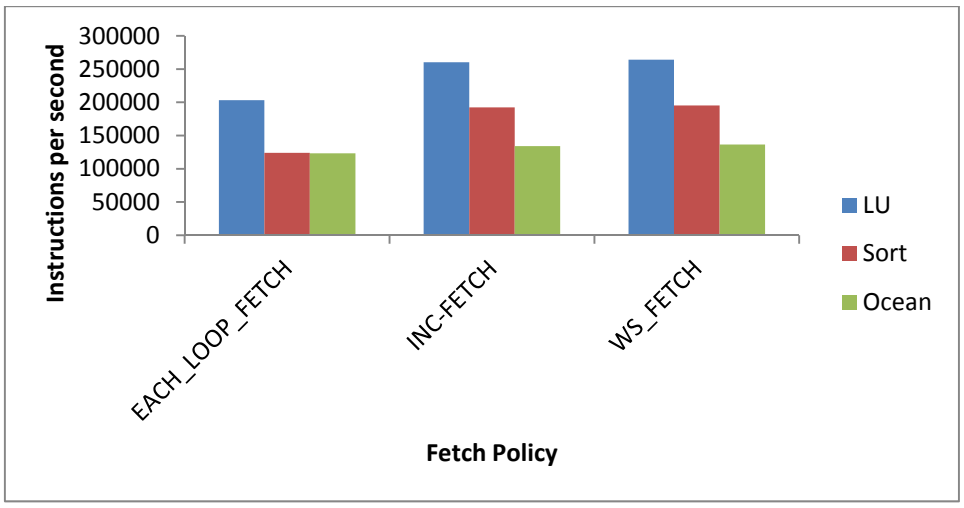

Figure 15. IPS metric using different fetch policies for three SPLASH benchmarks

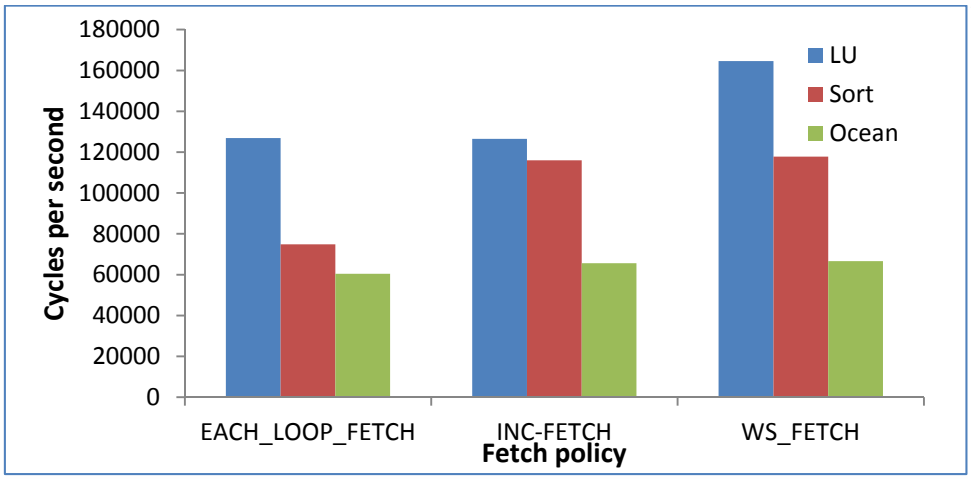

Figure 16. CPS metric using different fetch policies for three SPLASH benchmarks

6.3. Dynamic Fetch Policies Based on Transaction Metrics With Variable Fetch Queue size

In this section, the proposed algorithms have been applied and the system throughout is evaluated in terms of number of executed instructions per cycle (IPC). To accomplish that, suitable values for the following parameters should be selected:

- Epoch size that represents the period to calculate MTRANS metric.
- New fetch queue entries (step size) that migrate from a thread to another at each epoch.

The experiments have been repeated for different values of both epoch size and step size. As shown in Figure 17, the highest instructions per cycle is occurred at epoch value equal 2000 instructions and step size equal 5 in case of MTRANS algorithm.

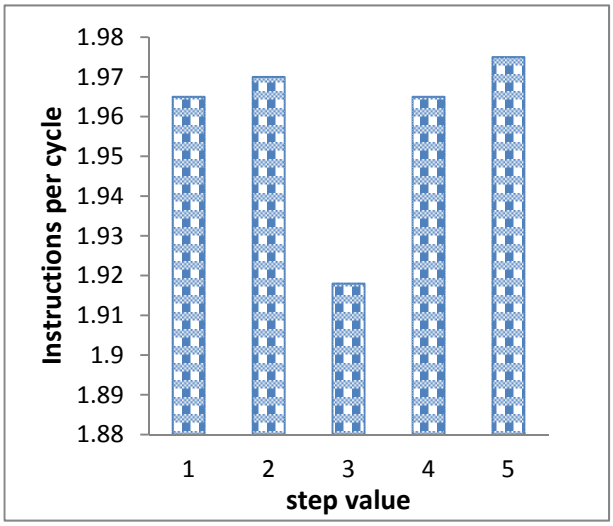

Figure 17. Average of instruction per cycles for SPLASH benchmark program at different step values (epoch =2000) for MTRANS algorithm.

Again, four benchmark applications, Sort, Ocean, Barnes and Fmm with two different fetch policies, one policy based on MTARNS and the other is static (no changes for queue fetch size). As illustrated in Figure 18, IPC for all mentioned benchmarks has higher values with MTRANS fetch policy.

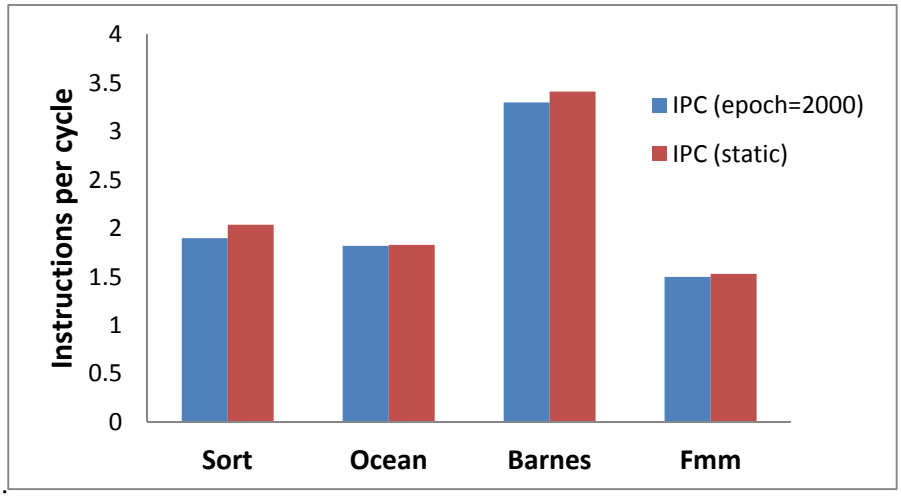

Figure 18. IPC for SPLASH benchmark programs when epoch=2000 instructions for static and MTRANS fetch policies

6.4. Dynamic Fetch Policies Based on Transaction Metrics and Data Cache Miss with Variable Fetch Queue Size

The achieved results of running four benchmark applications with two new fetch policies based on MTARNS and MTNMM are shown in Figure 19. It is not necessary that the best algorithm for certain application to be the best for all applications. It means that the MTNMM algorithm is an application specific algorithm. As shown in Figure 19, MTRANS fetch policy outperforms the MTNMM for all benchmarks except FFT application. This means that the maximum performance is obtained when the selected thread has maximum transaction and also it has minimum data cache misses in DL2 with respect of all other threads (MTNMM fetch policy). MATRNS fetch policy which is based on the condition of maximum transaction only, is not enough to gain the best performance.

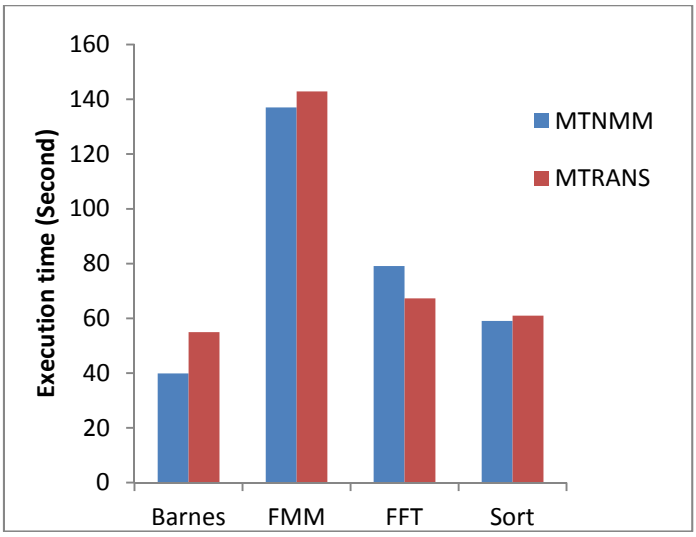

Figure 19. Execution time for SPLASH benchmark programs when epoch=2000 instructions for MTRANS and MTNMM algorithm

The results of running some mixed SPLASH benchmark applications using fetch policies based on MTRANS, MTNMM and MICRO_COUNT are shown Figure 20.

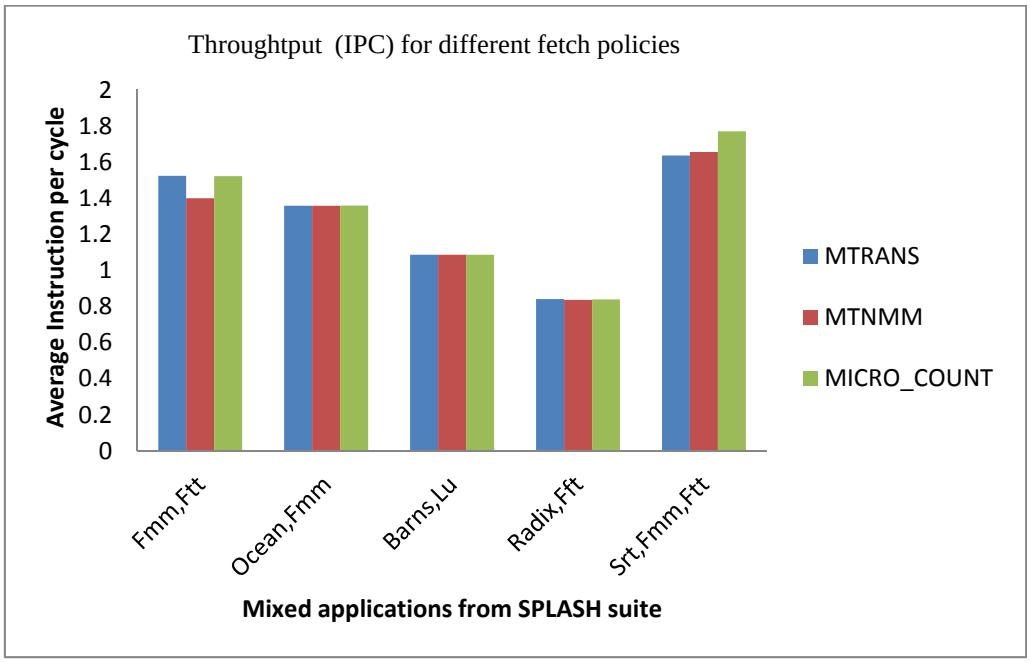

Figure 20. IPC for SPLASH benchmark programs (epoch=2000 instructions) for MTRANS, MTNMM and MICRO_COUNT algorithms

In case of mix_5, using MICRO_COUNT fetch policy is outperformed. Where the active thread must not have minimum opcode in micro opcode queue and it will have more resources.

7. CONCLUSION

In this paper, the simulation multicore tool, multi2sim, is adapted to cope with multicore dynamic design by enabling dynamic design for thread selection policy in fetch stage. Dynamic fetch queue size is based on dynamic distribution for these resources to threads. Where, the competing threads for a given resource able to give part of their resources to another threads without degrading the system performance. This paper presents a dynamically reconfigurable multicore architecture in fetch stage by developing two new algorithms for thread selection in fetch stage. Furthermore, a variable fetch queue size is suggested to avoid resource starvation to increase the system performance. Moreover, this paper presents three new fetch stage policies, EACH_LOOP_FETCH, INC-FETCH, and WZ-FETCH, based on Ordinary Least Square (OLS) regression statistic method. These new fetch policies differ on thread selection time which is represented by instructions' count and window size. Different workloads have been utilized to investigate the performance of the dynamic multicore processors in terms of execution time, IPC and number of cycles per second. It is found that there is a remarkable improvement of the overall system performance using the proposed approaches.

REFERENCES

[1] R. Ubal, J. Sahuquillo, S. Petit, and P. L_opez. *Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors.* In Proc. of the 19th Int'l Symposium on ComputerArchitecture and High Performance Computing, Oct. 2007.

[2] John P. Shen and Mikko H. Lipasti. s for Multicore Processor using Register File Protection". International Journal of Computer Application, Volume 92 - Number 7, April 2014. Published by Foundation of Computer Science, New York, USA.

[3] *Modern Processor Design: Fundamentals of Superscalar Processors*. July 2004.

[4] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: *An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures.* In Proc. of the 42nd Int'l Symposium on Microarchitecture, Dec. 2009

[5] L. Weng and Chen Liu : *Hardware-aided Monitoring of L1 and L2 D-Cache Misses in SMT*. Available at: http://web.eng.fiu.edu/cliu/Pub/posterfhpm.pdf, 2011

[6] L. Weng and Chen Liu, *Fetching According to the Evaluated L2 Cache Misses By OLS Regression in SMT Architecture*. Available at: http://web.eng.fiu.edu/cliu/Pub/ASPLOS2011.pdf, 2011
[7] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*. In Proc. of the 22nd Int'l Symposium on Computer Architecture, June 1995.
[8] D.H. Bailey. FFT's in External or Hierarchical Memory. Journal of Supercomputing, 4(1):23-35, March 1990.
[9] L. Greengard, The Rapid Evaluation of Potential Fields in Particle Systems. ACM Press, 1987.
[10] *The Multi2Sim Simulation Framework*. http://www.multi2sim.org.
[11] A. Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. Mathematics of Computation 31(138): 333-390, 1977
[12] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm, *Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor*. ISCA, 1996.
[13] F.J. Cazorla, A. Ramirez, M. Valero and E. Fernandez, *DCache warn: an I-fetch policy to increase SMT efficiency*. IPDPS, 2004.
[14] A. El-Moursy and D.H. Albonesi, *Front-end policies for improved issue efficiency in SMT processors*. HPCA, 2003.
[15] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M. Upton, Hyper-Threading Technology Architecture and Microarchi-tecture, Intel Technology J., vol. 6, no. 1, pp. 4-15, Feb. 2002.
[16] S.E. Raasch and S.K. Reinhardt, *The Impact of Resource Partitioning on SMT Processors*, Proc. 12th Int'l Conf. Parallel Architecture and Compilation Techniques, pp. 15-26, Sept. 2003.
[17] J.F. Martınez. "Dynamic Multicore Resource Management: A Macine Learning Approach", IEEE Micro, Volume 29 Issue 5, September 2009
[18] A.Z. Jooya and A. Baniasadi and M. Analou "History-Aware, Resource-Based Dynamic Scheduling for Heterogeneous Multi-core Processors"; IET Computers & Digital Techniques, (2011) 254-262.
[19] H.Wang, I. Koren and C. Krishna, "An Adaptive Resource Partitioning Algorithm in SMT Processors", Parallel and Distributed Systems, IEEE Transactions on, Volume: 22, Issue: 7 July. 2011.
[20] H. Konsowa, E.M. Saad, and M.H. Awadalla, "New Fetch Policies for Multicore Processor Simulator to Support Dynamic Design in Fetch Stage", Journal of computer science and engineering, volume 12, issue 2, Pp. 6-14, 2012.
[21] A. Cottrell, "Regression Analysis: Basic Concepts", [Online]. Available: Regression.pdf
[22] H. Konsowa, E.M. Saad, and M.H. Awadalla, "Dynamic Resources for Multicore Processor", Journal of computer science and engineering, JCSE 2013.