View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ethesis@nitr

Scheduling Techniques to avoid Contention in Multi-core Systems

Pallavi Thummala

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Scheduling Techniques to avoid Contention in Multi-core Systems

Thesis submitted in partial fulfilment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Computer Science)

by

Pallavi Thummala

(Roll No. 212CS1095)

under the supervision of

Prof. A. K. Turuk

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha, 769 008, India

June 2014

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***Scheduling Techniques to avoid Contention in Multi-core Systems*** by ***Pallavi Thummala*** is a record of an original research work carried out by her under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Computer Science in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: June 2, 2014

Dr. A. K. Turuk
Associate Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Prof. A. K. Turuk for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.

I extend my thanks to our HOD, Prof. S. K. Rath for his valuable advices and encouragement.

My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Pallavi Thummala
Roll: 212CS1095

Author's Declaration

I, **Pallavi Thummala** (Roll No. **212CS1095**) understand that plagiarism is defined as any one or the combination of the following

1. Un-credited verbatim copying of individual sentences, paragraphs or illustrations (such as graphs, diagrams, etc.) from any source, published or unpublished, including the Internet sources.
2. Un-credited improper paraphrasing of pages or paragraphs (changing a few words or phrases, or rearranging the original sentence order).
3. Credited verbatim copying of a major portion of a paper (or thesis chapter) without clear delineation of who did or wrote what.

I have made sure that all the ideas, expressions, graphs, diagrams, etc., that are not a result of my work, are properly credited. Long phrases or sentences that had to be used verbatim from published literature have been clearly identified using quotation marks.

I affirm that no portion of my work can be considered as plagiarism and I take full responsibility if such a complaint occurs. I understand fully well that the guide of the thesis may not be in a position to check for the possibility of such incidences of plagiarism in this body of work.

Place: NIT Rourkela
Date: June 2, 2014

Pallavi Thummala
Roll: 212CS1095
CSE Department
NIT Rourkela, Odisha

Abstract

One of the main problems in multi-core systems is the contention of shared resources such as cache, memory controller, pre-fetcher etc. among the cores. Due to the contention among shared resources, the processing unit's performance is degraded. Scheduling of applications in such a way that it reduces the contention among shared resources is one of the promising solutions. Scheduling is considered as an efficient and best technique as it doesn't require any extra hardware or any changes to be made to the OS or its underlying kernel. Scheduling can be implemented at user level by using system calls. In the prior works it was considered that the cache contention was the main cause of performance degradation and many hardware and software techniques were found to avoid or minimize it. But further experiments proved that the contention caused by pre-fetcher and memory controller is also having significant effect on performance degradation. Many scheduling policies and classification schemes have been designed to find out an efficient scheduling algorithm. Miss rate is considered to be simple yet efficient classification scheme to classify the threads as it not only considers contention due to cache but also the memory controller and pre-fetcher. Distributed Intensity is the first scheduling algorithm discussed which uses miss rate to classify threads and assign them to all cores in an efficient way so that miss rate is shared almost equally among the cores. Then Distributed Intensity is combined with Swap algorithm to further improve the performance by using dynamic optimization. Then by further studies it is found out that miss rate cant be efficient classification technique for memory intensive workloads. So the concepts of Contentiousness and Sensitivity are introduced to improve the efficiency of scheduling algorithm and to minimize the performance degradation due to contention.

Keywords: *Scheduling; Contention; Miss rate; Sensitivity; Threads; Multi-core systems*

Contents

Certificate i

Acknowledgment ii

Author's Declaration iii

Abstract iv

List of Figures vii

List of Tables viii

1 Introduction 2

1.1 Introduction 2

1.2 Literature Review 2

1.3 Motivation 4

1.4 Objective 4

1.5 Thesis Organization 4

2 Background 6

2.1 Factors causing contention: 6

2.2 The Perfect Scheduling Policy: 9

2.3 Thread classification schemes: 10

2.4 Basic concepts of Contention Characteristics 11

2.4.1 Contentiousness 11

2.4.2 Sensitivity 12

2.5 Tools used: 13

2.5.1 AKULA Toolset 14

2.5.2 MARSS 15

2.5.3 Oprofile 16

3 Implementation **18**

3.1 Distributed Intensity 18

3.1.1 Algorithm 19

3.1.2 Benchmarks and Workloads 19

3.1.3 Results 19

3.2 Swap 23

3.2.1 Algorithm 23

3.2.2 Benchmarks and Workloads 23

3.2.3 Results 24

3.3 Contentiousness and sensitivity 25

3.3.1 Experimental setup 25

3.3.2 Benchmarks 25

3.3.3 Results 25

4 Conclusion **28**

Bibliography **29**

List of Figures

2.1 Jiang Methodology [6] 9
2.2 Flowchart of AKULA [9] 14
2.3 MARSS Instance [10] 15

3.1 Comparision of schedulers 22
3.2 comparision of swap scheduler 24
3.3 Contentiousness vs Miss rate 25
3.4 Sensitivity vs Miss rate 26
3.5 Sensitivity vs Miss rate 26

List of Tables

3.1 Solo execution time 20

3.2 Degradation matrix 21

3.3 Miss rates 21

3.4 Solo execution time 21

3.5 Degradation matrix 21

3.6 Miss rates 21

Chapter 1

Introduction

Introduction
Literature Survey
Motivation
Objective
Thesis Organisation

Chapter 1

Introduction

1.1 Introduction

Multi-core processors have been immensely used in various fields because of their low power consumption and good performance with low space usage on the die which in turn decreased the heat dissipation when compared to multiprocessors. But as there are more than one processing unit (core) present on the same die they have to share some of the resources which in turn causes the problem of contention [1] which was not present in multiprocessors. Therefore, many measures have been taken to minimize this disadvantage and one of the efficient techniques was scheduling [2] [3].
The underlying OS scheduler doesn't have any idea about the shared resources and distributes the work among all the cores by the principle of load balancing without considering the effects of it [4]. Therefore, there is a need of scheduling algorithm that identifies the factors that cause the contention and amount of contention [5] and nature of workloads to distribute equally among all the cores reducing the contention and increasing the performance.

1.2 Literature Review

Zhuravlev et al. [6] described the problem of contention among shared resources which is addressed by providing thread scheduling as one of the efficient methods to minimize it. By extensive experiments, it is also found out that contention due to cache [7] is not the only the performance degradation factor, but

also the contention due to the memory bus, memory controller and prefecture also has a significant effect on performance. Different classification schemes and scheduling policies are studied and miss rate is found out to be a simple yet efficient classification scheme to divide the applications as intensive or not.
Blagodurav [8]found out that the problem of default OS scheduler is that it does load balancing blindly without considering the characteristics of threads by co-scheduling each thread with all other threads and it is found out that performance is improved up to 50% when the best case is considered rather than a worst case schedule. The concepts of different scheduling algorithms such as Distributed intensity and Distributed intensity online are also mentioned.
Zhuravlev [9] mentioned the main problems of developing a scheduling algorithm and addressed them as the implementation difficulty and the testing duration. They proposed AKULA toolset that eases the work of developers by providing API and by making debugging easier without any modification done to kernel or by using any system calls.
Avadh patel et al. [10] proposed a full system simulator MARSS is described which includes QEMU to provide cycle accurate simulation of many x86 core processors of same or different processing capacities. It can also emulate hardware like caches, interconnects, input output devices and chips. It has the ability to run many OS on the emulated hardware without any modifications done to them.
Blagodurav et al. [11] described the usage of hardware performance counters and instruction sampling is addressed which can be used by scheduling algorithms to take wise decision to minimize the contention problem. Clavis scheduler was developed to show the user level scheduling that can be done on Linux system by using the information provided by performance counters.
Zhuravlev et al. [12] did a on contention aware scheduling techniques and negative and positive impacts of shared resources. If resources are shared between multi threaded applications which share data among their threads then an increased in performance was observed. The OS thread level scheduler has to be changed according to the scenario for CMPs to take advantage of sharing the resources.

1.3 Motivation

Multi-core processors are advantageous and are growing rapidly even though they have the disadvantage of contention for shared resources because of their low power consumption and heat dissipation [5]. So there is a need of minimizing the contention problem and still improving the performance of multi-core processors. Scheduling of threads is one of the efficient solutions to minimize the contention. So there is a need of developing effective scheduling techniques to be used by underlying OS scheduler to minimize the contention problem among shared resources.

1.4 Objective

To develop an efficient thread scheduling technique to minimize the problem of contention among shared resources such as cache interconnects memory bus and prefetcher by the help of data collected in hardware counters [13] and performance monitoring tools which increases the performance of a multi-core processor.

1.5 Thesis Organization

Organization of thesis is done as following: Chapter-2 describes the basic concepts for this thesis. Chapter-3 discusses the scheduling techniques that uses miss rate as the classification scheme and their comparison with default OS scheduler. The scheduling techniques that uses two other metrics contentiousness and sensitivity are discussed and miss rate is proved to be non efficient metric when the workload is CPU intensive . Finally Chapter 4 concludes with the summary of work done.

Chapter 2

Background

Factors causing Contention
The perfect Scheduling policy
Classification schemes
Tools used

Chapter 2

Background

2.1 Factors causing contention:

Firstly, cache contention was considered as the major factor in performance degradation, but later by various experiments, it was found that contention of the memory bus, memory controller [14] and prefetcher hardware is also countable. Though it is not possible to find out how much degradation is caused by which resource individually as all are interrelated, a rough estimate was done by using an experimental system which consists of a server with two sockets [8]. A DRAM controller is shared between the sockets and in each socket four cores share a memory controller, whereas a pair of cores among the four cores share a LLC. By this experimental setup, it was easy to find out the approximate degradation of each resource by placing the threads on different cores and calculating their performance when compared to their solo performance.

The Experiments are done first without prefetching hardware enabled, and the values are calculated for each resource.

Solo_PF_OFF: the application is made to run alone when the preaching is not enabled.

SameCache_PF_OFF: The application is run along with the interfering application by sharing the last level cache when the prefetching is not enabled.

DiffCache_PF_OFF: The application is run along with the interfering application by sharing the same socket, but not the last level cache when the prefetching is not enabled.

DiffSocket_PF_OFF: The application is run along with the interfering application on different socket when the prefetching is not enabled.

Then to calculate the prefetcher contention the whole experiments were again conducted with prefetching hardware enabled

Solo PF_ON: the application is made to run alone when the prefetching is enabled.

SameCache_PF_ON: The application is run along with the interfering application by sharing the last level cache when the preaching is not enabled.

DiffCache_PF_ON: The application is run along with the interfering application by sharing the same socket, but not the last level cache when the prefetching is not enabled.

DiffSocket_PF_ON: The application is run along with the interfering application on different socket when the prefetching is not enabled.

Degradation in performance due to Contention of DRAM Controller

The applications are made to run on two cores of different sockets so that they share only DRAM controller then the degradation due to DRAM controller was calculated by using the following formula

DRAM contention = $\frac{DiffSocket_PF_OFF - Solo_PF_OFF}{Solo_PF_OFF}$

Degradation in performance due to Contention of FSB

The applications are executed on cores of same socket but on cores sharing different last level caches so that they share the front side bus then the degradation due to FSB can be calculated by using the formula

FSB Contention $=\frac{DiffCache_PF_OFF - DiffSocket_PF_OFF}{Solo_PF_OFF}$

Degradation in performance due to Contention of cache

The applications are executed on cores sharing same LLC then the degradation due to cache can be calculated using the following formula.

Cache Contention $= \frac{SameCache_PF_OFF - DiffCache_PF_OFF}{Solo_PF_OFF}$

Degradation in performance due to prefetched hardware

The performance degradation due to prefetching hardware can be calculated by the difference between the degradation caused by all resources and the degradation caused by contention due to FSB, cache and DRAM controller. The total degradation can be calculated as

Total Degradation $= \frac{SameCache_PF_ON - Solo_PF_ON}{Solo_PF_ON}$

Prefetcher contention is calculated as

Prefetcher contention $= TotalDegradation - DRAMContention - FSBContention - CacheContention$

Finally. Results showed that contention due to cache was not only the dominant cause, but the contention of bus, memory controller and prefetcher hardware also has a significant effect on performance degradation.

2.2 The Perfect Scheduling Policy:

Scheduling policy is one of the important components of a scheduling algorithm. The perfect scheduling policy proposed by jiang [15] is used to calculate the best and worst schedules. The performance degradation which an application experiences when it is made to run along with some other application is called as co-run degradation experienced by that application and the vice versa is called co-run degradation caused by that application.

After the co-run degradations of all applications are found out a graph is drawn with applications as nodes and their co-run degradations as the weights of the edges between them. If A and B are considered as two applications, then the weight of the edge between them is calculated by adding co-run degradations caused by each on the other. After the graph is drawn by finding the minimum weight among the edges to find the best schedule

The below graph shows a sample graph drawn with 4 applications and the best and worst schedules of them.

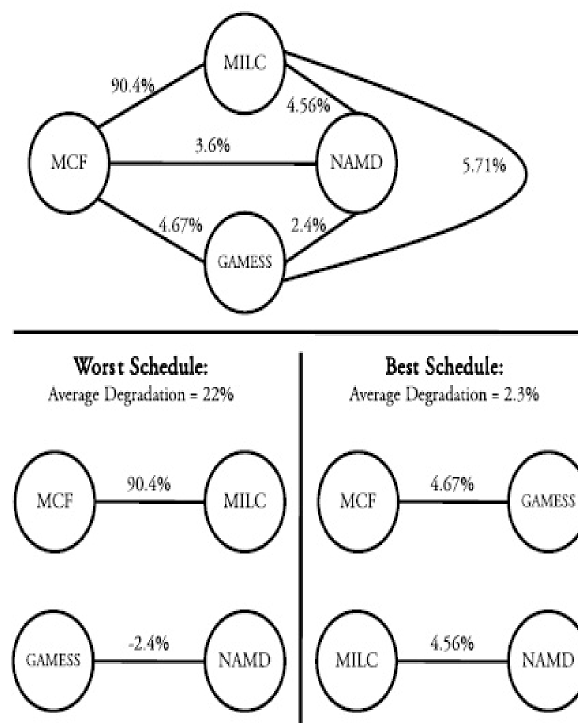

Figure 2.1: Jiang Methodology [6]

2.3 Thread classification schemes:

Thread classification schemes are the most important components of a contention aware scheduling algorithm. They are necessary to know which application should run with which application so that the contention is less by considering the characteristics of applications and the classes to which they are classified. To know which one of the classification schemes is more efficient they are compared with each other by using the perfect policy defined in the previous section as their scheduling policy.

By comparing the classification schemes with each other we get only the relative performance of them. In order to get their actual performance and efficiency results there is a need of finding the optimal scheme and compare them with that scheme. The optimal scheme should have results of degradation , that are actually caused by applications on real world systems. To get this result , certain benchmarks are considered and made to run with each other by considering two applications at a time and the degradations are noted down. Then the applications are made to run alone and execution time is noted down. Then the actual degradation is measured by the difference of above two experiments results.

Mainly four types of classification schemes have been used widely which uses the information collected by stack-distance profiles.

- SDC
- Animal Classes
- Miss rate
- Pain

Miss rate and pain are considered to be efficient schemes, but the pain metric as it considers the concepts of sensitivity (The amount of cache taken away by other applications because of co-running) and intensity (The amount of cache taken away from other applications because of co-running with them) it is more complex to be calculated online. So the Miss rate is considered as efficient classification

scheme as it is a simple metric and one more advantage is that it's unlike SDC considers not only the contention of cache, but also the contention of the other shared resources. Therefore, in algorithms where the parameters to classify are calculated online miss rate is simple and efficient.

Whereas in algorithms where the parameters are not calculated online, but are calculated prior to scheduling, Stack-distance profile data (extra misses because of co-running) is used.

2.4 Basic concepts of Contention Characteristics

Contentiousness and sensitivity are the two characteristics to measure the contention of applications [16]. Miss rate was used as a metric to decide whether the application is contentious or not and the highly contentious. But this conclusion is not always true because there are some applications where they occupy large amount of cache and uses the cache for long time without any misses. Such applications have low miss rate but they are highly contentious because they doesnt allow other applications which are co- running with them to use the resources. And miss rate cannot accurately calculate the bandwidth contention also. for example the applications which stream the data online doesnt need the cache much but they use use the bus vastly which cause the bandwidth contention and it is ignored by miss rate. Therefore miss rate is not an efficient metric to find the whether a memory intensive workload is highly contentious or not.

2.4.1 Contentiousness

Contentiousness of an application is the degradation of performance the application causes to the application(s) that are executing along with it because of its high demand of shared resources [17].An application As contentiousness can be defined using the following formula,

$$Contentiousness_A = \frac{\overline{IPC_{B_i(solo)}} - \overline{IPC_{B_i(co-run)}}}{\overline{IPC_{B_i(solo)}}}$$

Contentiousness of an application A when it is executed along with B_i and the average contentiousness can be shown as,

$Contentiousness_{A(corunB_i)} = \frac{IPC_{B_i(solo)} - IPC_{i(co-run_A)}}{IPC_{B_i(solo)}}$

Average contentiousness of application A is,

$Contentiousness_{A(avg)} = \frac{\sum_i^n Contentiousness_{A(corunB_i)}}{n}$

An application's contentiousness can also be defined as the amount of **pressure** it can put on the shared resources. So contentiousness could be directly predicted by the amount of shared resources that the application is using.

C $= a_1 \times LLCusage + b_1 \times BWusage + c_1 \times Prefusage$

Here C is contentiousness of application , BWusage is bandwidth usage and Prefusage is prefetcher usage and LLCusage is Last Level Cache usage of the application .

The coefficients a_1, b_1 and c_1 are used to denote the relative importance among shared resources contentions. By the approximation of PMU's contentiousness can be calculated by

$C = a_1 \times (L2LinesIn_rate - L3LinesIn_rate) + b_1 \times L3LinesIn_rate$

As L3LinesIn_rate and L2LinesIn_rate include the traffic due to prefetcher, an extra PMU is not needed to measure the usage of prefetcher.

2.4.2 Sensitivity

Sensitivity of an application can be defined as the amount of performance degradation suffered by the application because of the contentiousness of applications which are executing along with it. Sensitivity of an application A can be defined using the following formula,

$$Sensitivity_A = \frac{IPC_{A(solo)} - \overline{IPC_{A(co-run)}}}{IPC_{A(solo)}}$$

where A's IPC is denoted as $IPC_{A(solo)}$ when it is executing alone and when it executes along with some other random programs is given by $IPC_{A(co-run)}$

Sensitivity of an application A when it is executed along with B_i and the average Sensitivity can be shown as,

$$Sensitivity_{A(co-runB_i)} = \frac{IPC_{A(solo)} - IPC_{A(co-run_{B_i})}}{IPC_{A(solo)}}$$

Average sensitivity of application A is,

$$Sensitivity_{A(avg)} = \frac{\sum_i^n Sensitivity_{A(co-runB_i)}}{n}$$

Sensitivity of an application can also be defined as the reliance of the application on the shared resources.

S $= a_2 \times LLC_usage + b_2 \times BW_usage + c_2 \times Pref_usage$

where S is Sensitivity, BWusage is bandwidth usage and Prefusage is prefetcher usage of the application.
The coefficients a_1, b_1 and c_1 are used to denote the relative importance among shared resources contentions. By the approximation of PMU's sensitivity can be calculated by

$$S = a_2 \times (L2LinesIn_rate - L3LinesIn_rate) + b_2 \times L3LinesIn_rate$$

2.5 Tools used:

The Tools that are used to design and develop an efficient scheduling algorithm are

2.5.1 AKULA Toolset

The main problems of developing a scheduling algorithm are addressed as the implementation difficulty and the testing duration. AKULA toolset eases the work of developers by providing an API and by making debugging easier without any modification done in kernel or by using any system calls.

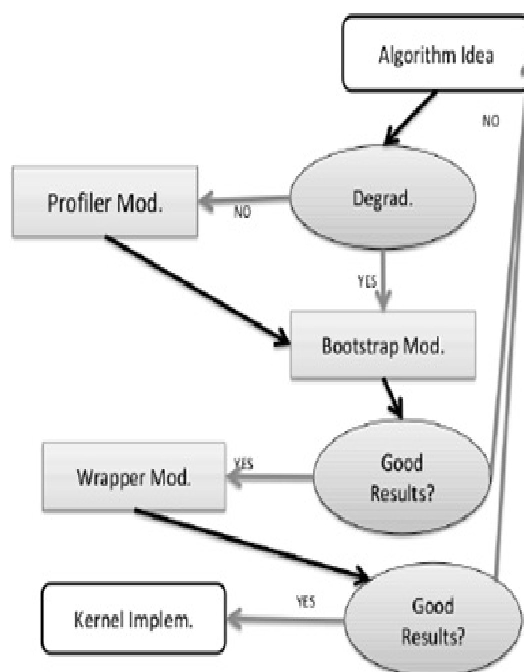

Figure 2.2: Flowchart of AKULA [9]

The toolset consists of three different modules namely

- **Bootstrapping module:** this is the module where the idea of the new scheduling algorithm is evaluated to check whether it can be really implemented. Here no benchmark program is executed for evaluation, but the data required by the algorithm is directly provided by user by calculating it prior to the evaluation.
- **Wrapper Module:** this module helps to run the algorithm in the real world system such and no further modification is needed by it to run a real world system and provides the same results as the evaluation results.

- **Profiler Module:** this module is used to find the parameters needed by the algorithm to perform scheduling. It calculates the miss rate, degradation matrix and solo execution times etc of the benchmark programs by using performance monitoring tools such as Perth or pfmon.

2.5.2 MARSS

A full system simulator MARSS is described which includes QEMU and PTLsim to provide cycle accurate simulation of many x86 core processors of same or different processing capacities. It can also emulate hardware like caches, interconnects, input, output devices and chips. It has the ability to run many OS on the emulated hardware without any modifications done to them.

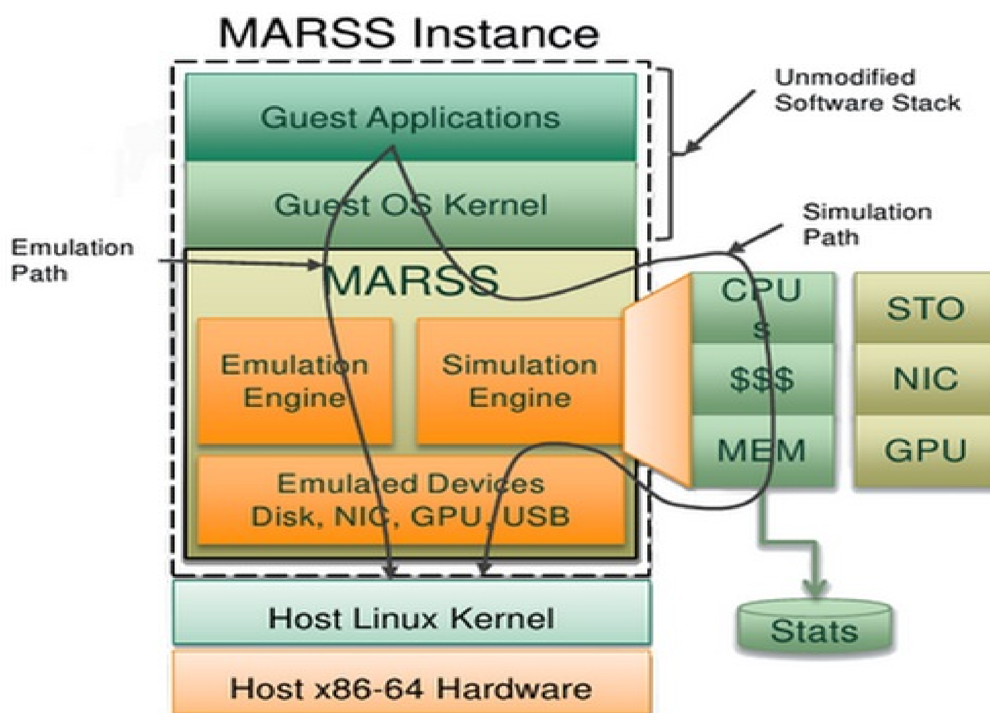

Figure 2.3: MARSS Instance [10]

2.5.3 Oprofile

It is a performance monitoring tool which monitors many performance events by using the information available from hardware performance counters and supplies it to the required user level programs to perform their tasks. It works for almost all architectures and the events that can be monitored varies from one architecture to another.

- To get all the available events in a system the following command can be used
  opcontrol –help
- And the profiling can be started by the command
  opcontrol start
- The profile information can be saved to a file by the following command
  opcontrol dump
- The profiler can be stopped by the command
  opcontrol stop
- The data available after profiling can be shown to users in the form of a report by the following command
  opreport or opreport symbols

Chapter 3

Implementation

Distributed Intensity

Swap

Contentiousness and sensitivity

Chapter 3

Implementation

The AKULA Toolset is used to implement the scheduling algorithms with the help of its API by defining all the system calls necessary to obtain the data from the kernel. Which in turn eases the work of the developer. The MARSS full system simulator is used to simulate a multi-core system with 4 cores and run Linux (Ubuntu) OS on top of it. Then the AKULA can be installed and scheduling algorithms can be run on top of the simulated system similar to as of the real world system.

3.1 Distributed Intensity

This algorithm uses the concept of sorting of threads according to their miss rates and then assigning to the cores in an efficient way so that miss rate distributed equally among the processing units. Assignment of threads are done first by arranging threads according to their miss rate in non-ascending order in an array. Then the assignment of first thread is done to the first core of one socket, then the assignment of second thread is done to the core of different socket and so on until all of the sockets are assigned one thread. Then the array is reversed and the threads are assigned in the same order so that the no socket is assigned with more miss rate threads than others and the miss rate is equally distributed among all cores.

3.1.1 Algorithm

Algorithm 1 Algorithm 1: Runs every time at the beginning of scheduling interval

```
1: // A boolean global array Orders[a] is initialized, every member of the array
   specifies the entities browsing order at the corresponding level of the memory
   hierarchy. OS gives every entity its own ID. If Orders[l]=0 then the entities
   are browsed in increasing order, Order[a]=1 then the entities are browsed in
   descending order.
2: for a=0;a < n.o of levels of memory hierarchy > ; a++ do
3:    Orders[a] := 0;
4:    all the entities of all levels are to be browsed in increasing order
5: end for
6: Applications Threads are sorted in descending order of miss rate
7: S be the sorted array of threads
8: // Application threads are spreaded among all the cores
9: while S! = ϕ do
10:   initial thread is taken from the array(the most aggressive) t∈ S
11:   DIalg() is invoked to assign the thread
12:   DIalg(t,< mac >, 0)
13:   Process is repeated for each interval
14: end while
```

3.1.2 Benchmarks and Workloads

- Two benchmarks based on their miss rates are considered
  - Devil (high miss rate)
  - Turtle (low miss rate)
- Three Workloads formed from three unique schedules of four applications are used (2 devils, 2 turtles).

3.1.3 Results

Profiling:

- The benchmarks are profiled and factors are found out by using Perf:
  - Solo Execution time
  - Degradation Matrix

Algorithm 2 Algo 2: (t, p_parent, memory hierarchy level a)

1: P_all is the entities array at level a+1
2: P_childrn be the entities array on level l+1 in container p_parent.
3: The Entities are browsed in P_childrn in order Orders[a+1] and entity with min n.o of threads allocated is found: p_minimum $\in$ P_childrn.
4: **if** p_minimum is a <core> **then**
5: // bottom of hierarchy is reached.
6: thread t is assigned to core p_minimum.
7: **else**
8: n.o of threads allocated to p_minimum are incremented
9: DIalg(t,p_minimum,a + 1);
10: // DIalg()is invoked recursively for thread assignment in low hierarchy level.
11: **end if**
12: **if** threads are allocated equally to each entity p $\in$P_all **then**
13: // browsing order is reversed in this level.
14: Orders[a+1] := /Order[a+1]
15: **end if**

- Miss rate
- IPC

- Profiling is done each time with each bench mark as input and provides 4 files of above factors as output.

Results of Profiling:

Default Scheduler:

1. Solos.txt

Thread name	Execution time
0	30
1	33

Table 3.1: Solo execution time

2.Degradation_matrix.txt:

Thread1	Thread2	Degradation
0	0	0.38118704256098573
0	1	0.49531665552043574
1	0	0.35729021361703934
1	1	0.32955678828701557

Table 3.2: Degradation matrix

3.Miss_rate.txt

Thread name	Miss rate
0	11.010513177133285
1	0.0

Table 3.3: Miss rates

DI Scheduler:

1. Solos.txt

Thread name	Execution time
0	29
1	75

Table 3.4: Solo execution time

2.Degradation_matrix.txt:

Thread1	Thread2	Degradation
0	0	0.3592374911626906
0	1	0.5103094358890173
1	0	0.7658604189036874
1	1	0.6826307624550999

Table 3.5: Degradation matrix

3.Miss_rate.txt

Thread name	Miss rate
0	10.23452347865
1	0.0

Table 3.6: Miss rates

Results of Simulation

Thread Name	Processor Time (s)	% Degrad
devil1	64.0	120.6896551724138
devil2	64.0	120.6896551724138
turtle1	101.0	34.66666666666667
turtle2	101.0	34.66666666666667

Total Length: 50.0

Max Degradation: 6.382978723404255

Average Degradation: 4.662077596996245

Top Half Degradation: 6.382978723404255

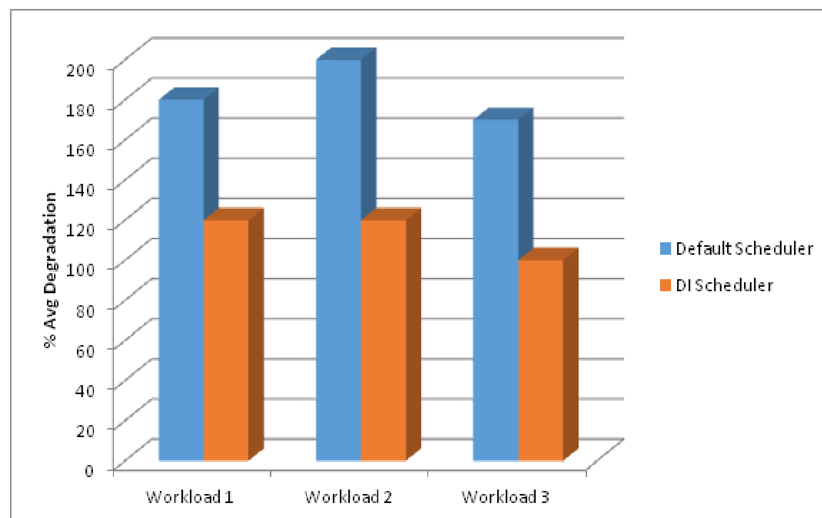

Figure 3.1: Comparision of schedulers

3.2 Swap

The swap algorithms works by finding the IPC per socket initially. Then it randomly selects a core and then picks the core of that thread and swaps with some other thread of a core present in another socket. then the IPC is calculated once again. If the IPC of both groups increase then the swap is considered as valid and threads are allowed to execute. Otherwise they are swapped back to their original position.

3.2.1 Algorithm

Algorithm 3 Swap(thread t1,t2, e_parent, hierarchy level l)

1: P_all be entities array of hierarchy level a+1 choose randomly two arrays and find the IPC (IPC_{prev1},IPC_{prev2})of them.
2: Let P_children1, P_children2 be the entities array at hierarchy level l+1 with containers p_parent1,p_parent2.
3: browse the entities in P_children1, P_children2 in order Orders[a+1] and determine any random entity p_ran1,p_ran2
4: **if** e_ran1 and e_ran2 are $< core >$ **then**
5: // bottom of the hierarchy is reached
6: Choose a thread t1,t2 from each and swap them
7: **end if**
8: IPC(IPC_{new1}, IPC_{new2}) of P_children1 and P_children 2 are found out.
9: **if** $IPC_{new1} > IPC_prev1$ and $IPC_{new2} > IPC_{prev2}$ **then**
10: good swap and noted down
11: **else**
12: Discarded and threads are swapped back
13: **end if**

3.2.2 Benchmarks and Workloads

- Two benchmarks based on their miss rates are considered
  - Devil (high miss rate)
  - Turtle (low miss rate)
- Three Workloads formed from three unique schedules of four applications are used (2 devils, 2 turtles).

3.2.3 Results

Simulation Results:

Thread Name	Processor Time (s)	% Degrad
Turtle1	35.0	2.941176470588235
Turtle2	35.0	2.941176470588235
devil1	50.0	6.382978723404255
devil2	50.0	6.382978723404255

Max Degradation: 120.6896551724138

Average Degradation: 77.67816091954023

Top Half Degradation: 120.6896551724138

Unfairness: 113.74206309334673

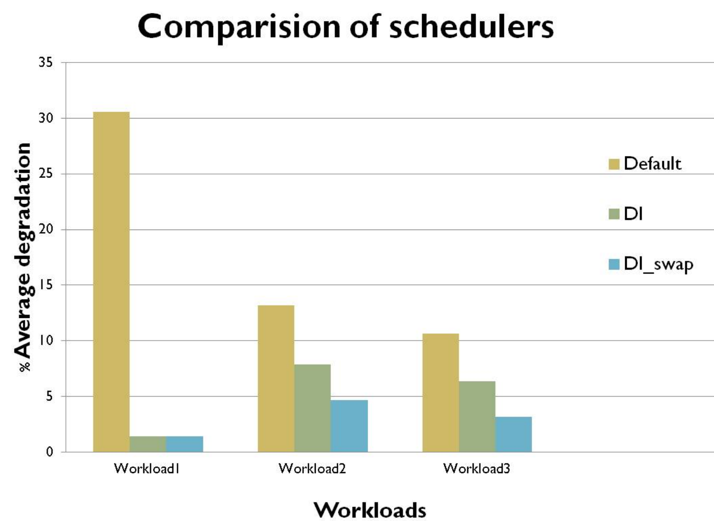

Figure 3.2: comparision of swap scheduler

3.3 Contentiousness and sensitivity

3.3.1 Experimental setup

The benchmarks are run on the Intel(R)core (TM) i7-3770 CPU @ 3.40 GHz quad core processor. The performance monitoring is done by Oprofile by collecting the information from hardware performance counters.

3.3.2 Benchmarks

The PARSEC benchmark suite is used for testing whether miss rate and miss ratio can efficiently calculate the contentiousness and sensitivity or not. The PARSEC benchmark suite consists of various benchmarks taken from different areas like mining, image processing etc. and are specially designed to run on multi-core machines to find the CPU, memory and other hardware characteristics in order to improve their performance.

3.3.3 Results

The relation between the contentiousness and miss rate is shown by the following graph

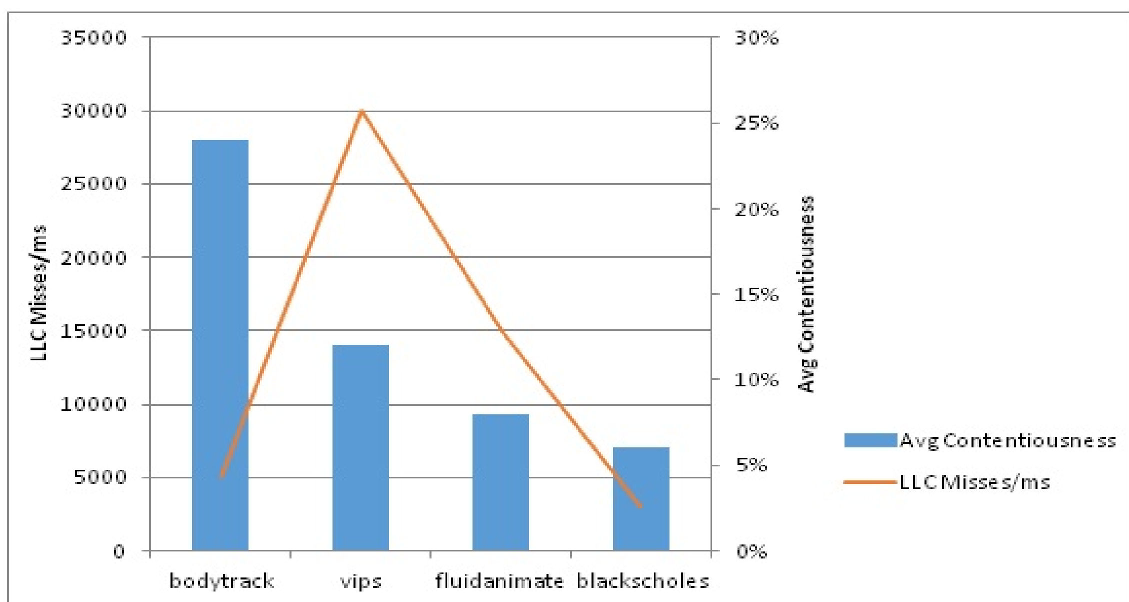

Figure 3.3: Contentiousness vs Miss rate

The relation between the sensitivity and miss rate is shown by the following graph

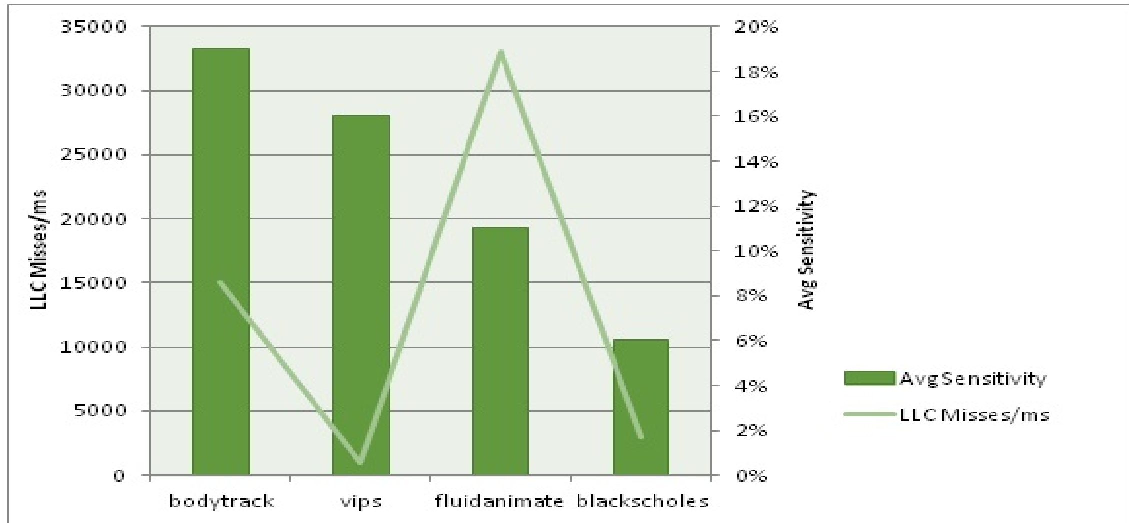

Figure 3.4: Sensitivity vs Miss rate

The relation between the contentiousness and LLC_LINES_In is shown by the following graph

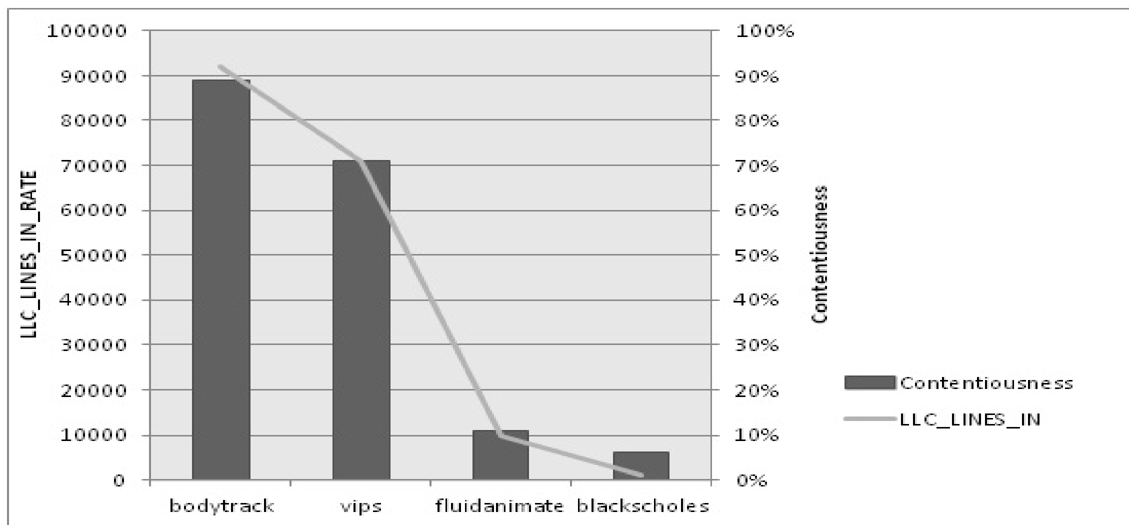

Figure 3.5: Sensitivity vs Miss rate

Chapter 4

Conclusion

Chapter 4

Conclusion

In our research work we have attempted to solve the contention problem of shared resources in multi-core architectures through different techniques.

- Factors which cause contention and cause degradation of performance are learned.
- Different classification schemes to classify applications and scheduling policies are learnt.
- Miss rate is found out to be easy and efficient metric to schedule workload consisting of almost equal cpu-intensive and memory-intensive applications
- Contentiousness and sensitivity are realised as pressure and reliance of applications and are found out to be not varying according to miss rate.
- By the information obtained from PMUs the contention characteristics are found and efficiency of contention aware algorithms is increased.

Bibliography

[1] T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald, "Methods for modeling resource contention on simultaneous multithreading processors," in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pp. 373–380, IEEE, 2005.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 340–351, IEEE, 2005.

[3] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 25–38, IEEE Computer Society, 2007.

[4] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using os observations to improve performance in multicore systems," *IEEE micro*, vol. 28, no. 3, pp. 54–66, 2008.

[5] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 257–265, ACM, 2010.

[6] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 129–142, ACM, 2010.

[7] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 76–86, IEEE, 2010.

[8] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, p. 8, 2010.

[9] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Akula: a toolset for experimenting and developing thread placement algorithms on multicore systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 249–260, ACM, 2010.

[10] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, pp. 1050–1055, ACM, 2011.

[11] S. Blagodurov and A. Fedorova, "User-level scheduling on numa multicore systems under linux," in *Linux Symposium*, p. 81, 2011.

[12] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 4, 2012.

[13] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 56–65, 2009.

[14] D. Xu, C. Wu, and P.-C. Yew, "On mitigating memory bandwidth contention through bandwidth-aware scheduling," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 237–248, ACM, 2010.

[15] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 220–229, ACM, 2008.

[16] L. Tang, J. Mars, and M. L. Soffa, "Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 12–21, ACM, 2011.

[17] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword, "Synchronization via scheduling: techniques for efficiently managing shared state," in *ACM SIGPLAN Notices*, vol. 46, pp. 640–652, ACM, 2011.