

Empirical Performance Studies on Implications of Shared Memory Hierarchy Resources in Multicore Computing Systems

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

by
Jitendra Kumar Rai



**Department of Computer & Information Sciences
University of Hyderabad
Hyderabad - 500046
INDIA
October 2012**



CERTIFICATE

This is to certify that the thesis work entitled “Empirical Performance Studies on Implications of Shared Memory Hierarchy Resources in Multicore Computing Systems” submitted by Jitendra Kumar Rai bearing Reg. No. 06MCPC04 in fulfillment of the requirements for the award of the degree of Doctor of Philosophy (Computer Science) of the University of Hyderabad, is a bona fide work carried out by him under our supervision.

The matter embodied in this thesis has not been submitted previously in part or in full to this or any other University or Institution for the award of any degree or diploma.

Dr. Atul Negi

(Supervisor)

Department of Computer & Information
Sciences (DCIS)

University of Hyderabad
Hyderabad-500056, INDIA

Dr. Rajeev Wankar

(Supervisor)

Department of Computer & Information
Sciences (DCIS)

University of Hyderabad
Hyderabad-500056, INDIA

HEAD

Department of Computer & Information
Sciences (DCIS)

University of Hyderabad
Hyderabad-500056, INDIA

DEAN

School of Mathematics & Computer and
Information Sciences

University of Hyderabad
Hyderabad-500056, INDIA

DECLARATION

I, **Jitendra Kumar Rai**, hereby declare that the work presented in this thesis entitled “**Empirical Performance Studies on Implications of Shared Memory Hierarchy Resources in Multicore Computing Systems**” has been carried out by me under the supervision of **Dr. Atul Negi** and **Dr. Rajeev Wankar** as per the Ph.D. ordinances of the University. I also declare that it has not been submitted previously in part or in full to this University or any other University or Institution for the award of any degree or diploma.

Jitendra Kumar Rai
Reg. No. 06MCPC04

Acknowledgements

This thesis has been made possible by generous support of many people. My supervisors Dr. Atul Negi and Dr. Rajeev Wankar have provided the guidance and wisdom. I am also grateful to members of Doctoral Review Committee, especially Dr. Arun Agarwal and Dr. Hrushikesh Mohanty as well as the faculty members of the Department of Computer & Information Sciences, who reviewed the work and provided directions for further explorations. I also thank the Head of the Department of Computer & Information Sciences, for the encouragement.

I am also thankful to Director ANURAG, Hyderabad, for support and encouragement. I thank all the members of the Department of Computer & Information Sciences and ANURAG, who in some or other way provided support.

I also express my thanks to developers and users of open source software tools, especially Stephane Eranian and Corey Ashford of perfmon2; Peter Reutemann, Mark Hall and Harri M.T. Saarikoski of weka. I thank the anonymous reviewers, who reviewed part of the work sent for publication.

I would like to thank my family members especially my parents and wife for their understanding and support. I express my gratitude to the almighty, without whose blessings nothing is possible.

Abstract

Multicore processor architectures evolved and became dominant in recent years. It is mainly due to the diminishing returns from the efforts to increase the performance with single execution core on the chip. However multicore architectures involve sharing of resources especially the memory hierarchy resources such as processor caches, prefetchers and bus among the cores. This sharing may cause performance degradation of the programs co-running on multicore processors as compared to their solo run performance.

In the literature attempts to solve this problem proposed analytical models, which are very complex and difficult for use in real systems. Many of the previous works proposed specialized support from processor hardware, which may take time to become available in future generations of multicore processors.

It is observed that programs running simultaneously on different cores of a multicore processor, have complex interactions due to the use of shared memory hierarchy resources. In this work, we take up a unique approach for modeling the performance implications of such interactions by applying machine learning techniques. Machine learning techniques focus on methods that learn to recognize complex patterns from data. Here we give emphasis to empirical model building.

The work demonstrates the application of machine learning techniques to build models for characterization and performance prediction on multicore processor based systems. By characterization, we mean to characterize program memory behavior especially with respect to utilization of shared caches on multicore processors. The model developed for the program memory behavior characterization is later-on used for improving the process scheduling (i.e. CPU scheduling) on multicore processors. We also developed methodology to build a model that predicts performance on multicore processors. Such methodologies can further be utilized in performance oriented research on multicores. We describe the application of the built model for improving the multicore simulation in the AKULA tool-set. AKULA has been recently developed by researchers for rapid prototyping and evaluation of scheduling algorithms for multicore processors.

The efficacy of the developed methodologies to build the models, was validated using existing commodity multicore processor based systems. We observed that machine learning techniques are helpful in performance related studies on multicore processor based computing systems. We observed performance improvement up to 76% for 4-cores and 54% for 8-cores as compared to default linux kernel process scheduler on our experimental multicore platform by im-

proving process scheduling. The process scheduling was improved by utilizing the model developed using machine learning techniques, so that the interference among the co-running programs due to usage of resources shared among the cores is mitigated. The approximate average cost of the model was about 0.00075% of the total time (i.e. 7-8 cycles per million cycles), as observed in the experiments. Thus here we demonstrate that machine learning methods have been effective in building program performance models on existing commodity multicore based platforms.

Contents

Title Page	i
Certificate	ii
Declaration	iii
Acknowledgements	iv
Abstract	v
List of Tables	x
List of Figures	xii
List of Publications	xiv
1 Introduction	1
1.1 Motivation	2
1.1.1 Evolution of Processor Architecture	2
1.1.2 Performance Related Studies	8
1.1.3 Commodity Multicore Processors	9
1.2 Research Objectives and Problem Statement	11
1.3 Research Contributions	12
1.4 Organization of the Thesis	14
2 Review of Performance Studies and Related Background	17
2.1 Shared Memory Hierarchy Related Studies	17
2.1.1 Hardware Based Solutions	18
2.1.2 Software Based Solutions	21
2.1.3 Our Approach for Program Memory Behavior Character- ization	26
2.2 Overview of Machine Learning Techniques	26

2.3	Performance Prediction	29
2.3.1	Previous Studies on Performance Prediction	30
2.3.2	Our Approach for Performance Prediction	31
2.4	Conclusions	32
3	Characterization of Program Memory Behavior	33
3.1	Performance Issues on Multicores	34
3.2	Program Memory Behavior	38
3.3	Metric to Characterize Program Memory Behavior	38
3.4	Program Memory Behavior Characterization Issues	42
3.5	Hardware Performance Counters	43
3.6	Machine Learning Algorithms Used	43
3.7	Experimental Setup	44
3.7.1	Experimental Platforms	45
3.7.2	Processor Memory Hierarchy on Experimental Platforms	46
3.7.3	Workload	47
3.8	Methodology	47
3.8.1	Solo-run Experiment	47
3.8.2	Concurrent-run Experiment	49
3.8.3	Generating Program Attributes and Class Variable . . .	49
3.9	Results on Prediction Accuracy for Predicting Solo-run Last Level Cache Stress	52
3.10	Assessing Transferability of the Trained Models for Predicting Solo-run Last Level Cache Stress	55
3.10.1	Difference in Last Level Cache Organization on Intel Xeon X5482 and Intel Core2 6300 Processors	56
3.10.2	Prediction Accuracy Metrics of Trained Regression Mod- els on Test Data-set	56
3.11	Conclusions	57
4	Improving Process Scheduling	58
4.1	Process Scheduler in Commodity OS	58
4.2	Process Scheduling Issues for Multicores	60
4.3	Role of Meta-scheduler	60
4.4	Policy Framework of Meta-scheduler	61
4.5	Performance Evaluation of Meta-scheduler	66
4.5.1	Experimental Platform	67
4.5.2	Workload	67
4.5.3	Experiments	68

4.6	Results	69
4.7	Conclusions	72
5	Performance Prediction	73
5.1	Experimental Setup	74
5.1.1	Experimental Platforms	74
5.1.2	Processor Memory Hierarchy on the Experimental Platforms	76
5.2	Experiments	77
5.3	Generation of Solo-run Program Attributes & Class Variable . .	78
5.4	Results on Prediction Accuracy for Predicting Concurrent-run Performance	82
5.5	Assessing Transferability of Model for Predicting Concurrent-run Performance	84
5.5.1	Differences in Last Level Cache Organization on Processors used to Generate Train-set and Test-set Data	85
5.5.2	Prediction Accuracy Metrics of Trained Regression Models on Test Data-set	86
5.6	Application of Machine Learning Based Performance Prediction for Multicore Simulation	87
5.6.1	Overview of the AKULA Tool-set	87
5.6.2	Improving Multicore Simulation in AKULA	90
5.7	Conclusions	92
6	Conclusions and Future Work	94
6.1	Summary of Contributions	94
6.2	Conclusions	96
6.3	Future Work	97
	References	101
A	Perfmon2 Interface	115
B	WEKA Machine Learning Workbench	117

List of Tables

1.1	List of some of the commodity multicore processors.	10
3.1	Solo-run and concurrent-run performance of SPEC cpu2006 programs on Intel Xeon X5482 processor.	34
3.2	Percentage degradation in performance of some of the SPEC cpu2006 programs during concurrent-run as compared to their solo-run on Intel Xeon X5482 processor.	36
3.3	Specifications of DELL Precision T7400 workstation used for generating data to build regression models.	45
3.4	Specifications of DELL Precision 390 workstation used for generating data to test regression models.	45
3.5	Sample of train-set data of program memory behavior from Intel Xeon X5482 processor.	51
3.6	Prediction accuracy metrics achieved in 10 times 10-fold cross validation for predicting solo-run last level cache stress on Intel XeonX5482 processor.	54
3.7	Time taken for training the algorithms to build regression models for predicting solo-run last level cache stress.	55
3.8	Last level cache related data of Intel Xeon X5482 and Intel Core2 6300 processors.	56
3.9	Prediction accuracy of trained models on test data from Intel Core2 6300 for predicting solo-run last level cache stress.	57
4.1	Specifications of the experimental platform used for performance evaluation of intelligent scheduling done by meta-scheduler.	67
4.2	Description of SPEC cpu2006 applications used in performance evaluation with meta-scheduler.	67
4.3	Sets of SPEC cpu2006 applications used for performance evaluation with meta-scheduler.	68
4.4	Characteristics of the workload sets used for performance evaluation with meta-scheduler.	68

4.5	IPC , IPC_M and normalized speedup for various sets of workloads in 4-core experiments on Intel quad-core Xeon X5482 processor based platform.	70
4.6	IPC , IPC_M and normalized speedup for various sets of workloads in 8-core experiments on Intel quad-core Xeon X5482 processor based platform.	71
5.1	Specifications of AMD quad-core Phenom 9650 processor based platform used for generating data to test regression models built for performance prediction.	75
5.2	Specifications of AMD triple-core Phenom 8450 processor based platform used for generating data to test regression models built for performance prediction.	75
5.3	Sample of train-set data for performance prediction collected from Intel Xeon X5482 processor.	83
5.4	Prediction accuracy results from 10 times 10-fold cross validation for predicting concurrent-run performance on Intel Xeon X5482 processor.	84
5.5	Time taken to build regression models using train-set having 3025 instances from Intel Xeon X5482 processor for predicting concurrent-run performance.	84
5.6	Last level cache related data of Intel Xeon X5482, Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors used in performance prediction experiments.	85
5.7	Prediction accuracy of the trained model on test data from Intel Core2 6300 processor for predicting concurrent-run performance.	86
5.8	Prediction accuracy of the trained model on test data from AMD Phenom 9650 processor for predicting concurrent-run performance.	86
5.9	Prediction accuracy of the trained model on test data from AMD Phenom 8450 processor for predicting concurrent-run performance.	86
5.10	Example solo-run performance data used by AKULA <i>bootstrap</i> module for multicore simulation.	89
5.11	Example performance degradation matrix for concurrent-run, used by AKULA <i>bootstrap</i> module for multicore simulation.	89

List of Figures

1.1	Schematic views of singlecore processors.	5
1.2	Schematic views of multicore and multicore-multithreaded processors.	7
1.3	Thesis organization.	15
3.1	Concurrent-run and solo-run performance of some of the SPEC cpu2006 benchmarks on Intel quad-core Xeon X5482 processor. .	36
3.2	Percentage degradation in performance of some of the SPEC cpu2006 programs during concurrent-run as compared to their solo-run on Intel Xeon X5482 processor.	37
3.3	Concurrent-run and solo-run last level cache (LLC) stress of SPEC cpu2006 benchmark program 429.mcf on Intel Xeon X5482 processor.	39
3.4	Suitability of solo-run last level cache stress as a metric to characterize the program memory behavior.	40
3.5	Phase behavior of SPEC cpu2006 benchmark program 433.milc on Intel Xeon X5482 processor.	41
3.6	Last level (L2) cache sharing on Intel Xeon X5482 processor used for generating data to build regression models.	46
3.7	Last level (L2) cache sharing on Intel Core2 6300 processor used for generating data to test regression models.	47
3.8	Methodology followed to build the model by training the machine learning algorithms.	48
4.1	Block diagram of the meta-scheduler.	62
4.2	Part of built model tree used by meta-scheduler.	64
4.3	Topology of Intel quad-core Xeon X5482 processor.	66
4.4	Normalized speedup with meta-scheduler in 4-core experiments on Intel quad-core Xeon X5482 processor based platform.	70
4.5	Normalized speedup with meta-scheduler in 8-core experiments on Intel quad-core Xeon X5482 processor based platform.	71

5.1	Last level (L3) cache sharing on AMD Phenom 9650 and AMD Phenom 8450 processors used for generating data to test regression models built for performance prediction.	76
5.2	Framework provided by AKULA tool-set.	88
A.1	Output of Pfmmon session on Intel quad-core Xeon X5482 processor based experimental platform.	116
B.1	Output of Weka Explorer.	118

List of Publications

Journals:

- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Characterizing L2 Cache behavior of Programs on Multi-core Processors: Regression Models and Their Transferability”, *International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)*, ISSN: 2150-7988 vol. 2 (2010), pp. 212–221. <http://www.mirlabs.org/ijcिसim>.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “A Machine Learning based Meta-Scheduler for Multi-core Processors”, *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, ISSN: 1947-9220 vol. 1, no. 4, (2010), IGI-Global, USA, 2010, pp. 46–59.
Also appeared in *Machine Learning: Concepts, Methodologies, Tools and Applications*, ed. Information Resources Management Association, USA, 2012, pp. 522–534, doi:10.4018/978-1-60960-818-7.ch311.
- J. K. Rai, A. Negi and R. Wankar, “Using Machine Learning Techniques for Performance Prediction on Multi-cores”, *International Journal of Grid and High Performance Computing (IJGHPC)*, ISSN: 1938-0259 vol. 3, no. 4, (2011), IGI-Global, USA, 2011, pp. 14–28.
Also in *Applications and Developments in Grid, Cloud, and High Performance Computing*, ed. Emmanuel Udoh, IGI-Global, USA, 2013, pp. 259-273, doi: 10.4018/978-1-4666-2065-0.ch017.

Conferences:

- J. K. Rai, A. Negi, R. Wankar and K.D. Nayak, “Using Machine Learning to Characterize L2 Cache behavior of Programs on Multicore Processors”, in *Proceedings of 2009 International Conference on Artificial Intelligence and Pattern Recognition (AIPR-09)*, Orlando, Florida, USA July 13-16, 2009, pp. 301–306.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “On Prediction Accuracy of Machine Learning Algorithms for Characterizing Shared L2 Cache behavior of Programs on Multicore Processors”, in *Proceedings of The 2009*

IEEE International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN2009), Indore, India, July 23-25, 2009, pp. 213–219.

- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Characterizing L2 Cache behavior of Programs on Multi-core Processors: Regression Models and their Transferability”, in *Proceedings of The Eighth IEEE International conference on Computer Information Systems and Industrial Management Applications (CISIM 2009)*, held in conjunction with World Congress on Nature Biologically Inspired Computing 2009, (NaBIC 2009), Coimbatore, India, Dec. 09-11, 2009, pp. 1673–1676.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Performance Prediction on Multi-core Processors”, in *Proceedings of The IEEE 2010 International Conference on Computational Intelligence, Communication Systems and Networks, (CICN 2010)*, Bhopal, India, Nov. 26-28, 2010, pp. 633–637.
- J. K. Rai, A. Negi and R. Wankar, “Machine Learning Based Performance Prediction for Multi-core Simulation”, in *Proceedings of The 5th Multi-Disciplinary International Workshop on Artificial Intelligence MI-WAI 2011*, Hyderabad, India, Dec. 07-09, 2011, LNAI 7080, Springer-Verlag Berlin Heidelberg, pp. 236–247.

Chapter 1

Introduction

This thesis is concerned with the interactions of software and emerging processor architecture, particularly multicore architecture. The cores present on multicore processors share various memory hierarchy resources such as processor caches, prefetchers and bus. The sharing of memory hierarchy resources among programs simultaneously running on different cores of a multicore processor, causes those programs to have complex interactions. These interactions may result in severe performance degradations. The thesis work involves modeling the performance implications of such interactions.

Previous research works (details of the previous works are given in [chapter 2](#)) proposed analytical models for program behavior with respect to use of memory hierarchy resources (especially caches). These models are fairly complex to be used in real systems. Some of the previous works proposed specialized support from processor hardware for this purpose. It may take time for such specialized support to become available on commodity multicore processors.

The thesis work uses machine learning techniques for modeling the performance implications of shared memory hierarchy resources in multicore computing systems. The discipline of machine learning provides techniques, which can be applied to automatically learn the complex patterns and make intelligent decisions based on data.

We measure how a selection of workloads perform on multicore processors and investigate how data driven methods such as machine learning techniques can be effectively utilized to build models for some of the performance critical aspects of computing systems based on these processors. As part of the study, we also explore the utilization of the built models for improving the process (i.e. CPU) schedule on multicores and simulation of multicore processors.

The chapter is organized as follows: [Section 1.1](#) provides the motivation behind the work carried-out in the thesis. In [section 1.2](#), we state the research

objectives and the problem statement. In section 1.3, we mention the research contributions of the thesis. Section 1.4 provides organization of the thesis.

1.1 Motivation

The motivation for the work basically comes from the developments in processor architecture and associated performance studies. Some of which, especially by Herescu et al. [1] and Fedorova et al. [2] identified the performance issues with emerging multicore architectures. We begin with an overview of the evolution of processor architecture in section 1.1.1. The architectural developments induce new research avenues, involving performance evaluations and characterization of the new upcoming systems. Section 1.1.2 provides a brief description of performance and characterization studies, which have been associated with the evolution of the processor architecture. Section 1.1.3 provides information on some of the commodity multicore processors available in the market that reflects the trend towards growth in number of cores.

1.1.1 Evolution of Processor Architecture

In this section, a review of developments in the processor architecture is given. We begin with the single-core processor architecture and describe the evolution towards multicores. We also mention the reasons, why the multicore architecture became prevalent and why they seem to be the computing engines for future platforms.

The first microprocessor that became commercially available was Intel 4004 [3]. It was released in year 1971. Over the past four decades, the microprocessor industry has seen consistent gains in application performance as a result of multiplicative effect of growth in transistor count and higher clock frequencies. The growth in transistor count has been the result of Moore's law. Gordon Moore predicted in the year 1975, that the number of transistors on an integrated circuit would double every two years [4]. This actually was a revision of his earlier prediction made in the year 1965, according to that the number of transistors would double every year [5]. The advances in the integrated circuit manufacturing technology made it possible to shrink the circuit elements, which allowed more and more transistors to be placed in a single microprocessor.

The three main areas, which contributed to performance gains in the past are clock speed, execution optimization and cache [6]. Increasing the processor clock speed resulted in getting more cycles, which increased the speed at which

the central processing unit (CPU) performed the work. Optimizing execution flow is about doing more work per cycle. It includes having more powerful instructions as well as various other optimizations like pipelining, branch prediction, making the pipelines deep and superscalar architectures [7]. Pipelining involves executing different sub-steps of sequential instructions simultaneously. Branch prediction allows processors to fetch and execute instructions without waiting for a branch to be resolved. Increasing the depth of pipeline (also called superpipelining) means using a longer pipeline with more stages. It was introduced to have architecture with more stages, where each stage does less work so that the processor can be scaled to higher clock frequency. Superscalar architectures include parallel execution units, which can execute instructions simultaneously. Processors were provided with bigger and multiple level of caches so that frequently accessed instructions and data can be kept in the fast accessed caches. This reduced the latency caused due to frequent memory access.

Out of order processing involves executing instructions in an order different from the order they appear in the program. It emerged as an execution optimization technique for execution efficiency to utilize the CPU while it stalled on memory. Out of order execution strove for instruction-level parallelism (ILP) within a sequence of instructions, or thread of control. It required knowing what instruction the program will execute in the future. Programs have branches and CPU needs to predict which instructions will be executed after the branch and predicting the future is difficult. The reliance on a single thread of control to find instruction-level parallelism limits the parallelism available for many applications, and the cost of extracting parallelism from a single thread became prohibitive due to large increase in overall complexity.

The performance scaling in the single core processors largely through increasing clock speed almost touched its limit. As the chip geometries shrink and clock speeds rise, the transistor leakage current increases, which leads to excessive power consumption and heat. The advantages of increased clock speed are also negated by memory latency, as the memory access speeds are not scaling on par with processor clock speeds. There is large and growing mismatch between the processor (CPU) and off-chip main memory in terms of speed as well as bandwidth. Many researchers have referred to this problem as “memory / bandwidth wall” in their works [8] [9] [10].

Another paradigm emerged to improve utilization of CPU resources by leveraging on thread and process level parallelism. A single physical processor can have one or multiple cores and each core can have one or multiple hard-

ware threads. A single core processor with multiple hardware threads is called multithreaded processor while a multicore processor having multiple hardware threads per core is also called multicore-multithreaded or chip-multithreaded processor. The multicore or chip-multiprocessing architectures provide a way to scale the performance, while keeping the heat dissipation and power consumption under limit [11]. Some of the approaches that emerged on the architecture front to exploit the parallelism are described as follows:

1. **Multithreading (MT):** The processor allows more than one thread of execution to exist on the CPU at the same time. It maintains hardware state (program counters and registers) for several threads. Some of the methods used to achieve multi-threading are mentioned as follows:
 - **Fine-grained Multithreading** – In one cycle the processor executes instructions from one of the threads. On the next cycle it switches to context of different thread and executes instructions from the new thread [12].
 - **Coarse-grained Multithreading** – Here a single context (thread) utilizes all the processor resources until it reaches a long-latency operation such as accessing memory due to cache miss, at that point the processor switches to another context (thread) [13].
 - **Interleaving** – A variant of fine-grained multithreading, where issuing of instructions is switched every cycle in round-robin manner between available contexts. Whenever a context encounters a long-latency operation, it becomes unavailable, in such case the processor squashes only those instructions in the pipeline which belong to the unavailable context [14][15].
 - **Simultaneous Multithreading** – It combines hardware features of wide-issue superscalars and multithreaded processors. It has the ability to issue multiple instructions each cycle like superscalars and it maintains hardware state for several programs (threads). Hence this processor can issue multiple instructions from multiple threads each cycle. It exploits both instruction-level and thread-level parallelism [16].
2. **Chip-multiprocessing (CMP):** It uses relatively simple single-thread processor cores on a die [17].
3. **Chip-multithreading (CMT):** Here each core on a die supports multiple hardware threads (contexts) [15].

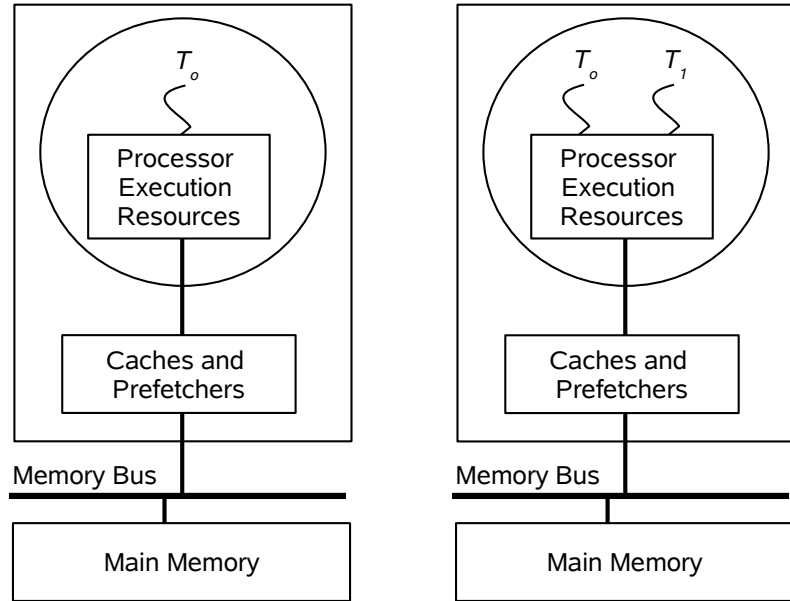
4. **Single ISA Heterogeneous (asymmetric) Chip Multiprocessors:**

It is a multicore processor where all the cores execute the same instruction set architecture (ISA) but have different performance and power characteristics. For example, on a single processor chip some cores are provided with wide issue and out of order processing capability while others are simple single issue CPUs [18].

5. **Multiple ISA Heterogeneous (asymmetric) Chip Multiprocessors:**

These processors have cores that execute instructions belonging to different instruction set architectures (ISAs). They typically address data-level and instruction-level parallelism simultaneously. Any given instruction can not be executed on all the cores [19]

We show schematic views of variants of singlecore as well as multicore architectures in figure 1.1 and figure 1.2. For simplicity we have shown only one level of on-chip caches in all these schematic views. The on-chip cache shown here is last level cache and is unified in nature i.e. it contains both instructions and data.



(a) Schematic view of a singlecore processor.

(b) Schematic view of a singlecore multithreaded processor.

Figure 1.1: Schematic views of singlecore processors. Here T_0 and T_1 are the hardware threads supported by the single core, which is shown as circle.

In figure 1.1a, we show schematic view of a singlecore processor. In this

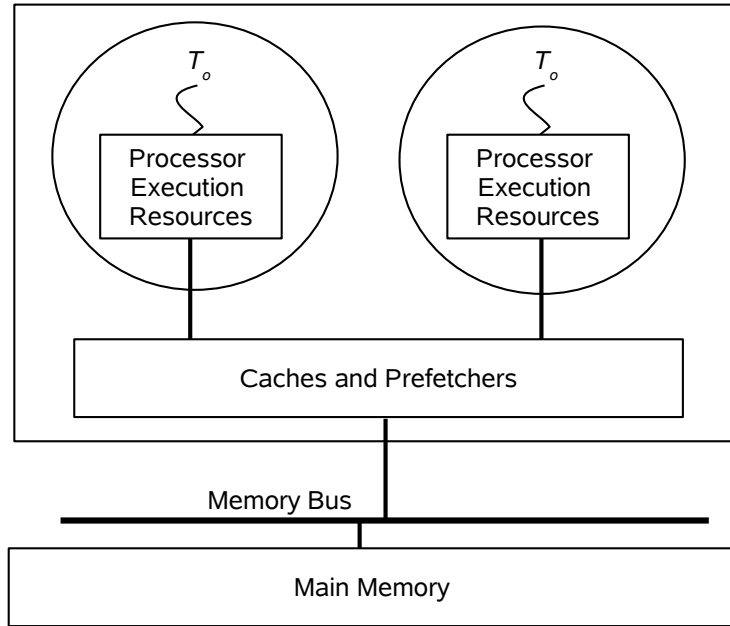
processor there is one core present on the chip, which has the processor execution resources. There is single hardware thread T_0 supported by the single core present on the chip. So there is single processor state maintained for the single hardware thread present on the core. Figure 1.1b shows schematic view of a singlecore multithreaded (MT) processor. In this a single core is present on the chip. The core supports two hardware threads T_0 and T_1 . There are two processor states maintained by the core, one state for each hardware thread. The two hardware threads present on the core share the execution resources by any of the previously mentioned methods used to achieve multi-threading. In this case apart from sharing some of the execution resources, the hardware threads T_0 and T_1 also share the resources present on the memory hierarchy of the processor. The memory hierarchy resources include the on-chip caches, prefetchers and interconnects such as memory bus.

Figure 1.2a shows schematic view of a multi-core processor. The processor chip contains two cores. Each core shown in figure, supports single hardware thread T_0 . In this case hardware thread supported by each of the core has its own private execution resources. Both cores present on the chip share the resources present on the memory hierarchy of the processor i.e. on-chip caches, prefetchers and memory bus. In turn the hardware threads (each supported by one core) also share the same memory hierarchy resources of the processor.

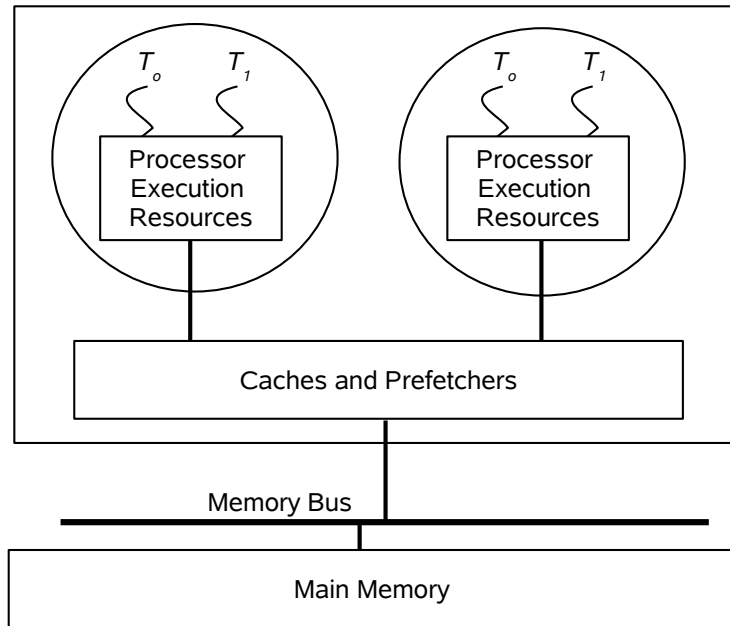
Figure 1.2b shows schematic view of a multicore-multithreaded processor. The processor chip contains two cores. Each core shown in figure, supports two hardware threads T_0 and T_1 . There are two processor states maintained by each of the core, one state for each hardware thread. The two hardware threads present on each of the core share the execution resources of the core by any of the previously mentioned methods used to achieve multi-threading. Both cores present on the chip share the resources present on the memory hierarchy of the processor. In turn all the four hardware threads (two hardware threads supported by each core) also share the same memory hierarchy resources of the processor.

In a nutshell it can be said that following are the significant factors, which led towards development and commercial availability of multicore processors:

- Shift towards thread-level parallelism (TLP) instead of relying only on instruction-level parallelism (ILP) to achieve improvement in processor performance.
- Continual growth in the number of transistors available on the microprocessor chip due to Moore's law[4], so that available transistors could be



(a) Schematic view of multicore processor or chip-multiprocessor (CMP).



(b) Schematic view of multicore-multithreaded or chip-multithreaded (CMT) processor.

Figure 1.2: Schematic views of multicore and multicore-multithreaded processors. Here T_0 and T_1 are the hardware threads supported by the cores, which are shown as circles.

used to provide additional execution cores on the same chip.

- The uncoupling of processor speed from transistor count, due to shift to thread-level parallelism (TLP).
- Rising memory wall i.e. large and growing disparity between processor (CPU) speed and off-chip main memory speeds, in terms of both latency and bandwidth [8] [9] [10].

In fact researchers predicted about multicore being the dominant architecture of future microprocessors due to their better performance per watt characteristics [20], which is also reflected in industry trends [21].

1.1.2 Performance Related Studies

In this section we provide a glimpse of performance related studies associated with architectural evolution, especially with the introduction of multiple hardware threads and cores on a single processor chip (details of the previous related works are given in chapter 2). Such studies include identifying the performance issues with emerging architectures and proposing the possible solutions to various problems encountered. Characterization of systems as well as workloads are one of the essential part of the performance studies [22]. The studies performed for the characterization of systems help in identifying the potential performance bottlenecks. In this way the characterization provides directions for further tuning the systems for better performance. The efforts towards performance improvement involve explorations of design space at the level of microarchitecture as well as software stack [23] [24]. The performance studies also help in creation of the workloads for future.

In year 1996, Olukotun et al. [25] advocated the multicore architecture for a general purpose processor. There have been several research projects to explore multi-core architecture like Hydra [26], Piranha [27] and Atlas [28]. These projects explored various issues such as microarchitectural design, compiler support, and speculative execution of user-level applications. In the past, performance studies have been done on simulators to analyze the operating system behavior in the presence of multiple hardware threads on a single processor [29]. Intel introduced hyperthreading (HT) on Xeon processor [30], which is an implementation of simultaneous multi-threading (SMT). On hyperthreaded Xeon processor, a single core supported two hardware threads. Hyperthreading makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the

two logical processors. Initial performance analysis have been done by Tuck and Tullsen [31] over hyperthreaded Intel Pentium 4 processor [32]. The focus of the study was to understand its performance and the underlying reasons behind that performance. Bulpin and Pratt [33] measured the multiprogramming performance of Intel Pentium 4 processor with hyperthreading and confirmed the findings of previous study done by Tuck and Tullsen [31]. They observed the mutual effect of processes simultaneously executing on the Intel Pentium 4 processor, and found that many performance results can be explained by considering cache miss rates and resource requirement heterogeneity of those processes. The general rule of thumb derived from the study was that threads with high cache miss rates can have a detrimental effect on simultaneously executing threads. In a later work Bulpin and Pratt [34] proposed process scheduling heuristics for hyper-threaded processor, so that pathological combinations of workloads that can give a poor system-throughput could be avoided.

Among studies on multicore processors, Herescu et al. [1] performed performance workload characterization on platform based on IBM POWER5 [35] processor. IBM POWER5 is a dual core processor with simultaneous multi-threading support. Each core on the processor supports two hardware threads. They observed performance impacts due to shared resources, such as caches, translation look-aside buffers (TLBs) and branch prediction hardware. Fedorova et al. [2] also made observation that the sharing of memory hierarchy on the processor such as last level caches among the cores may cause the co-running programs to suffer with performance degradations. In the next section we mention some of the commodity multicore processors, which reflect the trend towards growth in number of cores present on a single chip as well as sharing of processor caches among the cores / threads.

1.1.3 Commodity Multicore Processors

In year 2001, IBM introduced the first chip containing two single-threaded processor cores – the POWER4 [36], for use in general purpose computing. Since that time, several other chip makers have also introduced their multicore solutions for general purpose computing.

We mention some of the commodity multicore processors in table 1.1. The number of cores on the chip are increasing along with generations of the processors. We also see that in majority of the emerging multicore processors the memory hierarchy resources (especially the last level caches L2 or L3) are shared among the cores.

Table 1.1: List of some of the commodity multicore processors.

Year	Multicore Processor	No of cores	On-chip caches	Shared caches	Citations
2001	IBM POWER4	2	L1,L2	L2	[36]
2004	IBM POWER5	2	L1,L2	L2	[35]
2005	Intel Pentium	2	L1,L2	none	[37]
2005	AMD Opteron	2	L1,L2	none	[38]
2005	IBM-Microsoft PowerPC Xenon	3	L1,L2	L2	[39]
2005	Sun UltraSPARC T1 (Niagara1)	8	L1,L2	L2	[40]
2006	Intel Itanium 2	2	L1,L2,L3	none	[41]
2006	Sony-Toshiba-IBM Cell	9	L1,L2	none	[42] [43]
2006	Intel Xeon & Core 2 Extreme	4	L1,L2	L2	[44]
2007	Sun-Fujitsu SPARC64 VI	2	L1,L2	L2	[45]
2007	IBM POWER6	2	L1,L2	none	[46]
2007	AMD Opteron	4	L1,L2,L3	L3	[47]
2007	Sun UltraSPARC T2 (Niagara2)	8	L1,L2	L2	[48]
2008	Intel Xeon	6	L1,L2,L3	L3	[49]
2008	Sun-Fujitsu SPARC64 VII	4	L1,L2	L2	[50]
2009	AMD Opteron	6	L1,L2,L3	L3	[51]
2010	IBM POWER7	8	L1,L2,L3	L3	[52]
2010	Intel Xeon	8	L1,L2,L3	L3	[53]

Multicore processors have introduced sharing of resources among multiple execution cores present on the processor chip. Resources shared among the execution cores include on-chip caches, hardware prefetchers and system bus. In contrast, in the previous generation single-core processors the aforementioned resources were private for the execution core. Previous performance studies, mainly by Bulpin and Pratt [33], Herescu et al. [1] and Fedorova et al. [2] identified processor memory hierarchy resources shared among the cores / threads of the processors as one of the performance critical resource. Sharing of the resources among the cores of the multicore processors causes co-running programs to interfere with each-other. The mutual interference among the co-running programs may cause them to suffer with performance degradations. Thus the multicore architecture poses additional performance issues that need to be addressed for effective utilization of the systems.

At present the multicore processors have become the driving engine for variety of computing systems such as desktops, servers, game consoles as well as embedded systems. The eminent ubiquity of the multicore processors indicates that multicore is going to be the dominant architecture of future. It makes imperative for us to look into performance issues arising due to interactions of

multicore based systems and software. In next section we mention the research objectives and the problem statement for the thesis.

1.2 Research Objectives and Problem Statement

We aim to come up with the models, which can address the performance issues caused due to sharing of memory hierarchy resources among the cores of the multicore processors. The processor memory hierarchy resources include caches, prefetchers and memory bus.

The major objectives of the thesis are to:

- Investigate the performance issues due to sharing of the processor memory hierarchy resources (such as processor caches, prefetchers and bus) among the cores of the multicore processors.
- Review the existing models for memory behavior of programs and associated performance aspects of the multicore processors.
- Propose models for program memory behavior and associated performance aspects of the multicore processors.
- Suggest prospective applications of the proposed models towards solving the performance issues of multicore processors.

The thesis addresses the performance issues of multicore processor based computing systems with focus on program behavior with respect to utilization of the memory hierarchy resources, which are shared among the cores of the processor. The following are the key concerns addressed in the thesis:

- How to develop a model to characterize the program memory behavior on multicore processors (described in section 3.8 of chapter 3 on page 47).
- How to devise a mechanism for application of model developed in previous step so that the interference among co-running programs due to usage of memory hierarchy resources shared among the cores is mitigated (described in section 4.4 of chapter 4 on page 61).
- How to develop a model for predicting the performance degradations caused due to sharing of the processor memory hierarchy resources among the co-running programs (described in section 5.2 and section 5.3 of chapter 5 on page 77 and 78 respectively).

- A prospective application of the model developed for predicting the performance degradations (described in section 5.6 of chapter 5 on page 87).

Overall the work aims at modeling the performance aspects of multicore processor based computing systems and prospective applications of the developed models. It considers use of machine learning techniques for building the models.

1.3 Research Contributions

The domain of machine learning provides various techniques which can be used to build systems for characterizing complex phenomena. Programs simultaneously running on different cores of a multicore processor, have complex interactions due to usage of shared memory hierarchy resources. The thesis presents a unique approach for modeling the performance implications of such interactions by applying machine learning techniques.

The contributions that stem from our research are demonstrated by the publications generated from the thesis. The main contributions from the thesis could be summarized as follows:

- **Methodology to build model to characterize program memory behavior on multicore processors:** We proposed the methodology to build model to characterize program memory behavior on multicore processors. The methodology includes proposal of program attributes, using which the program memory behavior can be predicted in terms of solo-run last level cache stress. It involves use of machine learning techniques to capture the knowledge about processor memory hierarchy resource utilization behavior of running programs. The off-line trained model could be used later for guiding the system policies to mitigate the interference among the co-running programs due to usage of memory hierarchy resources shared among the cores.
- **Meta-scheduler for multicore processors:** We implemented a proof of concept meta-scheduler as an example application of the model developed for program memory behavior characterization. The meta-scheduler runs in user space and guides the process (CPU) scheduling decisions made by underlying operating system process scheduler so that the interference among the programs co-running on multicores is mitigated. We observed performance improvement up to 76% for 4-cores and 54% for 8-cores as compared to default linux kernel process scheduler on our

Intel quad-core Xeon X5482 processor based platform by improving process scheduling. The approximate average cost of the model was about 0.00075% of the total time (i.e. 7-8 cycles per million cycles), as observed in the experiments. The meta-scheduler does not require modifications in process scheduler of the operating system running on the platform.

- **Methodology to build model for performance prediction on multicore processors:** We also proposed the methodology to build model to predict the performance on multicore processors. The model takes the proposed solo-run program attributes as inputs and predicts the concurrent-run performance of the programs on multicores. The concurrent-run involves interference among the program and other programs co-running on cores, which share the memory hierarchy resources with the first program. Such models and techniques could be used for further performance oriented research on multicores as well as simulation of multicores.
- **Application of performance prediction model for simulation of multicore processors:** We also propose the prospective application of the model, which was developed for performance prediction in previous step. We describe the use of the model for simulation of multicores in AKULA tool-set [54], which was recently developed by Zhuravlev et al. for rapid prototyping and evaluation of scheduling algorithms for multicore processors. The use of machine learning based model adds the performance predictability to enable the multicore simulation for workload combinations for which the concurrent-run performance data is not available. The approach proposed in the thesis also supports simulation for processors having variable number of processor cores sharing the resources. For example, the number of cores sharing last level caches are four on Intel Xeon E5630 [55], three on AMD Phenom 8450 [56] and four on AMD Phenom 9650 [56] processors.

The models and the prototype meta-scheduler were developed and demonstrated on the existing commodity multicore processor based systems. It may take some time for some of the previous works done on simulators (Xie and Loh [57], Chandra et al. [58], Rafique et al. [59], Suh et al. [60] and Hsu et al. [61]) to become applicable on real systems. We used machine learning techniques to develop the models by capturing the knowledge about the interactions of the applications and the processor architecture. The focus of studies have been on interactions of programs related with processor memory hierarchy resources, which are shared among the cores present on multicore processors. The use of

machine learning requires training the algorithms to synthesize the models, on the other hand some of the analytical models proposed in previous works by Chandra et al. [58] and Fedorova et al. [2] are fairly involved for application in real systems. The methodologies and mechanisms presented in the thesis do not require specialized support from hardware as proposed in some of the previous works by Xie and Loh [57], Chandra et al. [58], Rafique et al. [59], Suh et al. [60], Hsu et al. [61] and Qureshi and Patt [62]. The work also does not require any modifications or recompilations of the applications. We provide experimental evidence that the developed methodologies using machine learning techniques can be utilized to achieve performance gains. The methodologies developed as part of our work create further research possibilities. Fellow researchers can use these methodologies as initial point, for using, improving and extending the ideas.

1.4 Organization of the Thesis

The thesis is organized into six chapters as mentioned below. Though the chapters are related with each other, we tried to present each chapter in self-contained manner to the extent possible, to ease the sequential reading of the thesis document. Thesis organization is also shown in figure 1.3.

- **Chapter 1: Introduction.** The chapter gives an overview of evolution of processor architecture and associated performance studies, which provided the motivation behind the work carried out in the thesis. It also mentions the research objectives of the work.
- **Chapter 2: Review of Performance Studies and Related Background.** This chapter provides the review of the previous studies related with performance issues of multicore based computing systems, with specific emphasis on sharing of processor memory hierarchy resources among the cores present on the processors. It also gives an overview of machine learning and some of its applications in computer systems research.
- **Chapter 3: Characterization of Program Memory Behavior.** This chapter presents the methodology for characterizing the program memory behavior on multicore processors. The chapter also provides brief description of machine learning algorithms as well as experimental platforms used in the study. It describes the program attributes and the experimental method to gather the data from a multicore processor based platform to generate the training data-set. It also describes the

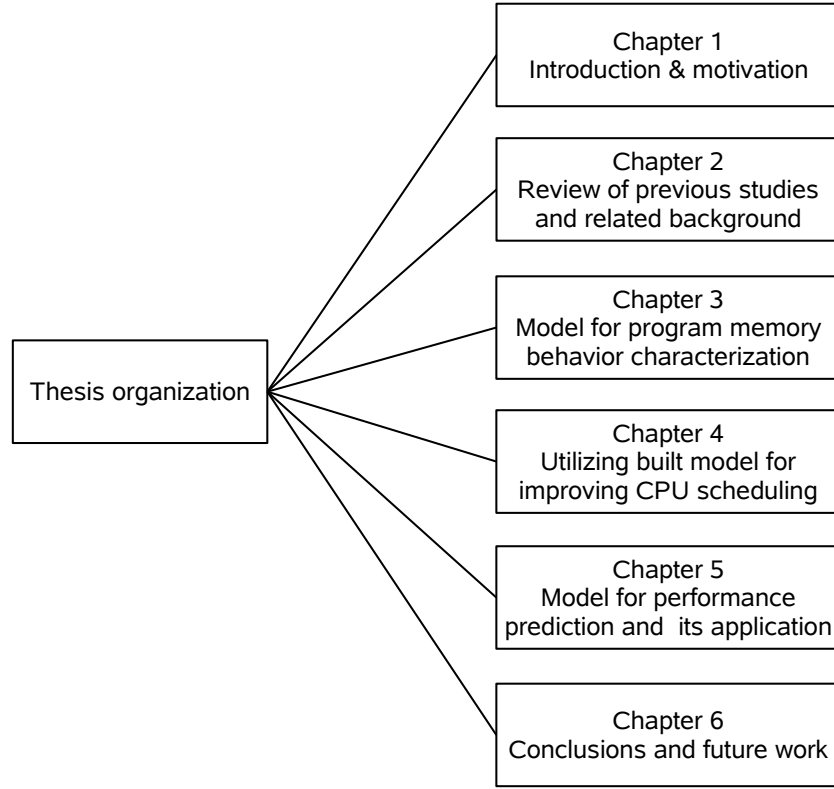


Figure 1.3: Thesis organization.

prediction accuracy results of the trained model and its transferability over other experimental multicore platform.

- **Chapter 4: Improving Process Scheduling.** This chapter describes a meta-scheduler as an example application of the model built for program memory behavior characterization. It also provides the results on improvement in performance, observed with the meta-scheduler along with details of experimental setup.
- **Chapter 5: Performance Prediction.** This chapter describes the methodology to build model for performance prediction on multicore processors along with experimental setup and results. It includes the proposed solo-run program attributes, which the model takes as inputs to predict the concurrent-run performance. It also describes the prospective application of the model (built to predict concurrent-run performance) for simulation of multicores.
- **Chapter 6: Conclusions and Future Work.** This chapter summarizes the major contributions of the work and mentions the observations. It also highlights the future research directions.

Overall the thesis covers the modeling of the performance aspects of multi-core processors with focus on sharing of processor memory hierarchy resources among the cores present on the multicore processors. The models were built by training the machine learning algorithms. The applicative aspects of the research reported in the thesis are reflected in the prospective use of the developed models for – (i) improving process scheduling on multicores and (ii) simulation of multicores.

Chapter 2

Review of Performance Studies and Related Background

This chapter presents review of relevant literature on performance studies accompanied with shift towards multicore architecture. The chapter is organized as follows: In section 2.1, we begin with the description of the studies related with shared memory hierarchy resources present on multicore processors. The work done in this thesis uses machine learning techniques to synthesize the models. In section 2.2, we provide an overview of machine learning and its applications in computer systems research. Section 2.3 describes the work related to performance prediction studies on multicore processors followed by conclusions.

2.1 Shared Memory Hierarchy Related Studies

This section describes previous works related with management of shared caches on multithreaded and multicore processors. The various solutions proposed by researchers to manage the interference among co-running programs using shared caches on multicore and multithreaded processors mostly fall into two kinds of solutions:

- Hardware based solutions
- Software based solutions

Both solutions in general require knowledge about the memory behavior especially the shared last level (e.g. level-2) cache related characteristics of the programs running on the processor. Hardware based solutions propose extra

support from the processor hardware. Software based solutions include scheduling by operating systems and adaptations in memory management subsystem, which involve page coloring based techniques.

2.1.1 Hardware Based Solutions

Among the hardware based solutions, Suh et al. [60] proposed memory monitoring scheme, which utilized a set of novel hardware counters. The counters provide the marginal gain in the cache hits as the size of the cache is increased. The scheme described in the work uses the counters, to get an accurate estimate of the isolated miss-rates of each process as a function of cache size under the standard LRU (Least Recently Used) replacement policy. Such information can be used to schedule jobs or to partition the cache to minimize the overall miss-rate. Unlike this work, our work does not require any new hardware counters and relies on the counters already available on existing commodity multicore processors.

In a later work Suh et al. [63] proposed a dynamic cache partitioning method for minimizing the overall miss rate and improving IPC (Instructions Per Cycle). The scheme uses a set of on-line counters to estimate gain or loss to each process in terms of the number of cache misses observed with different cache allocations. It changes the cache allocation so that more needy processes can get more cache space. This method requires a new cache replacement policy in place of the LRU replacement policy currently used in systems. The work was done on simulator. It requires changes in the hardware.

Kim et al. [64] studied fairness in cache sharing between threads in a chip multiprocessor (CMP) architecture. They evaluated cache fairness metrics for their correlation with the execution-time fairness. Execution-time fairness is defined as how uniform the execution times of co-scheduled threads are changed, where each change is relative to the execution time of the same thread for its solo-run. The work also proposed L2 (level-2) cache partitioning algorithms to be implemented in hardware to optimize fairness. They observed 4× improvement in fairness and 15% increase in the throughput (combined instructions per cycle) with fair caching algorithms, compared to a non-partitioned shared cache.

Chandra et al. [58] studied the impact of L2 cache sharing among the co-running threads on a chip multiprocessor architecture. They proposed performance models to predict the impact of cache sharing on co-scheduled threads. The input to the models is the isolated L2 cache stack distance or circular

sequence profile of each thread. Stack distance profile of an application is a compact summary of its cache-line reuse patterns. The models give an estimate of the number of additional L2 misses caused by sharing, compared to solo-run of the thread (i.e. without sharing the L2 cache with other co-runner). The study used a cycle-accurate simulator for a dual-core CMP architecture. The proposed models are fairly involved for implementation in hardware.

Hsu et al. [61] discussed various cache policies for chip multiprocessors. Cache policies were named according to the target to be achieved, like Communist cache policies for equal performance target and Utilitarian cache policies for overall performance target and the most common current model of a free-for-all cache as Capitalist policy. They used analytical models and behavioral cache simulation and observed that thread-aware cache resource allocation mechanism is required for CMPs.

Qureshi and Patt [62] proposed a utility based cache partitioning scheme. In this scheme the share of the cache received by an application is proportional to the utility rather than its demand. The scheme uses Utility Monitor (UMON) a special kind of counters implemented in hardware. We discuss about the performance prediction part of this work in section 2.3.

Rafique et al. [59] proposed architectural support for chip multiprocessors that enables operating system (OS) level cache management. The proposed scheme consists of three components: a hardware cache quota enforcement mechanism, an OS interface and a set of OS-level policies for changing the quotas. The hardware mechanism enforces OS-specified, cache quotas in shared lower level caches for each sharing entity. The OS can provide various cache management policies by manipulating the quotas via the quota specification interface. Thus the proposed hardware based cache quota system, can be used by operating system to use different policies for different applications in order to improve the overall performance in chip multiprocessors.

Chang and Sohi [65] proposed Cooperative Cache Partitioning (CCP) to allocate cache resources among concurrently running threads on chip multiprocessors. The proposed scheme uses multiple time-sharing partitions. They integrated the proposed cache partitioning scheme with cooperative caching [66] for chip multiprocessors. The work was done on simulator for a 4-core chip multiprocessor. Cooperative caching [66] tries to reduce the number of off-chip accesses by combining the strengths of private and shared caches adaptively. It requires additional modifications in existing cache replacement policy and coherence protocol.

Zhao et al. [67] investigated mechanisms for fine-grain monitoring of the use

of shared cache resources on chip-multiprocessors. They proposed the cache monitoring architecture named CacheScouts, consisting of tagging (software guided monitoring IDs), and sampling mechanisms (set sampling) to achieve shared cache monitoring on per application basis. The study was performed on a cache hierarchy simulator called CASPER [68]. They also mentioned about the use of CacheScouts [67] in operating systems and virtual machine monitors for – characterizing execution profiles, optimizing scheduling for performance management, providing quality of service (QoS) and metering for chargeback. The proposed scheme needs to be implemented in hardware.

Xie and Loh [57] proposed a new classification algorithm for determining the personalities of the programs with respect to their cache sharing behavior. The proposed algorithm needs to be implemented in hardware, to help the partitioning of the cache between the running programs to reduce the shared cache interference caused by co-running programs on multicore processors. The main focus of the work was to help in partitioning the caches between running programs, though the results can also be utilized for scheduling the programs. This work was performed on simulator for dual-core processor.

Srikantaiah et al. [69] proposed an operating system directed integrated processor-cache partitioning for chip multiprocessors. The scheme partitions both the available processors and the shared cache in a chip multiprocessor among running applications. The scheme uses a regression based model to predict the behavior of applications to find the most suitable processor and L2 cache partitions. In the proposed scheme cache partitioning is done by allocating one cache partition to each processor-set (processor partition), thereby encouraging constructive sharing among threads of the same application and alleviating the impact of interference among different applications in the cache. The proposed partitioning approach is iterative, that involves partitioning of processors and cache in a series of iterations. The processor partitioning performed in one iteration influences the cache partitions in the same iteration and the cache partition at the end of the current iteration influences the processor partitioning in the following iteration. They used Simics full system simulator [70] in the study. The scheme involves changes in both hardware as well as operating systems.

The solutions requiring modifications in the processor hardware may need time to get implemented and available in commodity processors. The approach taken in our work does not require any changes in cache replacement policies or any other components of the existing commodity hardware.

2.1.2 Software Based Solutions

Among the software based solutions, Bulpin and Pratt [34] proposed process scheduling heuristics for hyperthreaded Pentium 4 processor to avoid the pathological combinations of workloads, which can give a poor system throughput. The hyperthreads introduced by hyperthreading technology of Intel are abstracted by the hardware as logical processors. They observed that the existing operating systems process scheduling does not take account of the particular resource requirements of the individual threads, which can cause the sub-optimal schedules to take place. In their work multiple linear regression was used to model the speed-up ratio of a program in solo-run as compared to paired-run with co-running sibling thread. The off-line trained model was used to change the dynamic priority of a process in linux process scheduler. It was done so that a runnable process could be given a higher dynamic priority if it is likely to perform well with the process currently running on the other logical processor. In this scheme the kernel need to keep a record of the estimated system-speedups of pairs of processes. Their work demonstrated the scheduling heuristics using the standard linux-2.4 scheduler – a single-queue dynamic priority based scheduler where priority is calculated for each runnable task at each rescheduling point. In the next version, linux-2.6 introduced changes in the process scheduler which maintains a run queue per processor. The independence of scheduling between the processors complicates coordination of pairs of tasks. The investigation of the application of the proposed heuristics for linux-2.6 was part of their future work.

Fedorova et al. [2] proposed an L2-cache conscious scheduling algorithm for efficient utilization of the shared last level (L2) cache on multithreaded chip multiprocessors. Their OS scheduling algorithm is based on the balance-set principle proposed by Denning [71]. Balance-set scheduling involves scheduling the runnable threads into subsets or groups, such that the combined working set of each group fits in the cache. By making sure that the working set of each scheduled group fits in the cache, the proposed algorithm reduces cache misses. The work involves use of estimated cache miss ratios for each group of threads as metric for making scheduling decisions. The model [72] used in the work for estimating cache miss ratios is based on cache model for single-threaded workloads developed by Berg and Hagersten [73]. The cache miss ratios of multithreaded workloads estimated by model were within 17% of the actual values, on average. Estimation of cache miss ratios by the Berg-Hagersten model requires monitoring of memory re-use patterns of the threads. Implementation

of such monitoring requires to capture a sample of memory locations that a thread references and then to record how often those locations are reused. This approach is expensive for use in a real system, because it needs handling frequent processor traps. Fedorova et al. performed the study on simulator for chip multithreaded processor using benchmarks from the SPEC cpu2000 suite [74]. They observed that the L2-cache conscious scheduling algorithm had the potential to reduce the L2 cache miss ratios by 25-37%, thereby yielding a performance improvement of 27-45%.

In a later work Fedorova et al. [75] proposed operating system scheduling algorithm to improve performance isolation on chip multiprocessors. Poor performance isolation refers to variability in performance of an application due to the behavior of other applications co-running with it. This performance dependency is caused because of unfair, corunner-dependent cache allocation on chip multiprocessors. The cache-fair algorithm proposed in the work ensures that the application runs as quickly as it would under fair cache allocation, regardless of how the cache is actually allocated. If a thread executes fewer instructions per cycle (IPC) than it would under fair cache allocation, the scheduler increases CPU timeslice of that thread. This way, overall performance of that thread does not suffer because it is allowed to use the CPU longer. The work includes proposal of a heuristic cache model to determine the fair IPC (i.e. the IPC under fair cache allocation) in the scheduler. The model for fair IPC is comprised of two parts – estimation of the fair cache miss rate, and then estimation of the fair IPC for the given fair miss rate. Here fair cache miss rate is the miss rate experienced by the thread when it is allocated its fair cache share. Model building involves running a thread with several different corunners, to derive a relationship between the miss rates of that thread and its co-runners, and later on using that relationship to estimate fair miss rate of that thread. The model was proposed for two co-running threads sharing the cache and was validated on a simulated dual-core chip multiprocessor. However the applicability of the model needs to be checked for the cases where more than two cores / threads share the cache.

Knauerhase et al. [76] proposed observation mechanisms in the operating systems for multicore systems so that performance degradations due to usage of resources shared among the cores could be avoided. The policies proposed in the work use information about the behavior of the processes to alter OS scheduler decisions so that interference due to use of resources shared by the cores is mitigated. The proposal includes policies to affect process migration decisions of the operating system so that cache loads remains approximately equal across

last level caches. The work also investigated mechanisms to improve fairness in cache usage. They also proposed policy of observation-based migration among functionally asymmetric cores to handle cases in which cores provide different features. The proposed mechanisms were implemented inside operating systems. The work done on linux kernel version 2.6.20 involved changes in $O(1)$ scheduler. The mechanisms need re-implementations for the Completely Fair Scheduler (CFS) [77] present in current linux versions, as CFS has been the default scheduler of linux since version 2.6.23. The overall speedup observed in the study was up to 6% with SPEC cpu2006 applications [78] as workload.

Banikazemi M. et al. [79] also proposed a scheduling scheme to mitigate the performance bottlenecks caused by sharing of resources such as caches and main memory bandwidth in multicore systems. The meta-scheduler proposed in their work, uses algebraic cache model to predict the impact of new schedules. The model works iteratively to find new schedule. They observed that the overall system performance can be improved by as much as 14%. In a recent work Zhuravlev et al. [80] used running average of cache miss rate to make scheduling decisions. The work discusses about the performance bottlenecks caused by contention for last level caches as well as other resources on processor memory hierarchy such as prefetcher, memory bus and memory controller; which are shared among the cores of the multicore processors. The work also presents performance prediction technique using proposed “pain” metrics, which we mention in section 2.3 on performance prediction works.

Jiang et al. [81] studied the optimal co-scheduling problem on chip multiprocessors (CMPs). They analyzed the complexity of the problem and proved that it is NP-complete when the number of cores sharing the caches is greater than two. They also presented a polynomial-time algorithm for finding the optimal co-schedules on dual-core CMPs. The proposed algorithm first constructs a degradation graph, and then treats the optimal scheduling problem as a minimum-weight perfect matching problem. It solves the problem using *blossom* algorithm [82]. Based on the first algorithm, the work also proposed approximation algorithms for more complex CMP systems. In their work they assumed that all co-run performance is given, which may not be possible for a scheduler running on a real-world setup. In their study they also ignored difference between execution times of the co-running programs, as well as phase changes and rescheduling.

In a later study Tian et al. [83] showed that relaxing the constraints, particularly the assumptions on job lengths and reschedulings in previous study by Jiang et al. [81], increases the complexity of finding the optimal schedules

significantly. They developed an A*-search [84] based algorithm to determine optimal co-schedules for small problems. They also proposed approximation algorithms for co-scheduling large problems. The works by Jiang et al. [81] and Tian et al. [83] show that even if the exact performance for every possible schedule is known it is still a challenging task (NP-complete in the general case) to find the optimal schedule.

Scheduling as a solution to management of resources shared by the co-running threads on single-core multithreaded systems has been studied in the past by Snavely et al. [85] and Parekh et al. [86]. Both of these works have been done on simulators. The scheduling algorithms for single-core multithreaded systems discussed in these works, sample the space of possible thread schedules by randomly perturbing the set of threads that are scheduled together, collect hardware performance counter data, and using heuristics, determine which of the sampled schedules would perform best. These algorithms were shown to work reasonably well on single-core multithreaded systems, giving an average improvement in throughput of 9% over a random thread schedule (17% over the worst-case schedule). This technique could be applied on workloads with a handful of threads. If the number of threads is large, sampling the space of potential thread mixes may become less productive, because the size of the sample space grows exponentially with the number of threads.

Recent study by Mars et al. [87] proposed a Contention Aware Execution Runtime (CAER) environment to minimize cross-core interference due to usage of shared resources. CAER uses hardware performance counters present in multicore processors to infer and respond to contention. CAER is composed of a runtime on which all applications of interest run. The runtime classifies these applications into the latency-sensitive and batch categories. CAER probes the hardware performance monitoring unit (PMU) to collect information about the applications hosted by it. Information about the applications running on CAER is continually collected and analyzed throughout their lifetime. CAER uses heuristics to detect the contention. After detection of the contention, CAER dynamically adapts the batch applications to minimize the contention. The prototype mentioned in the work performed adaptations by throttling down the execution of the batch applications to reduce the pressure on the contended resource. The proposed CAER needs to be statically linked with the application binary.

Cho and Jin [88] proposed a software-based mechanism for L2 cache partitioning based on physical page allocation. The work was done on simulator that does not take the interference of the operating system into account. In another

software-based L2 cache partitioning scheme, Tam et al. [89] implemented a software mechanism in the operating system for partitioning of the shared L2 cache by guiding the allocation of physical pages. The L2 cache is partitioned among the running applications using page coloring. It involves reserving a portion of the cache space for each application, and allocating physical memory such that it should map to the reserved portion of the cache for an application. The size of the portion of the L2 cache space to allocate is determined with the help of miss rate curves (MRCs). The miss rate curves were used as a metric to predict performance as a function of L2 cache size. In this work, they assumed that per application L2 miss rate curves are available to the operating system as they are obtained during profiling runs and stored in a repository. In order to add a new application to the repository, these curves must be calculated by running the application (or at least a representative portion of it) several times. Berg and Hagersten [90] calculated miss rate curves on-line with the runtime overhead of 40%, by using a software approach based on data address watchpoints. In a later work Tam et al. [91] proposed a software-based on-line method to characterize the cache requirements of processes on IBM POWER5 processor, using data from hardware performance counters. They used memory access trace of running programs for getting L2 cache miss rate curves, which can be used for partitioning the cache. The authors used continuous data address sampling, a performance monitoring unit (PMU) feature available on IBM POWER5 processor.

Zhang et al. [92] proposed improvements for page coloring based cache partitioning schemes. They observed that the page coloring causes additional constraints on allocation of memory, which may conflict with memory needs of the application. Due to imposition of page color restrictions on an application, only a portion of the memory can be allocated to it. When the system runs out of pages of a certain color, the application may come under memory pressure while there still may be sufficient memory available in other colors. It makes necessary for the application to either evict some of its own pages to secondary storage or steal pages from other page colors. The former can cause slowdown due to swapping of pages while the latter may cause performance deterioration to other applications due to cache conflicts. Page coloring also involve high overhead of on-line recoloring to adapt cache partitioning policies in a multi-programmed execution environment. Recoloring a page involve memory copying that takes several microseconds on commodity systems. Frequent recoloring of a large number of application pages may incur excessive overhead that negates the benefit of page coloring. The work by Zhang et al. [92] proposed a

hot-page coloring approach that requires enforcement of cache mapping colors on a small set of frequently accessed (or hot) pages for each process. It also mentions an approach for tracking application page hotness on-the-fly, which involves periodic scan of page table entries. The proposed hot-page coloring is aimed to reduce memory allocation constraint and on-line recoloring overhead of all-page coloring in an adaptive and dynamic environment.

The software-based cache partitioning approaches require non-trivial modifications in virtual memory sub-system, which itself is a complex component of the operating systems. The schemes also may require copying of the physical memory, if the portion of the cache allocated to an application need to be reduced or reallocated.

2.1.3 Our Approach for Program Memory Behavior Characterization

Our proposal involves using machine learning techniques for characterizing memory behavior of programs running on multicores. Solo-run last level cache stress (described in section 3.3 on page 38), is the metric we use to describe the memory behavior of the programs. Our work is done on existing commodity hardware platforms without additional specialized hardware support. The model built by using machine learning techniques could be used by a system management entity to appropriately frame the system policies. As an example application of the built model; we developed a proof of concept meta-scheduler (mentioned in chapter 4). The meta-scheduler uses the trained model to guide operating system CPU scheduler to improve the process schedule on multicore processors. Our work does not involve changes to the hardware, the operating system process scheduler or the application. In the next section we give an overview of machine learning techniques and their applications in previous related works.

2.2 Overview of Machine Learning Techniques

Machine learning [93] is concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data. It involves use of computational methods for improving performance by mechanizing the acquisition of knowledge from experience. Machine learning techniques focus on automatically learning to recognize complex patterns and make intelligent decisions based on data. The core objective of a learner is to generalize

from its experience. A learner tries to capture characteristics of interest from the data given to it. The data serves as examples that illustrate relations between observed variables. The training examples come from some generally unknown probability distribution and the learner has to extract from them something more general, something about that distribution, that allows it to produce useful answers in new cases.

Models of system behaviors are useful for prediction, diagnosis, and optimization in self-managing systems. Machine learning approaches have an important role in model building because they can infer system models automatically from instrumentation data collected as the system operates.

Machine learning algorithms could be organized into a taxonomy [94], as follows:

- Supervised learning is inferring a function from supervised (labeled) training data. It generates a function that maps inputs to desired outputs. The training data consist of a set of training examples. Each example is a pair consisting of an input object (typically a vector) and a desired output value. A supervised learning algorithm analyzes the training data and produces an inferred function, which is called a classifier (if the output is discrete) or a regression function (if the output is continuous).
- Unsupervised learning refers to finding hidden structure in unlabeled data. The examples given to the learner are unlabeled, hence there is no error or reward signal to evaluate a potential solution. Unsupervised learning models a set of inputs, like clustering.
- Semi-supervised learning combines both labeled and unlabeled examples to generate an appropriate function or classifier. As an example, co-training is a machine learning algorithm, which is used when there are small amounts of labeled data and large amounts of unlabeled data. Co-training is used in text mining for search engines.
- Reinforcement learning, learns how to act given an observation of the world. Every action has some impact on the environment, and the environment provides feedback in the form of rewards that guides the learning algorithm. Reinforcement learning is concerned with taking actions in an environment so as to maximize some notion of cumulative reward. Reinforcement learning differs from supervised learning in that correct input/output pairs are never presented. There is a focus on on-line per-

formance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

- Transduction is reasoning from observed, specific training cases to specific test cases without constructing a model. In contrast, induction is reasoning from observed training cases to general rules (model), which are then applied to the test cases. Transduction tries to predict new outputs based on training inputs, training outputs, and test inputs.
- Inductive transfer or transfer learning is an approach that focuses on storing the knowledge gained while solving one problem and applying it to a different but related problem. For example, the knowledge gained while learning to recognize cars could be applied when recognizing buses or trucks.

Computer systems researchers have been using machine learning techniques to attack problems in real-world computer systems. The domain of problems includes reliability and performance issues in large-scale systems and networks, power efficiency in sensor networks and self-configuration in complicated systems. The motivation for application of machine learning techniques is mainly building empirical models. Machine learning techniques help the researchers to cope with the challenges of scale and complexity of current and future systems. Machine learning techniques have also been used in code generation and optimization. Some researchers have also applied machine learning to microarchitecture research.

Machine learning techniques such as neural networks have been used by Yoo et al. [95] for workload characterization. The focus of the study is different from our focus i.e. characterizing the memory (especially last level cache) related behavior of the programs. Their study characterizes a 3-tier web service in terms of functional characteristics of the application.

Kishore and Negi used machine learning to characterize the workload [96] and improve process scheduling in linux operating system [97]. Their work is not focused on memory behavior of the workload. The study uses various attributes from ELF (Execution & Linking Format) executables and the previous execution history of the processes to characterize the workload.

Ould-Ahmed-Vall et al. [98], used model trees in performance analysis. The focus of the study is not on characterizing the program memory behavior. In this study, the generated model tree is used to gain insights into bottlenecks affecting processor performance. The generated model from machine learning is used for post-facto kind of analysis as opposed to on-line use.

In the context of multi-core processors, machine learning techniques have been used by Ganapathi et al. [99] for optimizing the performance. The work uses machine learning for exploration of the auto-tuning parameter space, for compilers on multicore.

Fedorova et al. proposed to use reinforcement learning to improve operating system scheduling policies for a heterogeneous multicore system [100]. The focus of our study has been on homogeneous multicores.

İpek et al. [101] used artificial neural networks to cull high-performing configurations from very large design spaces for both single-core and multicore microarchitectures.

Bitirgen et al. [102] proposed implementation of artificial neural networks based framework in hardware to coordinate the management of multiple interacting resources in chip multiprocessors. In the subsequent work Martinez et al. [103] reported use of machine learning techniques for resource management on multicore architecture. The work proposed self-optimizing on-chip hardware agents to optimize the resource allocation in multicore processors.

In our work supervised machine learning is used to derive the model to predict solo-run last level cache stress of programs running on multicores, while sharing last level cache with other co-running programs. In addition to this we also used supervised machine learning to build the model to predict performance on multicore processors. In the next section we describe the previous work related with performance prediction on multicores.

2.3 Performance Prediction

The performance prediction on multicores involves development of techniques and methods to understand and predict the program behavior with respect to utilization of resources shared among the cores (e.g. shared last level cache) and associated performance implications. The program behavior can manifest in two kinds of performance implications, which arise due to use of resources shared among the cores:

- The extent by which a program suffers with performance degradation due to interference of its co-runners.
- The extent by which a program can degrade performance of its co-runners.

2.3.1 Previous Studies on Performance Prediction

Previous studies on program behavior resulted in classification schemes for understanding the performance as well as workload creation. These studies include works by Moreto et al. [104], Lin et al. [105] and Qureshi and Patt [62].

Moreto et al. [104] explained the speedups, which can be achieved by cache partitioning. The metrics used in their study to classify the programs, requires comparison against the performance when using full L2 cache, which in turn requires a complete redundant execution of the programs.

Lin et al. [105] studied the cache partitioning using operating system level page coloring. As part of their work they classified programs into four classes by considering the performance degradation observed when running a program using only a 1 MB L2 cache compared to the baseline configuration with 4 MB cache. Their classification scheme does not involve prediction task. It is a post-facto kind of scheme for creation of workloads.

Qureshi and Patt [62] studied the problem of partitioning a shared cache between multiple concurrently executing applications. They observed that a higher demand for cache resources does not always correlate with a higher performance from additional cache resource. They proposed utility-based cache partitioning (UCP), which uses Utility Monitor (UMON) a special kind of counters implemented in hardware.

Xie and Loh [57] proposed classification scheme for determining the personalities of the programs with respect to their cache sharing behaviors. The scheme classifies the programs into four animal personalities based on a few simple heuristic metrics. The animal personalities are: Turtle, Sheep, Rabbit and Tasmanian Devil. Turtles are applications that do not make much use of the shared last-level (L2) cache. Sheeps are applications which are not very sensitive to space allocated to them in shared last-level (L2) cache, hence are not easily perturbed by other co-running applications. Rabbits are applications which are sensitive to space allocated to them in shared last-level (L2) cache. Tasmanian Devils are applications, which tend to negatively impact other co-running applications. The study was performed on simulator and the scheme needs to be implemented in hardware.

Zhuravlev et al. [80] performed “Pain classification” of programs based on cache sensitivity and intensity. By combining the sensitivity and intensity of two applications “pain” estimate of a co-schedule is made. Sensitivity of a program was obtained from its stack distance profile, obtaining which by binary instrumentation using Pin tool [106] [107] is very time consuming. Pin

[106] is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. Dynamic binary instrumentation involves performing instrumentation at run time on the compiled binary files.

Xu et al. [108] proposed shared cache aware performance model for multicore processors. It estimates the performance degradation due to cache contention of processes running on multicores. Their model uses reuse distance diagram of the program. Their method to obtain the reuse distance diagram involves multiple runs of the given program with a synthetic benchmark called stressmark. Our work is closest to this work, where we estimate (predict) the concurrent-run performance of programs running on multicores. We do not use reuse distance diagram, obtaining which generally is a costly operation.

Among other performance prediction related works, Hoste et al. [109] proposed an approach for predicting the performance of given application on a number of platforms, to determine which platform yields the best performance. The scheme involves measuring a number of microarchitecture-independent characteristics for the given application, and subsequently relating those characteristics to the characteristics of the programs from a previously profiled benchmark suite. Performance prediction of given application is made based on the similarity of the given application with programs in the benchmark suite. The focus of the study is different from our work, which is focused on the performance implications due to sharing of memory hierarchy resources among the cores of the multicore processors.

2.3.2 Our Approach for Performance Prediction

The approach taken in this thesis for performance prediction uses machine learning to build the model. We use solo-run program attributes to predict the concurrent-run performance. The concurrent-run involves sharing of resources with other programs simultaneously running on other cores of the processor. The solo-run attributes are calculated from data collected from hardware performance counters present on commodity multicore processors.

The methodology developed in our work does not require specialized hardware support as required in some of the previous studies by Xie and Loh [57], and Qureshi and Patt [62]. It does not require time-consuming program run under binary instrumentation as mentioned by Zhuravlev et al. [80], that uses Pin [106] based tools [107] for collecting the information about program behavior in the form of stack distance profile for performance prediction. The efforts involved in the process of training machine learning algorithm to build

the model are amortized because a trained model could be used for performance prediction afterwards.

We also proposed a prospective application of the model developed for performance prediction, to improve simulation of multicores in AKULA [54] tool-set (mentioned in chapter 5 at page 87). AKULA is a recently developed tool-set that provides a platform for rapid prototyping and evaluation of thread scheduling algorithms on multicore processors. In AKULA [54], a *bootstrap* module works on the basis of previously collected performance data of programs, to simulate program execution on multicores. Our approach augments such a *bootstrap* module with the model (for performance prediction) built using machine learning techniques. The use of the offline-trained model extends the ability of *bootstrap* module to predict degradation in performance (due to sharing of resources) where previous performance data is not available for pairing/co-scheduling of applications. Also the approach proposed in the thesis allows greater scalability for variable number of processor cores sharing the resources.

2.4 Conclusions

This chapter provided an overview of the research in connection with some of the performance issues observed on multicore processors. The research spawns activities on both fronts viz. hardware as well as software. The hardware based solutions need time to become available on commodity multicore processor based systems. We are investigating the software based techniques, so that the policies could be adapted to mitigate the performance bottlenecks. We mentioned the approach taken in our work, and contrasted it with the previous works.

Chapter 3

Characterization of Program Memory Behavior

This chapter describes the methodology to develop model to characterize the memory behavior of the programs on multicore processors. The approaches to mitigate the interference of co-running programs require knowledge about the memory behavior of those programs. We proposed use of machine learning to help the acquisition of the knowledge about program memory behavior in the form of trained model. This knowledge can be used by a system management entity to frame and adapt the policies to alleviate the performance bottlenecks. The machine learning algorithms were trained using program performance data collected from hardware performance counters of the processors.

The chapter is organized as follows: Section 3.1 provides a view of the performance issues along-with performance results observed by us on commodity multicore processor. Section 3.2 provides information about the program memory behavior. Section 3.3 describes the metric used by us for its characterization. In section 3.4, we discuss the issues related with applicability of the methodology on available commodity platforms. Section 3.5 provides the information on hardware performance counters. Section 3.6 provides brief description of the machine learning algorithms used in the work. In section 3.7, we provide details of the experimental setup. In section 3.8, experimental method to generate the data to train the machine learning algorithms is described. Section 3.9 provides the results on prediction accuracy of the machine learning algorithms used in the study. In section 3.10, we discuss about assessing the transferability of the trained models followed by the conclusions at the end.

3.1 Performance Issues on Multicores

This section describes certain performance issues due to sharing of memory hierarchy resources along-with performance results observed on one of our experimental platforms. Programs simultaneously running on different cores, which share the memory hierarchy resources may interfere with each other. For example, programs running on the cores sharing the last level cache (e.g. L2 cache), may compete for space in the last level cache, thereby may cause eviction of cache lines allocated to each other. Hence it will cause those programs to experience increased cache misses, as compared to their solo-run (the solo-run does not involve sharing of the resources). Because of this mutual interference the applications running on multicore processors suffer from performance degradation.

Table 3.1 mentions solo-run and concurrent-run performance (in terms of cycles per instruction i.e. CPI) of some of the SPEC cpu2006 benchmark programs [74] [78], running on Intel quad-core Xeon X5482 processor based experimental platform (details of the platform are described in section 3.7.1 on page 45). Please note that on Intel Xeon X5482 processor, two processor cores share the resources such as last level (L2) cache, hardware prefetch unit and Front Side Bus (FSB) [55].

Table 3.1: Solo-run and concurrent-run performance of SPEC cpu2006 programs on Intel Xeon X5482 processor.

Program Names	Performance in terms of CPI, for			
	Solo-run	Concurrent-run, when co-runner is		
		429.mcf	433.milc	459.GemsFDTD
429.mcf	5.89	8.37	10.15	10.04
433.milc	2.41	2.89	3.46	3.54
459.GemsFDTD	1.62	1.98	2.21	2.39
462.libquantum	1.67	2.24	3.05	3.06
410.bwaves	1.17	1.36	1.45	1.53

The performance results mentioned in table 3.1 are also shown in figure 3.1a, figure 3.1b and figure 3.1c. We can see that the performance of the programs observed during the concurrent (paired) run is degraded as compared to their solo-run (more the cycles per instruction, more is the degradation in performance). During the concurrent run the program shares the resources such as last level (L2) cache, prefetcher and Front Side Bus (FSB) with the program co-running on other core of Intel Xeon X5482 processor.

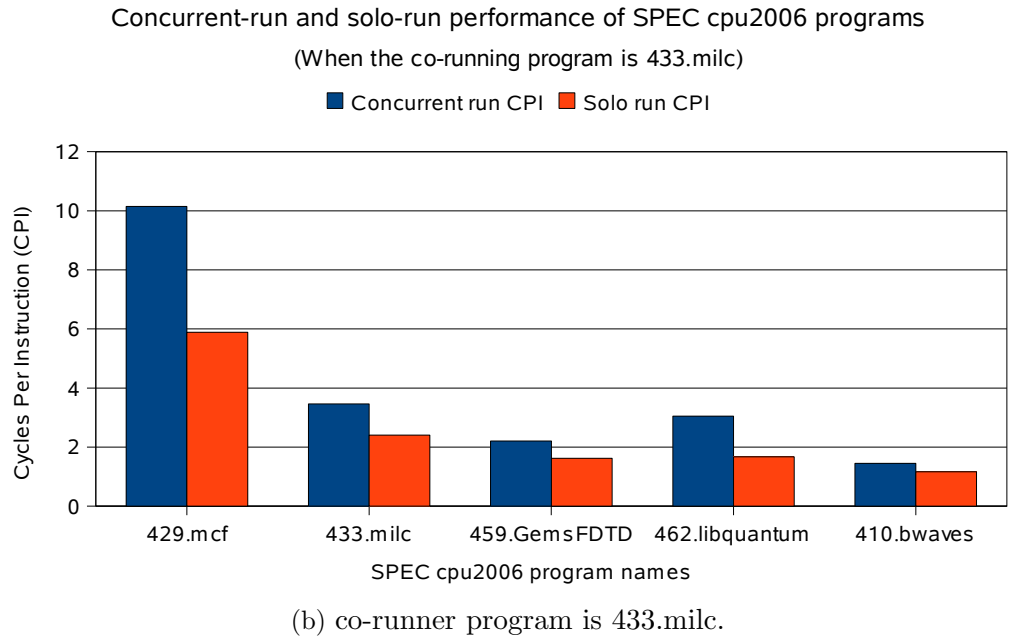
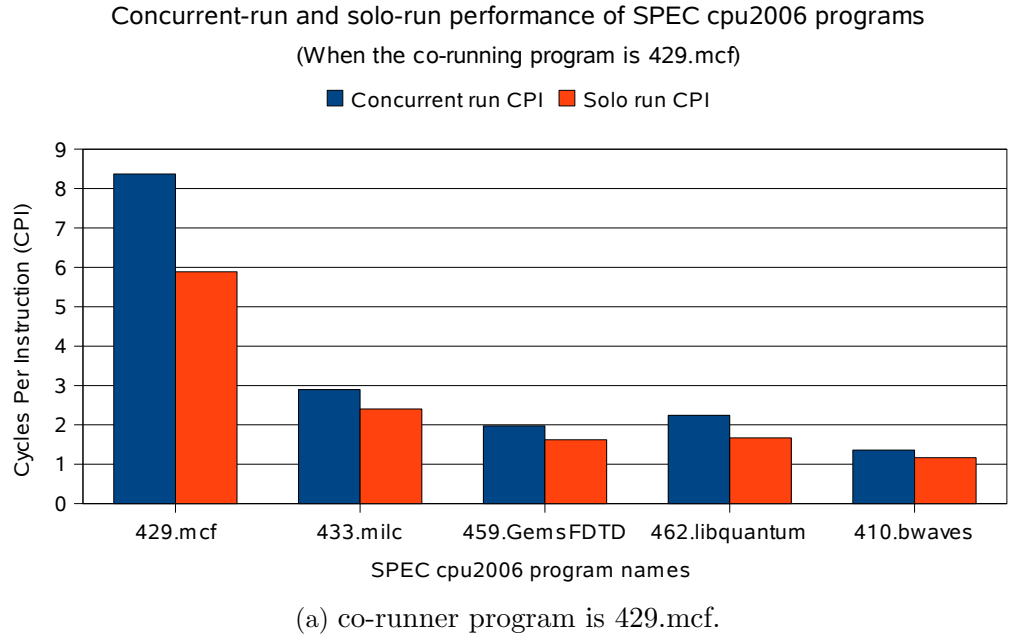
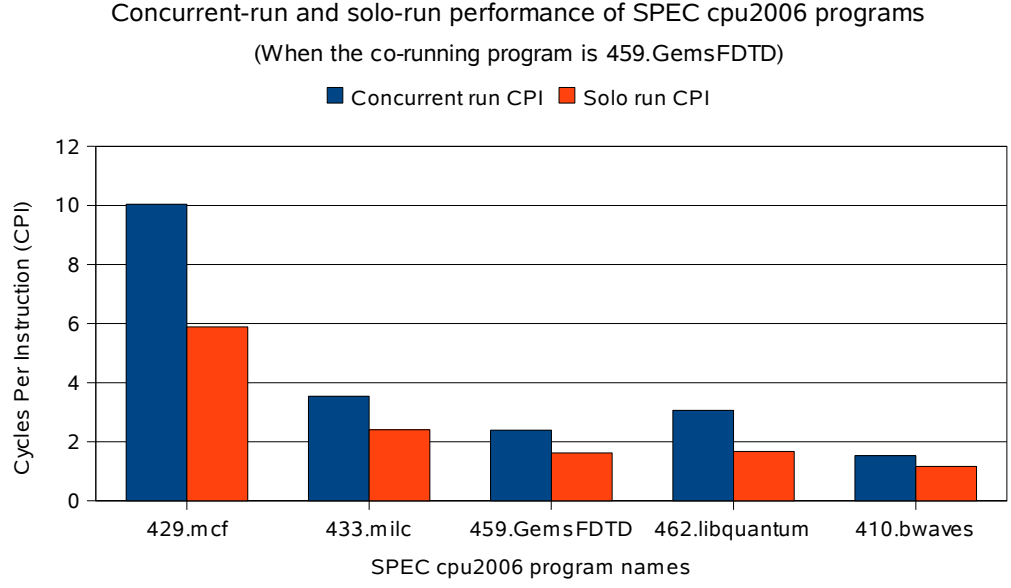


Figure 3.1: Concurrent-run and solo-run performance (in terms of CPI) of some of the SPEC cpu2006 benchmarks on Intel quad-core Xeon X5482 processor. The performance of the programs (program names mentioned along with x-axis) observed during the concurrent (paired) run is degraded as compared to their solo-run. More the cycles per instruction, more is the degradation in performance. The name of the program, which was paired with the programs shown along with x-axis is mentioned at the top as well as bottom. (part (a) and (b))



(c) co-runner program is 459.GemsFDTD.

Figure 3.1: Concurrent-run and solo-run performance (in terms of CPI) of some of the SPEC cpu2006 benchmarks on Intel quad-core Xeon X5482 processor. (part (c))

The data on percentage degradation in performance experienced by the programs during their concurrent-run are mentioned in table 3.2. The percentage degradation in performance is calculated as shown in equation 3.1:

$$\% \text{ degradation in performance} = \frac{(\text{concurrent_run_CPI} - \text{Solo_run_CPI}) \times 100}{\text{Solo_run_CPI}} \quad (3.1)$$

Table 3.2: Percentage degradation in performance of some of the SPEC cpu2006 programs, observed during concurrent-run as compared to their solo-run on Intel Xeon X5482 processor.

Program Names	Percentage degradation in performance during concurrent-run, when co-runner is		
	429.mcf	433.milc	459.GemsFDTD
429.mcf	42.13	72.34	70.49
433.milc	20.33	43.83	47.07
459.GemsFDTD	21.87	35.92	47.07
462.libquantum	34.41	82.55	83.39
410.bwaves	16.64	24.36	31.22

The data on performance degradation (mentioned in table 3.2) are also plotted in figure 3.2. In this figure vertical(y)-axis shows percentage degradation in

performance for SPEC cpu2006 programs mentioned along with horizontal(x)-axis.

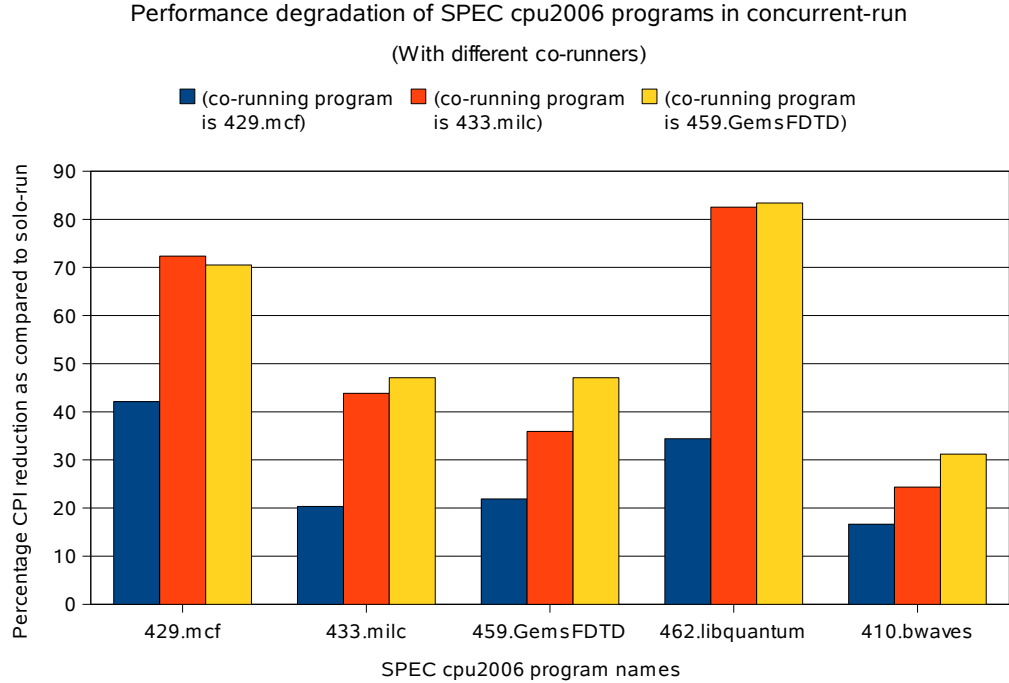


Figure 3.2: Percentage degradation in performance of some of the SPEC cpu2006 programs, observed during concurrent-run as compared to their solo-run on Intel Xeon X5482 processor. The vertical(y)-axis shows percentage degradation in performance of the SPEC cpu2006 programs. Each of the three bars for a program name mentioned along with horizontal(x)-axis, corresponds to a different co-running program, which was paired with the first program. Co-running program names are mentioned at the top.

The observations from the data on performance degradation are:

- Same program suffers with different amounts of performance degradation when run with different co-running programs
- The quantum of performance degradation differs for different pairs of co-running programs

The quantum of performance degradation for a program during concurrent-run depends on:

- own memory behavior
- co-running programs memory behavior

Hence a strategy to mitigate performance degradation would require the knowledge about the characteristics of the programs with respect to their utilization of the shared memory hierarchy resources. In the next section we discuss about the memory behavior of the programs.

3.2 Program Memory Behavior

We use term “program memory behavior” for behavior of the programs with respect to utilization of processor memory hierarchy resources. The processor memory hierarchy resources include the caches, which are provided to obviate the frequent accesses to main memory. Processors are provided with multiple levels of caches to reduce the cache miss penalty [7]. Our focus of study has been on the last level cache, which is shared among the cores present on multicore processors. The penalty of getting increased misses (which are caused due to interference among co-running programs sharing the caches) at the last level shared caches is highest as compared with any other level caches, as the missed cache line is to be fetched from the memory, which consume lots of processor cycles.

Different programs have varied requirements of space in the last level cache. The requirement of space in last level cache for a given program changes along with the progress of the program during runtime, according to the phase behavior of the program. For example, a program may have high requirements of space in the last level cache in one part of it and in the next part this requirement may be low; based on the logic of the program. The duration of the program, that has reasonably stable behavior (say high cache space requirement) is called one phase of the program [110]. One program may pass via many different phases during its complete run (example given in next section with figure 3.5 on page 41). Thus the memory behavior of a program varies throughout its execution. In the next section we discuss about the metric used in the work for characterizing the program memory behavior.

3.3 Metric to Characterize Program Memory Behavior

We used solo-run last level cache stress as a metric to characterize program memory behavior. The solo-run last level cache stress is the total number of last level cache lines brought in due to miss and prefetch activities per kilo (10^3)

instructions retired, when the program is running without sharing the last level cache with programs co-running on other cores.

The last level cache stress for programs co-running on cores sharing the memory hierarchy resources is different from their solo-run last level cache stress. This is so because the last level cache stress observed during concurrent-run results from the interactions of the individual cache access patterns and working sets of the co-running programs based on their locality.

The values of the solo-run as well as concurrent-run last level cache (LLC) stress of 429.mcf (one of the SPEC cpu2006 benchmark program) measured on our Intel quad-core Xeon X5482 processor based experimental platform are shown in figure 3.3. Concurrent-run last level cache (LLC) stress is the total

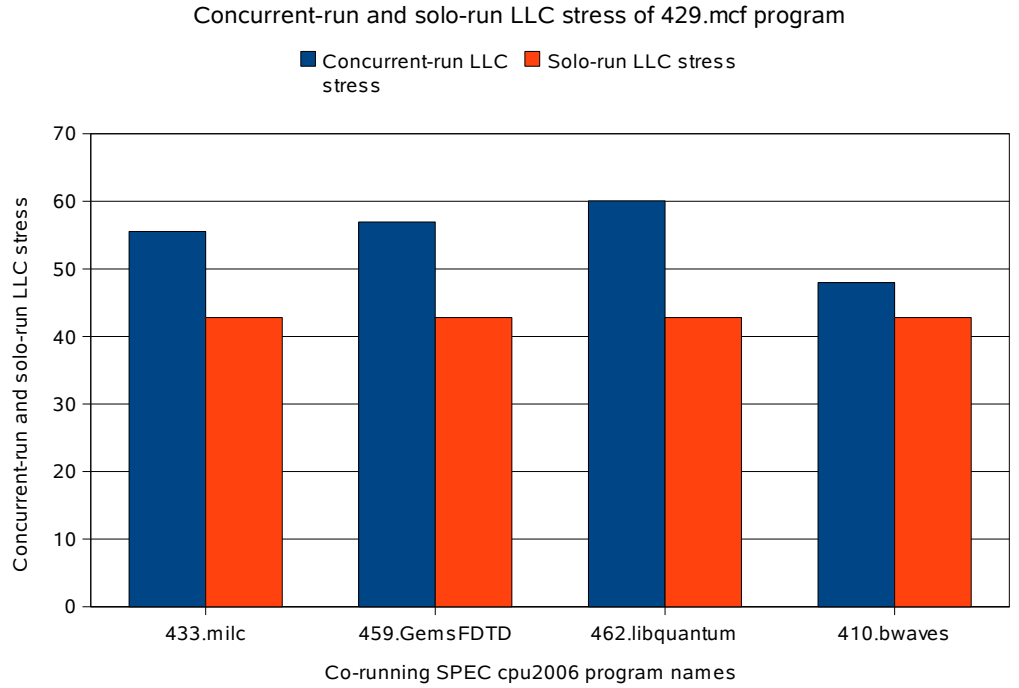


Figure 3.3: Concurrent-run and solo-run last level cache (LLC) stress of SPEC cpu2006 benchmark program 429.mcf on Intel Xeon X5482 processor. The names of co-running SPEC cpu2006 benchmark programs are shown along with horizontal(x)-axis. Values of concurrent-run and solo-run last level cache (LLC) stress are shown along with vertical(y)-axis. The values of concurrent-run last level cache stress differs for various runs based on the co-running benchmark program.

number of last level cache lines brought in due to miss and prefetch activities per kilo (10^3) instructions retired for a program, when that program is running on a core sharing the last level cache with other program co-running on other core of the multicore processor. It can be seen that the values of concurrent-

run last level cache stress of the program 429.mcf differs for various runs based on the co-running benchmark programs. It is so because the concurrent-run last level cache stress results from the interactions of the memory behavior of co-running programs sharing the last level cache.

The value of solo-run last level cache stress includes the total number of last level cache lines brought in due to miss as well as prefetch activities. It represents the amount of traffic happening between memory and last level cache of the processor. Thus solo-run last level cache stress indicates the stress put by a program on shared memory hierarchy resources such as last level cache, hardware prefetching unit and Front Side Bus (FSB). We measured the “solo-run last level cache (LLC) miss per kilo instructions retired” as well as “solo-run last level cache (LLC) miss and prefetch per kilo instructions retired” (i.e. solo-run last level cache stress) for programs of SPEC cpu2006 suite [74] [78] on Intel quad-core Xeon X5482 processor based experimental platform. The values of the same for some of the SPEC cpu2006 benchmarks are shown in figure 3.4. The large difference between the two indicates better suitability of solo-run

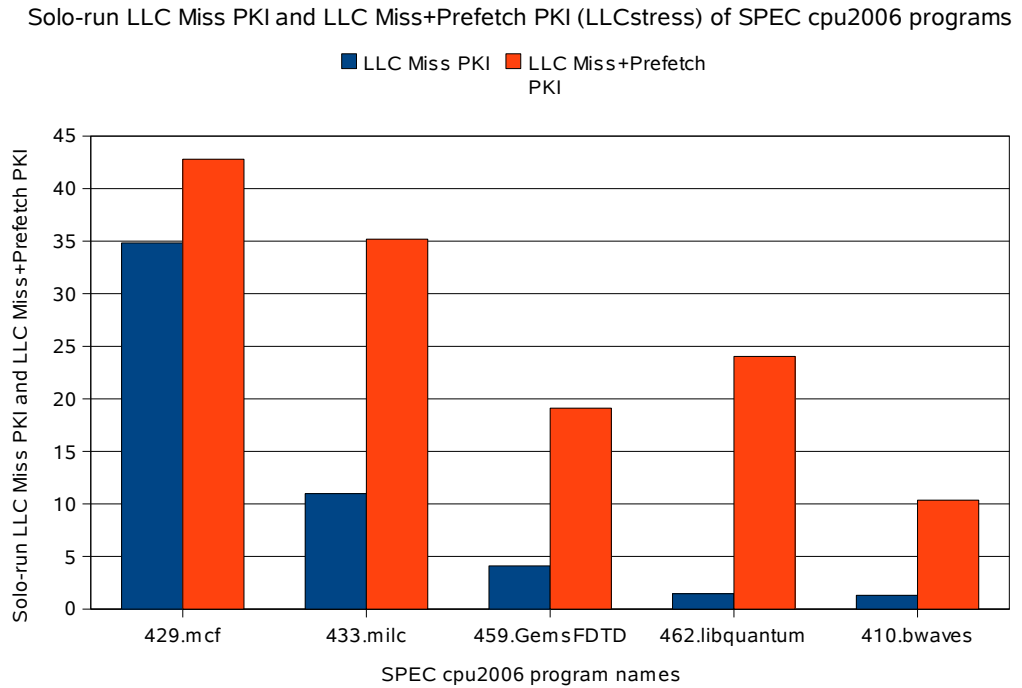


Figure 3.4: Values on vertical(y)-axis are “solo-run last level cache (LLC) miss per kilo instructions retired” and “solo-run last level cache miss and prefetch per kilo instructions retired” (i.e. solo-run last level cache stress) for SPEC cpu2006 benchmark programs on Intel Xeon X5482 processor. Horizontal(x)-axis shows names of the programs. The solo-run last level cache stress has much larger values as it represents the total traffic between last level cache and memory.

last level cache miss and prefetch rate (i.e. solo-run last level cache stress) as a metric to characterize the program memory behavior, as it represents the total traffic between last level cache and memory.

The values of solo-run last level cache stress, sampled during the complete run of one of the SPEC cpu2006 benchmark program 433.milc are shown in figure 3.5 to show phase-wise variation in memory behavior. In this figure the values of solo-run last level cache stress (for the interval of every 100 million completed instructions) are plotted for complete run of the program. The values of solo-run last level cache stress follows a pattern in periodic manner during the complete run of the program. This indicates phase-to-phase variations in resource requirements of the same program during its complete run. Different programs have different phase behaviors. We chose behavior of pro-

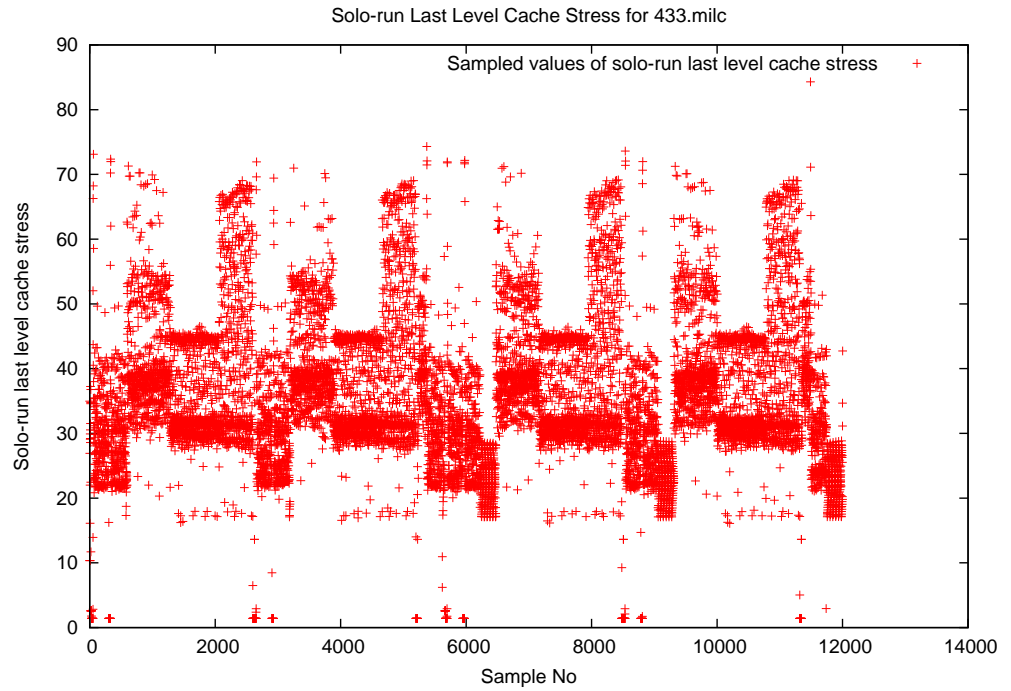


Figure 3.5: Phase behavior of SPEC cpu2006 benchmark program 433.milc on Intel Xeon X5482 processor. Vertical(y)-axis shows solo-run last level cache stress, sampled for every 100 million instructions during the complete run of the SPEC cpu2006 benchmark program 433.milc. Horizontal(x)-axis shows the progress of the program in terms of number of samples, where each sample refers to completion of 100 million instructions of the program. The values of solo-run last level cache stress follows a pattern in periodic manner during the complete run of the program.

gram 433.milc (it is a quantum chromodynamics application [74] [78]) for this figure, as this program has periodicity in phase behavior that can very easily be seen from the plot shown in figure 3.5.

In the next section we mention the issues related with applicability of the methodology for program memory behavior characterization on available commodity platforms.

3.4 Program Memory Behavior Characterization Issues

The program memory behavior is a complex phenomenon. It varies throughout the execution of a program according to its phase behavior. The concurrent-run program memory behavior observed at run-time contains the impact of mutual interactions of the phase behaviors of co-running programs.

A system management entity needs to have information about the memory behavior of the programs throughout their complete run. Such information enables it to appropriately frame the policies to manage the co-runner interference due to use of memory hierarchy resources shared among the cores. For example, an operating system scheduler can use the solo-run last level cache stress of the programs as a metric to intelligently schedule them to reduce the co-runner interference, which we describe in the next chapter.

There are some additional issues for usability of the methodology for program memory behavior characterization on existing commodity multicore processor based platforms. The program memory behavior characterization should not require:

- Modification in source code of an existing application
- Recompile of an existing application
- Any changes in the processor hardware

We used machine learning to derive the model to predict the memory behavior (in terms of solo-run last level cache stress) of the programs running on multicore processors. The machine learning algorithms were trained using the data collected from the hardware performance counters of the processors on experimental platforms. The use of hardware performance counters does not require modification or recompilation of the profiled software. The next section gives overview of the hardware performance counters.

3.5 Hardware Performance Counters

Modern processors have hardware performance monitoring unit (PMU) built into them. The performance monitoring unit provides special registers called hardware performance counters (HPCs). These hardware performance counters can count various performance events occurring on the processor such as number of instructions retired, L2 cache misses etc. The tracking of the various events by hardware performance counters does not cause any slowdown of the operating system kernel or applications. The counts for the performance events can be measured for programs running on the processors by reading the hardware performance counters. These registers can be configured to measure various events of interest. Details regarding the number of performance counters and the way to configure them are described in processor manuals [55]. There have also been recommendations given by the researchers from Berkeley to utilize the information from hardware performance counters to aid the resource management decisions and optimizations within software stack running on the systems [111]. The next section provides brief description of the machine learning algorithms used in the work.

3.6 Machine Learning Algorithms Used

We used supervised learning [93], in which the machine learning algorithms generate regression models by the process of training. These generated models can be used later on for predicting the variable of interest in a real world scenario. The process of regression consists of fitting a model that relates a dependent variable y to a set of independent variables x_1, x_2, \dots, x_n expressed in the form of equation shown below:

$$y = f(x_1, x_2, \dots, x_n) \quad (3.2)$$

The term “attributes” is also used to refer to the independent variables that are used to predict the dependent variable of interest (also referred as “class variable”).

Different machine learning algorithms correspond to different concept description spaces searched with different biases. Some problems are served well by different description languages and biases, while others are not served well or even served badly. This entails the study of various machine learning algorithms belonging to different families to check their efficacy to solve a given problem across various scenarios [112].

Hence we used various machine learning algorithms from weka-3.6.2 machine learning workbench [112] [113]. In this work we used default settings of the parameters to the machine learning algorithms in weka-3.6.2, as the default settings work well for variety of problems [114]. The machine learning algorithms used in the study are: Linear Regression (LR), Artificial Neural Networks (ANN), Model Trees (M5'), K-nearest neighbors classifier (IBK), KStar (K*) and Support Vector Machines (SVM).

Linear regression algorithm [112] performs least-squares linear regression.

Artificial neural networks [93] are based on the mechanism of co-operative processing of information, as done by neurons in the brain. In a multilayer neural network, there is an input layer, an output layer and a number of hidden layers. Each layer has a number of neurons (nodes) organized in it. The input layer takes the information to be processed as input. The first hidden layer takes the results from input layer as its inputs and forwards its results as inputs to the next layer. The output layer takes the results of the last hidden layer as inputs and produces the prediction result. In this work back-propagation was used to train feed-forward multilayer neural network.

K-nearest neighbors classifier (IBK) and KStar (K*) are lazy or instance-based learning algorithms [93] [115] [116]. In this case a new regression equation is fitted each time, when the model needs to predict on a new instance (i.e. a new query point).

Model trees are a kind of regression tree [117]. We chose M5' algorithm [118], which is an improved version of original M5 algorithm invented by Quinlan [119]. Model trees recursively partition the input space until the linear models at the leaf nodes can explain the remaining variability in those partitions.

Support Vector Machines (SVM) [120] try to find instances that are at the boundary of the classes. These instances are called support vectors. Then they generate functions that discriminate those vectors as widely as possible. For training the support vector machine, a generalization of Sequential Minimal Optimization (SMO) algorithm by Shevade et al. [121] was used. The next section describes the experimental setup.

3.7 Experimental Setup

This section describes the platforms used to perform the experiments for gathering the performance counter data. We also provide brief on the workload used in the experiments.

3.7.1 Experimental Platforms

We performed experiments on two platforms. The first platform is a dual-socket DELL Precision T7400 workstation with two Intel quad-core Xeon X5482 processors (3.2 GHz) and 32 GB of memory. The specifications of the DELL Precision T7400 workstation are described in table 3.3.

Table 3.3: Specifications of DELL Precision T7400 workstation used for generating data to build regression models.

Item	Specification
Number of chips	2
Number of cores	4 per chip
Number of hardware threads	1 per core
CPU cores	Intel Xeon X5482, 3.2 GHz
L1 Instruction Cache	32KB 8-way set-associative per core
L1 Data Cache	32KB 8-way set-associative per core
L2 Unified Cache	Total 12MB, Each 6MB 24-way set-associative L2 cache block is shared by two cores
RAM	32 GB

The second platform is a single socket DELL Precision 390 workstation with one Intel dual-core Core2 6300 processor (1.86 GHz) and 2 GB of memory. Table 3.4 describes the specifications of DELL Precision 390 workstation.

Table 3.4: Specifications of DELL Precision 390 workstation used for generating data to test regression models.

Item	Specification
Number of chips	1
Number of cores	2 per chip
Number of hardware threads	1 per core
CPU cores	Intel Core2 6300, 1.86 GHz
L1 Instruction Cache	32KB 8-way set-associative per core
L1 Data Cache	32KB 8-way set-associative per core
L2 Unified Cache	2MB 8-way set-associative shared by two cores
RAM	2 GB

The operating system kernel on both experimental platforms was linux-2.6.30. For collecting data from the hardware performance counters [55] of

the processors present on the experimental platforms, we used perfmon2 [122] interface.

3.7.2 Processor Memory Hierarchy on Experimental Platforms

Both Intel quad-core Xeon X5482 and Intel dual-core Core2 6300 processors have two levels of caches. On both of the processors, there is separate level-1 (L1) instruction and level-1 (L1) data cache, each is 8-way set-associative and is of size 32KB per core. The level-2 (L2) cache on both processors, is unified in nature, i.e. it contains both instructions and data.

The Intel quad-core Xeon X5482 processor has level-2 (L2) cache of size 12MB, which is organized in two blocks. Each 6MB 24-way set-associative L2 cache block is shared by two processor cores.

The Intel dual-core Core2 6300 processor has level-2 (L2) cache of size 2MB, which is 8-way set-associative. It is shared by both of the processor cores.

The sharing of last level caches between cores of Intel Xeon X5482 as well as Intel Core2 6300 processors is shown in figure 3.6 and figure 3.7 respectively.

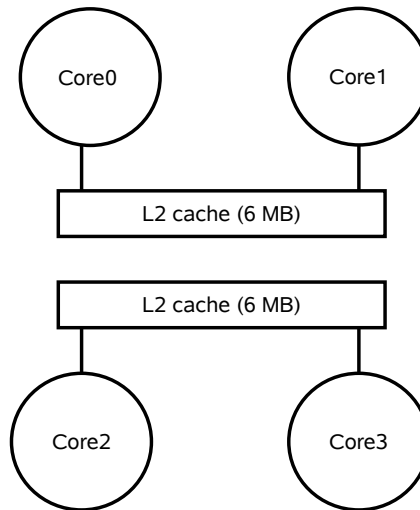


Figure 3.6: Last level (L2) cache sharing on Intel quad-core Xeon X5482 processor. There are two L2 cache blocks, each of size 6MB. Each of the L2 cache block is shared by two processor cores.

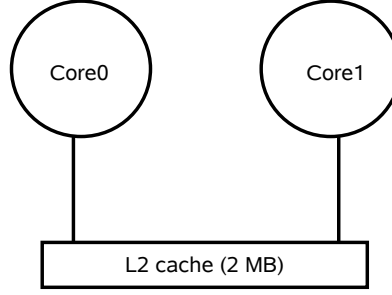


Figure 3.7: Last level (L2) cache sharing on dual-core Intel Core2 6300 processor. The L2 cache size is 2MB. It is shared by both cores.

3.7.3 Workload

We used benchmark programs from SPEC cpu2006 benchmark suite [74] [78]. The benchmark suit consists of 12 integer programs and 17 floating point programs. We used reference inputs for the benchmark programs. With reference inputs the total workload consists of 35 integer programs and 20 floating point programs i.e. total 55 programs. The next section describes the methodology to generate the data for training the machine learning algorithms.

3.8 Methodology

We performed solo-run and concurrent-run experiments on the two platforms. Data collected from the Intel Xeon X5482 processor based platform, were used to train the machine learning algorithms to build the regression models. Data collected from the Intel Core2 6300 processor based platform, were used to assess the transferability of the built regression models. The methodology followed to build the model by training the machine learning algorithms is shown in figure 3.8.

3.8.1 Solo-run Experiment

We ran each program from SPEC cpu2006 benchmark suite on a core of the processor and disallowed scheduling programs on other core that shares last level cache with the previous core. The class variable solo-run last level cache stress was calculated from hardware performance counter data collected for complete run of each benchmark program.

Methodology followed to build models using machine learning

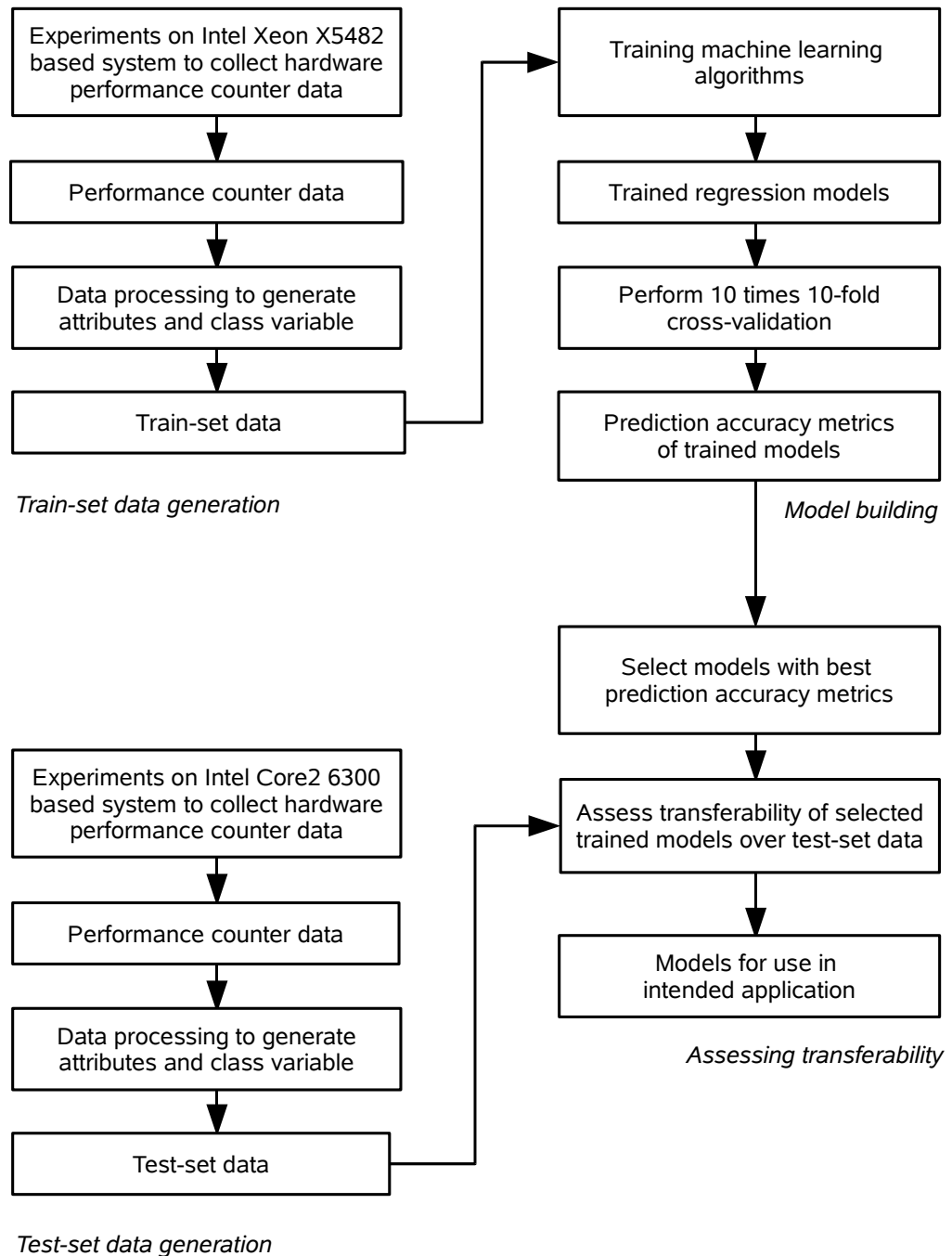


Figure 3.8: Methodology followed to build the model by training the machine learning algorithms.

3.8.2 Concurrent-run Experiment

In this experiment, programs from SPEC cpu2006 benchmark suite were run concurrently on the processor cores sharing last level cache. Event data from the hardware performance counters of the processor were collected for each program. These data were used to compute the program attributes. Each of these programs takes different amount of time for completion. Hence we allowed one of the co-scheduled programs to run till completion. The other program was run in an infinite loop to execute repeatedly and was stopped only as the first program finished. The workload with 55 programs, generates 55×55 (i.e. 3025) pairs of programs to run on the system.

3.8.3 Generating Program Attributes and Class Variable

This section describes about computation of the program attributes and class variable. Please note that we use term “class variable” to denote the variable to be predicted later and it contains a numeric value.

The Intel Xeon X5482 and Core2 6300 processors support hundreds of microarchitectural performance events, which can be measured for the programs running on the system. These events are mentioned in processor manuals [55]. We chose the performance events, which are specific to the processor memory hierarchy of the system.

The event data collected from the hardware performance counters of Intel Xeon X5482 processor are mentioned as follows (same applies to Intel Core2 6300 processor also, data collected from which were used to to assess the transferability of the built regression models):

- *INSTRUCTIONS_RETIRED*: It measures the number of completed instructions.
- *LAST_LEVEL_CACHE_REFERENCES*: It measures the number of references to last level (L2) cache.
- *L2_LINES_IN_SELF_ANY*: It measures the total number of last level (L2) cache lines brought in as a result of encountering last level (L2) cache misses as well as prefetching activities.

Consider two programs $p1$ and $p2$ are simultaneously running on two cores of Intel Xeon X5482 processor, and sharing the last level (L2) cache during the concurrent-run experiment. Here $p1$ is the program that ran to completion and

program $p2$ was run in an infinite loop till completion of $p1$. The attributes and class variables are computed from the hardware performance counter data collected for programs $p1$ and $p2$ in the following manner (prefix $p1_$ and $p2_$ refer to values collected for program $p1$ and $p2$ respectively):

The **first attribute** ($p1_L2_REF_PKI$) is last level (L2) cache references per kilo instructions retired. It gives information about the extent of last level (L2) cache usage of the program $p1$.

$$p1_L2_REF_PKI = \frac{(p1_LAST_LEVEL_CACHE_REFERENCES \times 1000)}{p1_INSTRUCTIONS_RETIRED} \quad (3.3)$$

The **second attribute** ($p1_L2_IN_PKI$) is last level (L2) cache lines brought in (due to miss and prefetch) per kilo instructions retired. It is indicative of the stress put by program $p1$ on last level (L2) cache.

$$p1_L2_IN_PKI = \frac{(p1_L2_LINES_IN_SELF_ANY \times 1000)}{p1_INSTRUCTIONS_RETIRED} \quad (3.4)$$

The **third attribute** ($p1_L2_IN_PK_REF$) is last level (L2) cache lines brought in (due to miss and prefetch) per kilo last level (L2) cache lines referenced. It gives an indication of re-referencing characteristic of the program $p1$. Lower value of this attribute indicates higher tendency of the program $p1$ to re-reference the previously referenced last level (L2) cache lines.

$$p1_L2_IN_PK_REF = \frac{(p1_L2_LINES_IN_SELF_ANY \times 1000)}{p1_LAST_LEVEL_CACHE_REFERENCES} \quad (3.5)$$

The **fourth attribute** ($p1_L2_FO$) is the fractional last level (L2) cache occupancy of the program $p1$. It gives a rough estimate of fraction of space occupied by the program $p1$ in last level (L2) cache, while sharing it with other co-running program $p2$.

$$p1_L2_FO = \frac{p1_L2_LINES_IN_SELF_ANY}{(p1_L2_LINES_IN_SELF_ANY + p2_L2_LINES_IN_SELF_ANY)} \quad (3.6)$$

The **class variable** to be predicted is “solo-run last level cache stress”. It is similar as second attribute i.e. last level (L2) cache lines brought in (due to miss and prefetch) per kilo instructions retired, but for solo-run of the same program $p1$. We represent it by $s1_L2_IN_PKI$ (here $s1_$ denotes solo-run of first program $p1$ out of pair $p1$ and $p2$).

All the four attributes mentioned above were calculated from the hardware performance counter data collected in concurrent-run experiment.

The class variable was calculated from the hardware performance counter data collected in solo-run experiment.

An instance of training data could be represented as a five-tuple comprising of four program attributes and class variable, as shown below:

$$(p1_L2_REF_PKI, p1_L2_IN_PKI, p1_L2_IN_PK_REF, p1_L2_FO, s1_L2_IN_PKI) \quad (3.7)$$

A sample of train-set data of program memory behavior generated from Intel Xeon X5482 processor is shown in table 3.5.

Table 3.5: Sample of train-set data of program memory behavior from Intel Xeon X5482 processor. First four columns in the table are program attributes and last column lists class variable “solo-run last level cache stress”.

<i>p1_L2_REF_PKI</i>	<i>p1_L2_IN_PKI</i>	<i>p1_L2_IN_PK_REF</i>	<i>p1_L2_FO</i>	<i>s1_L2_IN_PKI</i>
9.118	1.026	112.500	0.198	0.629
19.786	11.339	573.048	0.500	9.763
17.506	4.259	243.272	0.377	3.296
17.279	8.219	475.658	0.489	7.771
16.216	8.551	527.291	0.514	8.092
18.028	9.605	532.804	0.522	9.153
18.902	9.707	513.531	0.520	9.176
21.602	15.362	711.116	0.549	14.805
17.740	10.431	588.010	0.517	10.092
17.456	2.471	141.527	0.294	1.562
136.079	50.925	374.231	0.594	42.801
17.070	0.229	13.422	0.064	0.135
15.357	0.807	52.575	0.126	0.506
13.708	0.680	49.570	0.110	0.422
13.239	2.349	177.430	0.296	2.176
15.150	0.933	61.590	0.142	0.700
12.331	0.581	47.131	0.098	0.393
5.656	1.131	199.947	0.207	0.176
3.290	0.561	170.644	0.104	0.402

The next section describes the prediction accuracy results of various machine learning algorithms used in the study, for predicting the solo-run last level cache stress.

3.9 Results on Prediction Accuracy for Predicting Solo-run Last Level Cache Stress

We used 10-fold cross validation [123] to evaluate the prediction accuracy of the different machine learning algorithms. This technique involves dividing the overall data in ten disjoint subsets. Each algorithm is trained using nine of the subsets and then evaluated using the tenth subset. This process is repeated ten times and each time, a different subset is used for testing and the remaining nine subsets are used for training. The algorithm is evaluated by averaging the prediction accuracy metrics from ten different models built in the process.

The prediction accuracy metrics [112] used in the study are described below. Here p_i and a_i refer to predicted and actual values of i^{th} instance respectively and N is total number of instances.

Correlation Coefficient (C): It measures the extent of relationship between predicted (p_i) and actual (a_i) values. Its value ranges from -1 to 1, where 1 corresponds to ideal case. It is given by:

$$C = \frac{S_{PA}}{\sqrt{S_P S_A}} \quad (3.8)$$

where S_{PA} is covariance of predicted and actual values

$$S_{PA} = \frac{\sum_{i=1}^N (p_i - \bar{p})(a_i - \bar{a})}{N - 1} \quad (3.9)$$

S_P is variance of predicted values

$$S_P = \frac{\sum_{i=1}^N (p_i - \bar{p})^2}{N - 1} \quad (3.10)$$

S_A is variance of actual values

$$S_A = \frac{\sum_{i=1}^N (a_i - \bar{a})^2}{N - 1} \quad (3.11)$$

\bar{p} and \bar{a} are mean of predicted and actual values respectively.

Mean Absolute Error (MAE): For calculating this, the absolute value of the error between predicted and actual values is used. In ideal case its value should be zero. It is calculated as shown below:

$$MAE = \frac{\sum_{i=1}^N |p_i - a_i|}{N} \quad (3.12)$$

Root Mean Squared Error (RMSE): It is measured in the same unit as that of the measured quantity (for this case solo run last level cache stress). Its value ranges from zero to infinity, where zero indicates ideal case. It is calculated in following manner:

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (p_i - a_i)^2}{N}} \quad (3.13)$$

Root Relative Squared Error (RRSE): The relative squared error is made relative to what it would have been if a simple predictor had been used. The simple predictor is just the average of the actual values from the training data. It is calculated as:

$$RRSE = \sqrt{\frac{\sum_{i=1}^N (p_i - a_i)^2}{\sum_{i=1}^N (a_i - \bar{a})^2}} \quad (3.14)$$

Its value is expressed in percentage, where the value 0 corresponds to ideal case.

Relative Absolute Error (RAE): The relative absolute error is similar to root relative squared error, but uses absolute values of errors in calculation. Its value is also expressed in percentage, where the value 0 corresponds to ideal case.

$$RAE = \frac{\sum_{i=1}^N |p_i - a_i|}{\sum_{i=1}^N |a_i - \bar{a}|} \quad (3.15)$$

The afore-mentioned various prediction accuracy metrics values provide an indication on the suitability of machine learning algorithms to capture the knowledge about the given problem. In other words it helps in knowing which algorithms seem appropriate for fitting the given data. Among these metrics *mean absolute error* could be used as an indication for the error in predictions made by the trained model, as it does not exaggerate the effect of outliers. *Root mean squared error* tends to exaggerate the effect of outliers. Outliers are the instances whose prediction error is larger than the other instances. *Correlation coefficient* is a scale independent metric, which means that if we take a particular set of predictions, the *correlation coefficient* remains unchanged if all the predictions are multiplied by a constant factor and the actual values are left unchanged. Good performance of a machine learning algorithm leads to larger values of *correlation coefficient*, whereas the other metrics being error values,

indicate smaller values for good performance. The values of relative error metrics try to compensate for basic predictability or unpredictability of the output variable (also known as class variable). If the output variable tends to lie fairly closely to its average value (which indicates good predictability) and hence the relative metrics compensates by the denominator term used in its calculation. Otherwise, if the output variable does not lie close to its average value then the values of error between predicted values and actual values are far greater because the quantity is inherently more variable and therefore harder to predict [112].

We performed 10 times 10-fold cross validation [124] using weka-3.6.2 experimenter [112]. In 10 times 10-fold cross validation, for every 10-fold cross validation step the data are shuffled in different manner. It ensures that the ten subsets of data used during a 10-fold cross validation, are different than the subsets used in other nine 10-fold cross validations. The total number of instances of data (collected from Intel Xeon X5482 processor based platform) used in the study were 3025. The evaluation results for the different machine learning algorithms used in this study are shown in table 3.6.

Table 3.6: Results of prediction accuracy metrics achieved in 10 times 10-fold cross validation for predicting solo-run last level cache stress on Intel XeonX5482 processor.

Algorithms	<i>Prediction Accuracy Metrics</i>				
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>	<i>RRSE %</i>	<i>RAE %</i>
LR	0.99	0.64	1.06	11.05	8.80
ANN	1.00	0.43	0.66	6.93	6.02
IBK	1.00	0.07	0.37	3.88	1.04
K*	1.00	0.02	0.20	2.05	0.29
M5'	1.00	0.28	0.54	5.69	3.84
SVM	0.99	0.57	0.16	11.98	7.93

The prediction accuracy metrics obtained as shown in table 3.6, indicate that the best performing algorithm is KStar (K*), followed by K-nearest neighbors classifier (IBK) and Model Trees (M5').

The KStar (K*) and K-nearest neighbors classifier (IBK) are lazy or instance-based learning algorithms [93] [115] [116]. In the case of these algorithms all the train-set instances are stored in the model. A new regression equation is fitted each time, when the model needs to predict on a new instance (i.e. a new query point). In case of other four algorithms (mentioned in table 3.6) the fitting of train-set data happens during the training phase. The time taken to build the models (for predicting program memory behavior) using train-set having 3025

instances of data is shown in table 3.7.

Table 3.7: Time taken for training the algorithms to build regression models for predicting solo-run last level cache stress (using train-set with 3025 instances of data).

Algorithms	Time (in seconds) taken to build model
LR	0.35
ANN	16.1
M5'	1.33
SVM	138.6

3.10 Assessing Transferability of the Trained Models for Predicting Solo-run Last Level Cache Stress

Transferability of a regression model refers to usability of the regression model trained in one scenario in predicting the dependent variable in another scenario. In our case a scenario refers to architectural attributes of the memory hierarchy of a multicore processor. It includes interactions between architecture and application with reference to shared last level cache. A regression model trained on architecture A is considered transferable to architecture B if it can be used to accurately predict the solo-run last level cache stress on architecture B . Transferability of trained regression model is an important property for utilization of the model across various scenarios. It amortizes the efforts involved in training.

We performed solo-run and concurrent-run experiments on the other experimental platform (which is based on Intel Core2 6300 processor) to assess the transferability of trained regression model. We collected the program attributes and class variable i.e. solo-run last level cache stress data for experiments conducted on this platform. In the next section we mention the difference between the processors of the two experimental platforms with respect to their last level cache organization to have a view of differences in the scenarios the two platforms offer.

3.10.1 Difference in Last Level Cache Organization on Intel Xeon X5482 and Intel Core2 6300 Processors

The processors present on the experimental platforms i.e. Intel quad-core Xeon X5482 and Intel dual-core Core2 6300 have unified level-2 (L2) cache shared between two cores, which is the last level cache. Table 3.8 shows the last level (L2) cache related data for both processors. It can be seen that the last level cache organization of the two processors (which were used for generating the train-set and test-set data respectively) differs in terms of cache size and ways of associativity.

Table 3.8: Last level cache related data of Intel Xeon X5482 and Intel Core2 6300 processors. The last level cache organization of the two processors differs in terms of cache size and ways of associativity.

Processor	No of cores sharing last level cache	Size (MB) shared between the cores	Ways of associativity
Xeon X5482	2	6	24
Core2 6300	2	2	8

We tested the regression models (built using data collected from Intel Xeon X5482 processor) using data collected from Intel Core2 6300 processor. The testing for transferability across the two different cache organizations was performed for the three best performing algorithms in 10-fold cross validation (described in section 3.9), i.e. KStar (K*), K-nearest neighbors classifier (IBK) and Model Trees (M5'). Next we describe the prediction accuracy metrics of trained regression models observed on the test data.

3.10.2 Prediction Accuracy Metrics of Trained Regression Models on Test Data-set

The results of prediction accuracy metrics of trained regression models on the test data generated from Intel Core2 6300 processor are shown in table 3.9. The number of instances of test data from Intel Core2 6300 processor were 418. These numbers of test instances were generated based on the availability of the platform for performing experiments. The values of prediction accuracy metrics obtained for the test data shown in table 3.9 indicate that the regression

models trained by data from Intel Xeon X5482 are reasonably transferable for predicting the solo-run last level cache stress on other processor i.e. Intel Core2 6300.

Table 3.9: Prediction accuracy of trained models on test data from Intel Core2 6300 for predicting solo-run last level cache stress.

Algorithms	<i>Prediction Accuracy Metrics</i>		
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>
IBK	0.93	2.49	5.78
K*	0.80	2.72	8.55
M5'	0.95	1.60	4.67

The results shown in table 3.9 indicate that among the three algorithms, Model Trees (M5') seems to be the best performing algorithm.

3.11 Conclusions

This chapter described the methodology to build model to characterize program memory behavior on multicore processors. The models were built by training the machine learning algorithms. The trained models were found reasonably transferable to other architecture, that indicates the suitability of potential use of offline trained model by a system management entity. So that the system management entity can adapt the policies to alleviate the performance bottlenecks due to memory hierarchy resource sharing among the cores of the processors.

Chapter 4

Improving Process Scheduling

This chapter describes the application of the trained regression model to improve process scheduling on multicore processors. We implemented a proof of concept meta-scheduler that utilizes the trained model. The meta-scheduler runs in user-mode (unprivileged mode) and guides the scheduling decisions of the underlying operating system process scheduler. Each core of the multicore processor appears as a CPU to the operating system, hence during the description we use the terms core and CPU interchangeably.

The chapter is organized as follows: Section 4.1 begins with discussion of the process scheduler in commodity operating systems (OS) on multicores. In section 4.2, some of the process scheduling issues for multicores are discussed. Section 4.3 gives description of the role of proposed meta-scheduler in addition to the underlying operating system process scheduler present on the system. Section 4.4 describes the policy framework of the meta-scheduler, which provides the design implementation of the meta-scheduler. In section 4.5, we describe the performance evaluation of the meta-scheduler, which includes the details of the methodology adopted along with experimental platform and workload. Section 4.6 provides the results of performance evaluation and information on cost of the trained regression model to predict the solo-run last level cache stress of running programs. At the end we conclude.

4.1 Process Scheduler in Commodity OS

This section describes the role played by process scheduler present in commodity operating systems on multicore processors. It also provides brief overview of the default process scheduler of linux-2.6.30 operating system kernel, which is installed on the experimental platforms used in the work.

The main goal of the process schedulers (also called CPU schedulers) present

in contemporary commodity operating systems (e.g. linux [125]) on multicore systems is to fairly distribute the workload across the available cores. We use terms processes and threads interchangeably, for the perspective of scheduling them on CPU. The strategies employed to place processes on cores of the processor focus on balancing the runnable processes across the available resources to ensure fair distribution of CPU time and minimize the idling of cores.

Process schedulers of contemporary multiprocessor operating systems, such as linux [125] generally have two main components:

- Run-queue management subsystem
- Load balancing subsystem

The run-queue management subsystem uses a distributed run queue model with separate run-queue for each core and fair scheduling policies to schedule processes on each core. The load balancing subsystem redistributes the processes across the cores. The first component does scheduling of the processes in time, while the second component does scheduling of the processes in space.

The default CPU scheduler in linux is the Completely Fair Scheduler (CFS) [77], which was merged into mainline linux kernel version 2.6.23. In CFS, each core is provided with a separate run-queue. The processes which are currently assigned (mapped) to a core are stored in the run-queue of that core. The CFS considers the waiting time of a process (how long it has been in the run-queue and was ready to run) for scheduling. The process with the highest need for CPU time is always scheduled next. The CFS tries to ensure that no process is treated unfairly. The unfairness is directly proportional to the waiting time of the process. Every time the scheduler is called, the process with the highest waiting time is picked and assigned to the CPU. In this way, no large unfairness gets accumulated for any process, and the unfairness will be evenly distributed among all the processes in the system. The CFS maintains a time-ordered red-black tree for each run-queue.

The processes have different completion times as well as different blocking intervals, thereby some run-queues may run out of work, leaving the corresponding processor cores idle, while other processor cores would have processes waiting in their run queues. This is called load imbalance, where the “load” on each core is basically the number of processes in its run-queue. In CFS, load-balancer is periodically invoked to balance the lengths of run-queues to migrate processes from the busiest core (CPU) to less-loaded cores (CPUs). The load balancer ensures that the process selected for migration is not running at the moment or has not been running on a core (CPU) recently. This is done so

that the process should not lose its cache affinity due to migration. It is also required that the process to be migrated must have permission to execute on the destination core (CPU) i.e. the CPU affinity of the process has not been set to prohibit it to run on the destination core (CPU).

In a nutshell the process schedulers in commodity operating systems on multicores are mainly concerned with – distributing the CPU time fairly among all the running programs and distributing the workload fairly among the available cores. Next section mentions some of the process scheduling issues for multicores.

4.2 Process Scheduling Issues for Multicores

An ideal process scheduler for multicore processors should consider resource requirements of the processes as well as interference / contention among the co-running processes for the resources shared among the cores of the processor. The topology of multicore processors includes the information on sharing of the resources (such as last level caches) among the cores of the processors. Hence to achieve next level of performance, the process scheduler (also called CPU scheduler) should be made aware of multicore processor topologies and the characteristics of the processes with respect to utilization of resources shared by the cores.

The schedulers present in contemporary commodity operating systems do not characterize the processes running on the systems with reference to utilization of the resources shared among the cores. The contemporary CPU schedulers do not construct or utilize performance-models to determine more optimal process-to-core mappings. They in general do not take advantage of the hardware performance counters available on the processors. The next section describes the role played by the proposed meta-scheduler.

4.3 Role of Meta-scheduler

The actions of the proposed meta-scheduler are focused only on assigning (mapping) of processes on cores (CPUs). In other words the actions of the meta-scheduler are concerned with space-sharing than time-sharing. Rest of the tasks of the CPU scheduler like run-queue management (which includes the time-multiplexing and enforcing priorities) and load-balancing are left for the underlying process (CPU) scheduler of the operating system running on the system. The meta-scheduler attempts to map the processes on the cores so

that the processes having complementary behavior for utilization of resources shared among the cores become co-runners. As the focus of our study has been on memory hierarchy resources shared among the cores; a pair of memory intensive and compute intensive applications is an example of programs having complementary behavior.

Moreover the meta-scheduler requires the workload to be mix of the memory intensive and compute intensive applications. It performs the space-sharing on the basis of the characteristics of the applications running on the system. If all the applications are equally memory intensive or compute intensive then there is nothing much to be done by the meta-scheduler.

The meta-scheduler also needs that the processor architecture must have multiple cores, where subsets of the cores share different resources. In case of Intel quad-core Xeon X5482 processor (kindly refer to figure 4.3 on page 66), there are two subsets of cores: (i) core0 and core1 share one block of last level (L2) cache, (ii) core2 and core3 share separate block of last level (L2) cache. Thus there are two subsets of cores sharing different resources, thereby allowing the meta-scheduler to function effectively. But in case of AMD quad-core Phenom 9650 processor (kindly refer to figure 5.1a on page 76) and AMD triple-core Phenom 8450 processor (kindly refer to figure 5.1b on page 76) all the processor cores share last level (L3) cache, thereby restricting the meta-scheduler to suggest better mapping of the processes to cores. In any alternate mapping on these processors, all the co-running processes will be sharing the last level caches thereby effectively reducing all the mappings to become similar to each-other with respect to utilization of resources shared among the cores. The next section describes the policy framework of meta-scheduler.

4.4 Policy Framework of Meta-scheduler

The meta-scheduler needs information on following additional aspects of the system to improve the process schedule:

- The topology of the multicore processors: It provides the information about the core layout and memory hierarchy of the processor i.e. sharing of the caches among various cores.
- The requirements of the running programs especially with respect to utilization of the resources shared by the cores of the processor.

The information on processor topology is available with the processor, which is provided by the commodity operating systems like linux [125] via `/sys` or

/proc interface. The information on requirements of the programs with respect to utilization of the resources shared by the cores of the processor is obtained via the trained regression model. The regression model takes the program attributes as inputs and predicts the “solo-run last level cache stress”. The program attributes are calculated from the event data collected from hardware performance counters of the multicore processor.

The mechanism of meta-scheduler could be shown in four blocks viz. Performance monitoring subsystem, Trained regression model, Processor topology knowledge and Decision making. The block diagram of the meta-scheduler is shown in figure 4.1.

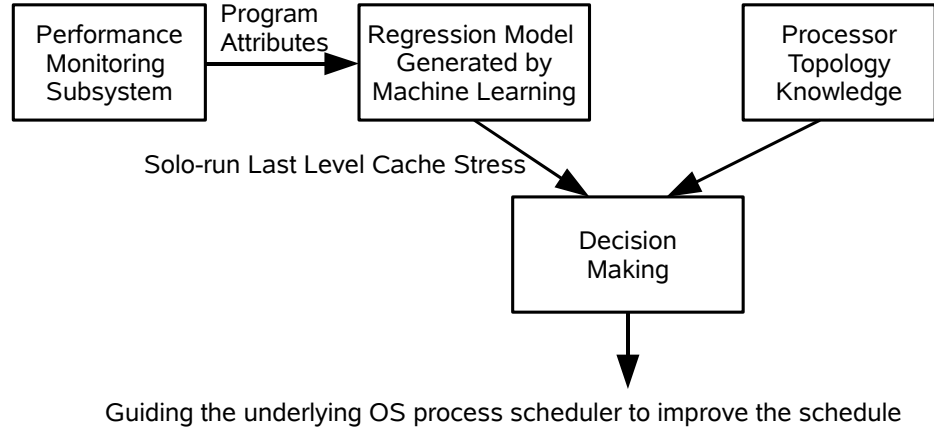


Figure 4.1: Block diagram of the meta-scheduler.

The performance monitoring subsystem does periodic sampling of the following performance event data from the hardware performance counters for the programs running on the processor (Intel Xeon X5482 processor [55] on our experimental platform):

- *INSTRUCTIONS_RETIRED*: It measures the number of completed instructions.
- *LAST_LEVEL_CACHE_REFERENCES*: It measures the number of references to last level (L2) cache.
- *L2_LINES_IN_SELF_ANY*: It measures the total number of last level (L2) cache lines brought in as a result of encountering last level (L2) cache misses as well as prefetching activities.

The sampling period is 500 milliseconds. The sampled performance event data are used to calculate the four attributes for running programs (calculation is similar as described in section 3.8.3 of chapter 3, on page 49). In case if two programs $p1$ and $p2$ are co-running on cores of the processor, which share the last level (L2) cache, the four attributes for the program $p1$ could be described as follows:

- $p1_L2_REF_PKI$: Last level (L2) cache references per kilo instructions retired for the program $p1$.
- $p1_L2_IN_PKI$: Last level (L2) cache lines brought in (due to miss and prefetch) per kilo instructions retired for the program $p1$.
- $p1_L2_IN_PK_REF$: Last level (L2) cache lines brought in (due to miss and prefetch) per kilo last level (L2) cache lines referenced for the program $p1$.
- $p1_L2_FO$: Fractional last level (L2) cache occupancy of the program $p1$. It gives a rough estimate of the fraction of space occupied by the program $p1$ in last level (L2) cache, while sharing it with other co-running program $p2$.

The meta-scheduler uses the off-line trained regression model (developed using the methodology described in previous chapter 3), in on-line manner to deduce the solo-run last level cache stress of running programs. The regression model used by meta-scheduler is a model tree built by training the Model Tree (M5') machine learning algorithm of weka machine learning workbench [112] [113] (Model Tree (M5') machine learning algorithm has been observed as one of the best performing algorithm, described in previous chapter 3).

Model trees are binary decision trees, where the leaf nodes contain linear regression functions. The model tree generation process uses divide and conquer approach to recursively partition the input space (training set) into a number of disjoint hyperspaces. The partitioning is reflected in the generated tree model. The disjoint hyperspaces generated in the process become linear models at leaf nodes.

Figure 4.2 shows part of the built model tree used by meta-scheduler to deduce solo-run last level cache stress of programs running on multicore processors. At the root of the model tree, value of attribute $p1_L2_IN_PKI$ is checked. At each internal node, the decision on path to be followed to go further down is made on the basis of one of the four attributes used to train the machine learning algorithm. After reaching the leaf node the corresponding linear model

Part of built Model Tree

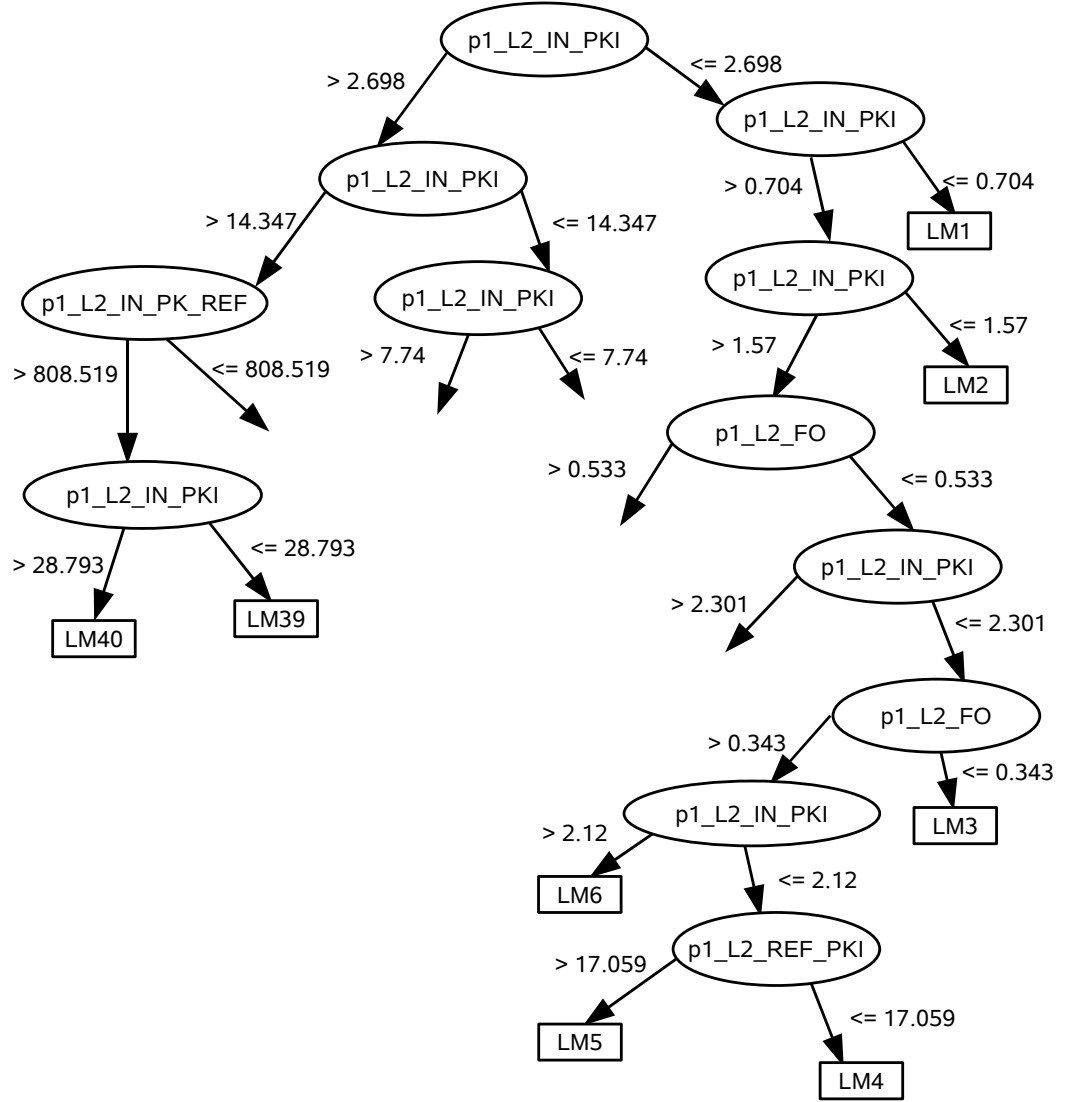


Figure 4.2: Part of built model tree used by meta-scheduler to deduce solo-run last level cache stress of programs running on multicore processors. The LM1, LM2, ..., LM39, LM40 are linear models at the leaves of the model tree.

is used to calculate the solo-run last level cache stress. The LM1, LM2, ..., LM39, LM40 shown in figure 4.2, are linear models at the leaves of the model tree. For example, the linear functions for LM1 and LM2 are shown in equation 4.1 and equation 4.2 respectively (here *p1_solo-run last level cache stress* is solo-run last level cache stress for program *p1*).

$$\begin{aligned}
 LM1 : p1_solo - run\ last\ level\ cache\ stress = & -0.0007 * p1_L2_REF_PKI \\
 & +0.0244 * p1_L2_IN_PKI - 0 * p1_L2_IN_PK_REF \quad (4.1) \\
 & +0.0204 * p1_L2_FO + 0.1721
 \end{aligned}$$

$$\begin{aligned}
 LM2 : p1_solo - run\ last\ level\ cache\ stress = & -0.0015 * p1_L2_REF_PKI \\
 & +0.056 * p1_L2_IN_PKI - 0.0001 * p1_L2_IN_PK_REF \quad (4.2) \\
 & +0.0645 * p1_L2_FO - 0.429
 \end{aligned}$$

The built model tree contains the knowledge learnt during the process of training, which it utilizes later to predict the solo-run last level cache stress, for any given values of the four program attributes.

Inside the meta-scheduler the solo-run last level cache stress of the program *p1* is deduced by using the four attributes (calculated from data sampled from hardware performance counters) as inputs to the off-line built regression model as shown in equation 4.3.

$$\begin{aligned}
 RegressionModel(p1_L2_REF_PKI, p1_L2_IN_PKI, p1_L2_IN_PK_REF, p1_L2_FO) \\
 \rightarrow p1_solo - run\ last\ level\ cache\ stress \quad (4.3)
 \end{aligned}$$

The model tree used by the meta-scheduler is generated by train-set containing 2209 instances, where each instance correspond to a pair of SPEC cpu2006 applications. The set of SPEC cpu2006 applications used for training the machine learning algorithm does not include the SPEC cpu2006 applications used as workload for performance evaluation of the meta-scheduler (mentioned in section 4.5.2 on page 67).

Using processor topology information the meta-scheduler divides the CPUs available on the system into two sets, such that the CPUs present in a set do not share the resources such as last level caches on the processor. For the topology of Intel quad-core Xeon X5482 processor (shown in figure 4.3), the meta-scheduler divides the four cores (CPUs) into two sets: Set-1: {0, 2} and Set-2: {1, 3}. The cores (CPUs) present in each set do not share last level (L2) cache among them.

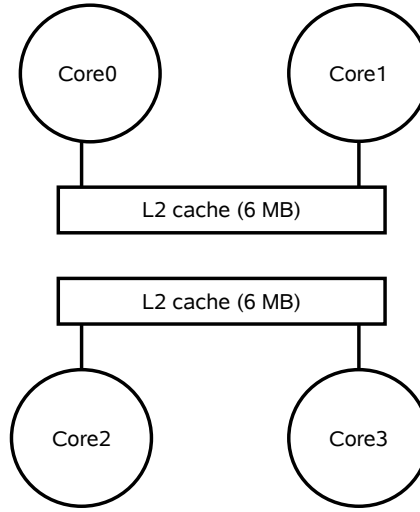


Figure 4.3: Topology of Intel quad-core Xeon X5482 processor showing the sharing of last level (L2) cache among the cores of the processor. The meta-scheduler divides the four cores (CPUs) into two sets: Set-1: {0, 2} and Set-2: {1, 3} such that the cores (CPUs) present in each set do not share last level (L2) cache among them.

The meta-scheduler divides all the running programs into two groups on the basis of their solo-run last level cache stress: higher-solo-run-last-level-cache-stress-group and lower-solo-run-last-level-cache-stress-group. Processes in higher-solo-run-last-level-cache-stress-group are bound with cores (CPUs) present in Set-1, and those in lower-solo-run-last-level-cache-stress-group are bound with cores (CPUs) present in Set-2. Thus the programs having higher solo-run-last-level-cache-stress run with programs having lower-solo-run-last-level-cache-stress as their co-runner and the interference / contention due to use of resources (e.g. last level cache) shared among the cores is minimized. The meta-scheduler uses `sched_setaffinity` system call for binding the programs to the cores.

4.5 Performance Evaluation of Meta-scheduler

This section presents the experimental methodology used for performance evaluation of the meta-scheduler. First we mention about the experimental platform and workload.

4.5.1 Experimental Platform

We used dual-socket DELL Precision T7400 workstation to measure the improvement in performance due to use of meta-scheduler. The specifications of the experimental platform are described in table 4.1. The experimental

Table 4.1: Specifications of the experimental platform used for performance evaluation of intelligent scheduling done by meta-scheduler.

Item	Specification
Number of chips	2
Number of cores	4 per chip
Number of hardware threads	1 per core
CPU cores	Intel Xeon X5482, 3.2 GHz
L1 Instruction Cache	32KB 8-way set-associative per core
L1 Data Cache	32KB 8-way set-associative per core
L2 Unified Cache	Total 12MB, Each 6MB 24-way set-associative L2 cache block is shared by two cores
RAM	32 GB

platform has two Intel quad-core Xeon X5482 processors. We performed experiments using four as well as eight cores of the system.

4.5.2 Workload

We used set of applications from SPEC cpu2006 benchmark suite [74] [78] with reference inputs, as workload to evaluate the performance. Table 4.2 describes the applications from SPEC cpu2006 benchmark suite, which were used to form various sets of workloads. Out of total eight applications men-

Table 4.2: Description of SPEC cpu2006 applications used in performance evaluation with meta-scheduler.

Name	Description	Characteristic
429.mcf	Combinatorial Optimization	Memory-bound
471.omnetpp	Discrete Event Simulation	Memory-bound
450.soplex	Linear Programming, Optimization	Memory-bound
482.sphinx3	Speech recognition	Memory-bound
462.libquantum	Quantum Computing	Memory-bound
453.povray	Image Ray-tracing	CPU-bound
444.namd	Biology/Molecular Dynamics	CPU-bound
464.h264ref	Video Compression	CPU-bound

tioned in table 4.2, five applications are memory-bound and three applications are CPU-bound. Among the five memory-bound applications, one application 482.sphinx3 is moderate and remaining four are heavy users of memory.

The formations of sets of workloads used in the experiments are mentioned in table 4.3.

Table 4.3: Sets of SPEC cpu2006 applications used in experiments performed for performance evaluation with meta-scheduler.

Set 1	Set 2	Set 3	Set 4	Set 5
450.soplex	450.soplex	429.mcf	429.mcf	482.sphinx3
482.sphinx3	429.mcf	462.libquantum	471.omnetpp	444.namd
444.namd	444.namd	444.namd	464.h264ref	453.povray
453.povray	453.povray	453.povray	444.namd	464.h264ref
Set 6	Set 7	Set 8	Set 9	Set 10
450.soplex	482.sphinx3	482.sphinx3	482.sphinx3	482.sphinx3
444.namd	450.soplex	450.soplex	471.omnetpp	471.omnetpp
453.povray	429.mcf	429.mcf	429.mcf	429.mcf
464.h264ref	444.namd	464.h264ref	453.povray	444.namd

Table 4.4 describes the characteristics of the workload sets used for performance evaluation. Set 1, 2, 3 and 4 are composed of two memory-bound and two CPU-bound applications; set 5 and 6 are composed of one memory-bound and three CPU-bound applications and set 7, 8, 9 and 10 are composed of three memory-bound and one CPU-bound applications.

Table 4.4: Characteristics of the workload sets used in experiments performed for performance evaluation with meta-scheduler.

Characteristics of the application mix	Set numbers
2 Memory-bound + 2 CPU-bound	1, 2, 3, 4
1 Memory-bound + 3 CPU-bound	5, 6
3 Memory-bound + 1 CPU-bound	7, 8, 9, 10

4.5.3 Experiments

Each set of workload consisted of four applications. Each application in a set was run in an infinite loop like the method used by Bulpin and Pratt [34], i.e. it was restarted if it got finished. The period of experimental run was thirty minutes.

We ran one copy of each application per core (CPU). Hence in four-core experiments all the four programs from a set of workload were run on four cores of the processor.

The eight-core experiments involved running of two copies of each application from a set of workload to get total eight programs to run on all the eight cores.

We collected the event data for instructions retired and core cycles consumed by each application during the experimental run from respective hardware performance counters of the processor.

In both four-core as well as eight-core experiments, each set of workload was run two times in manner mentioned above and the performance event data were collected. First time the workload set was run without meta-scheduler (i.e. only with the process scheduler of linux OS running on the system). Second time the workload set was run with the meta-scheduler. In this way the performance data were collected with and without the meta-scheduler for each set of workload. Next section presents the performance evaluation results.

4.6 Results

The overall (aggregate) speedup was calculated as shown in equation 4.4.

$$Aggregate\ Speedup = \frac{averageIPC_M}{averageIPC} \quad (4.4)$$

The overall (aggregate) speedup is improvement in performance observed with meta-scheduler as compared (normalized) to default operating system (i.e. linux-2.6.30) process scheduler. The $averageIPC_M$ and $averageIPC$ are average instructions retired per cycle across all the processes 1 to n , with meta-scheduler and without meta-scheduler (i.e. with default linux operating system process scheduler) respectively.

The values of average instructions per cycle were calculated from the event data for instructions retired and core cycles consumed by each of the program as shown in equation 4.5.

$$averageIPC = \frac{\sum_{i=1}^n Instructions_retired}{\sum_{i=1}^n Cycles} \quad (4.5)$$

Table 4.5 and table 4.6 list the values of $averageIPC$, $averageIPC_M$ and normalized speedup for various sets (mixes) of programs, observed in 4-core and 8-core experiments respectively.

The normalized speedup observed in 4-core and 8-core experiments is also shown in figure 4.4 and figure 4.5 respectively.

Table 4.5: IPC , IPC_M and normalized speedup for various sets of workloads in 4-core experiments on Intel quad-core Xeon X5482 processor based platform.

Set no	$averageIPC$ (4-core)	$averageIPC_M$ (4-core)	Normalized speedup (4-core)
Set 1	1.029	1.199	1.16
Set 2	0.623	0.806	1.29
Set 3	0.663	0.752	1.13
Set 4	0.498	0.594	1.19
Set 5	1.168	1.421	1.22
Set 6	1.173	1.372	1.17
Set 7	0.450	0.741	1.64
Set 8	0.417	0.733	1.76
Set 9	0.380	0.431	1.13
Set 10	0.381	0.507	1.33

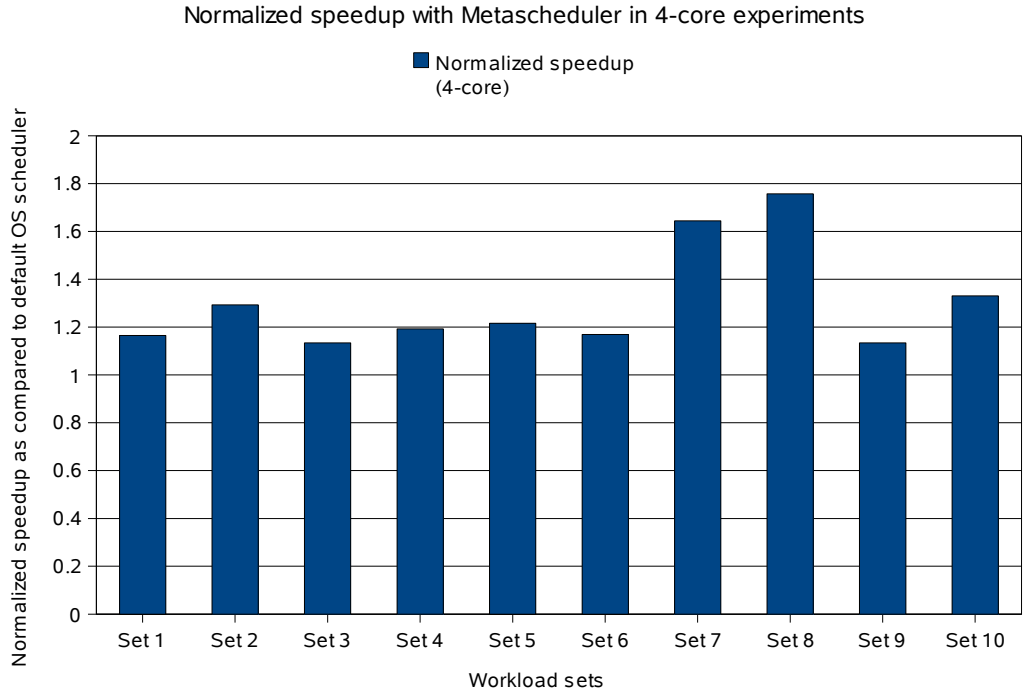


Figure 4.4: Normalized speedup with meta-scheduler in 4-core experiments on Intel quad-core Xeon X5482 processor based platform.

Table 4.6: IPC , IPC_M and normalized speedup for various sets of workloads in 8-core experiments on Intel quad-core Xeon X5482 processor based platform.

Set no	$averageIPC$ (8-core)	$averageIPC_M$ (8-core)	Normalized speedup (8-core)
Set 1	0.774	0.957	1.24
Set 2	0.201	0.221	1.1
Set 3	0.456	0.593	1.3
Set 4	0.295	0.356	1.21
Set 5	1.391	1.422	1.02
Set 6	1.386	1.395	1.01
Set 7	0.438	0.509	1.16
Set 8	0.268	0.412	1.54
Set 9	0.280	0.406	1.45
Set 10	0.312	0.467	1.5

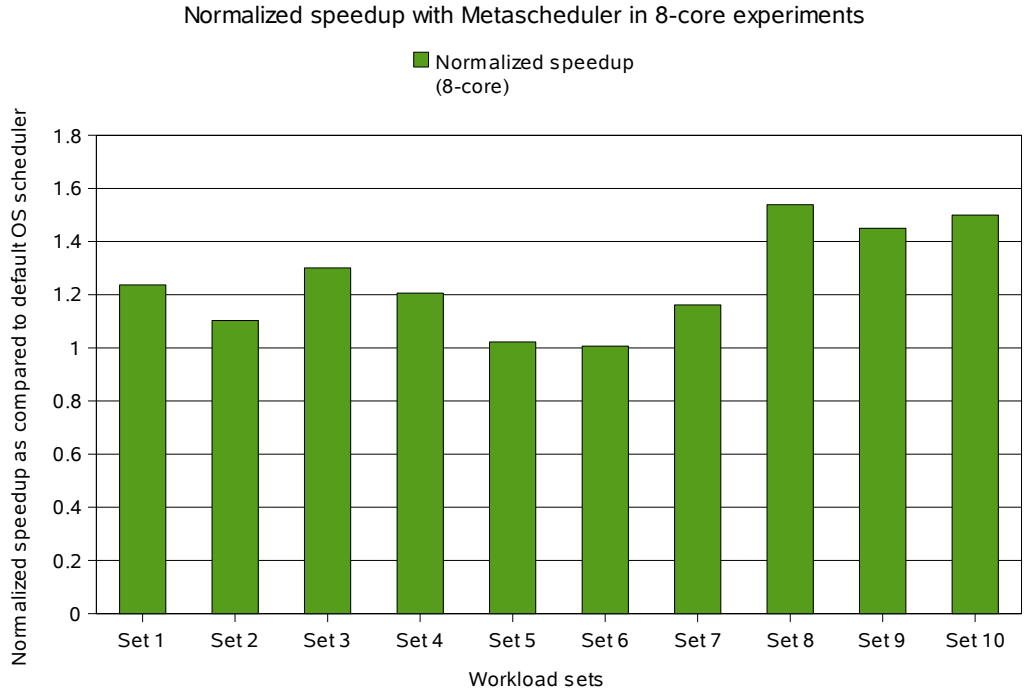


Figure 4.5: Normalized speedup with meta-scheduler in 8-core experiments on Intel quad-core Xeon X5482 processor based platform.

In performance evaluation experiments performed with various sets (mixes) of workload, we observed overall speedup of up to 76% (for 4-cores) and 54% (for 8-cores) with meta-scheduler as compared to default operating system process scheduler of linux-2.6.30.

The cost of model includes the cost in terms of time required for the computation of solo-run last level cache stress by offline-trained regression model. To find the cost of model for deducing the solo-run last level cache stress, we measured the time in terms of core cycles consumed (from the respective hardware performance counters of the processor) to perform all the required computation done by the trained regression model. The average time consumed in calculating four program attributes and using trained model to predict the solo-run last level cache stress is 0.00075 % (i.e. 7-8 cycles per million cycles) of total time for the experiments conducted in the study.

4.7 Conclusions

In this chapter we described an example application of the offline-trained model, which was built for characterization of program memory behavior. The meta-scheduler described in the chapter used the offline-trained model to improve the CPU scheduling. The meta-scheduler runs on existing commodity multi-core processor based platform and does not require any specialized support or modifications in the existing hardware. It also does not require any modifications in the operating system process scheduler as well as applications. This demonstrates that the model built by the process of machine learning could be utilized by a system management entity to make or improve its policy decisions.

Chapter 5

Performance Prediction

The focus of this chapter is to predict the performance of the applications on multicore processors. Performance prediction on multicores should take into account the following performance implications, which are manifested due to resource sharing among the cores:

- The extent by which a program suffers with performance degradation due to interference of its co-running programs.
- The extent by which a program can degrade performance of its co-running programs.

We observed the peculiarities of the performance impacts arising due to resource sharing among the cores (kindly refer to figure 3.2 on page 37), which are mentioned below:

- The performance impact caused by interference among co-running programs due to use of resources shared by the processor cores is non-uniform.
- The amount of performance degradation a program suffers during concurrent run, differs when it is run with different co-running program.
- The quantum of degradation in performance, depends on the resource utilization behavior of the program itself and the co-running programs, with those the first program shares the resources.

The concurrent-run performance of a program on multicore results from the interactions of the behavior of itself and its co-runners. It contains the effects of all the peculiarities mentioned.

In this chapter we use machine learning techniques to predict the concurrent-run performance of programs on multicore processors. We propose the solo-run attributes of programs to predict their concurrent-run performance. Such

performance prediction techniques could be useful for creation of workloads for performance studies on multicore architecture. It can also be used in simulation of multicore processors.

The chapter is organized as follows: In section 5.1, we describe the experimental setup used in the study. Section 5.2 describes the experimental methodology. In section 5.3, we discuss about the generation of solo-run performance attributes of the programs. Section 5.4 describes the prediction accuracy results. In section 5.5, we provide results on transferability of the trained model. Section 5.6 describes the application of the trained model to improve multi-core simulation in AKULA [54], which is a recently developed tool-set for rapid prototyping of scheduling algorithms on multicores. At the end we conclude.

Reader may kindly refer to section 3.6 at page 43 for description of the machine learning algorithms used.

5.1 Experimental Setup

In this section we provide information on the experimental platforms and workload used for performance prediction. The section also describes memory hierarchy of the processors on the platforms used in the study.

5.1.1 Experimental Platforms

We performed experiments on four platforms. The first experimental platform is dual-socket DELL Precision T7400 workstation with two Intel quad-core Xeon X5482 processors (3.2 GHz) and 32 GB of memory (table 3.3 on page 45 may please be referred for specifications). Data collected from the first experimental platform were used to build the regression models by training the machine learning algorithms.

Data collected from the other three experimental platforms were used to assess the transferability of the regression models (those were built using the data collected from first experimental platform based on Intel Xeon X5482 processor).

The second experimental platform is single socket DELL Precision 390 workstation with one Intel dual-core Core2 6300 processor (1.86 GHz) and 2 GB of memory (table 3.4 on page 45 may please be referred for specifications).

The third experimental platform is a single socket personal computer with one AMD quad-core Phenom 9650 processor (1.15 GHz) and 4 GB of memory. The specifications of the AMD quad-core Phenom 9650 processor based

platform are described in table 5.1.

Table 5.1: Specifications of AMD quad-core Phenom 9650 processor based platform used for generating data to test regression models built for performance prediction.

Item	Specification
Number of chips	1
Number of cores	4 per chip
Number of hardware threads	1 per core
CPU cores	AMD Phenom 9650, 1.15 GHz
L1 Instruction Cache	64KB 2-way set-associative per core
L1 Data Cache	64KB 2-way set-associative per core
L2 Unified Cache	512KB 16-way set-associative per core
L3 Unified Cache	2MB 32-way set-associative shared by four cores
RAM	4 GB

The fourth experimental platform is a single socket personal computer with one AMD triple-core Phenom 8450 processor (1.05 GHz) and 2 GB of memory. The specifications of the AMD triple-core Phenom 8450 processor based platform are described in table 5.2.

The operating system kernel on all the experimental platforms was linux-2.6.30. For collecting data from the hardware performance counters of the processors [55] [126] present on the experimental platforms, we used perfmon2 [122] interface.

We used all the 55 benchmark programs from SPEC cpu2006 benchmark suite [74] [78] as workload for performance prediction studies.

Table 5.2: Specifications of AMD triple-core Phenom 8450 processor based platform used for generating data to test regression models built for performance prediction.

Item	Specification
Number of chips	1
Number of cores	3 per chip
Number of hardware threads	1 per core
CPU cores	AMD Phenom 8450, 1.05 GHz
L1 Instruction Cache	64KB 2-way set-associative per core
L1 Data Cache	64KB 2-way set-associative per core
L2 Unified Cache	512KB 16-way set-associative per core
L3 Unified Cache	2MB 32-way set-associative shared by three cores
RAM	2 GB

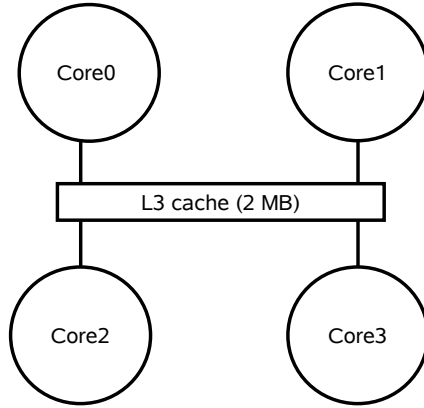
5.1.2 Processor Memory Hierarchy on the Experimental Platforms

Kindly refer to section 3.7.2 on page 46 for description of the memory hierarchy on Intel quad-core Xeon X5482 as well as Intel dual-core Core2 6300 processors.

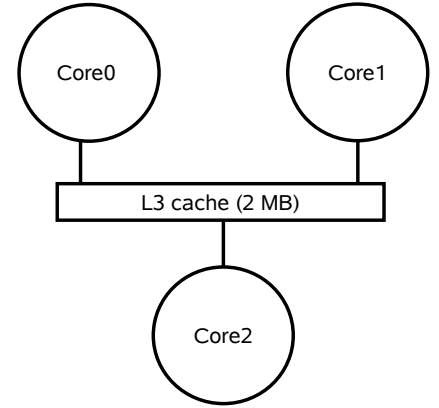
Both AMD Phenom 9650 and AMD Phenom 8450 processors have three levels of caches. Each core on both processors has level-1 (L1) instruction and level-1 (L1) data cache, each is 2-way set-associative and is of size 64KB.

Each core also has its own private level-2 (L2) cache, which is 16-way set-associative and is of size 512KB. The level-2 (L2) cache is unified in nature, i.e. it contains both data as well as instructions.

There is unified level-3 (L3) cache of size 2MB. The L3 cache is the last level cache, which is 32-way set-associative. On AMD Phenom 9650 processor, the L3 cache is shared by all the four processor cores, and on AMD Phenom 8450, it is shared by all the three processor cores. The sharing of last level cache among the cores of AMD Phenom 9650 and AMD Phenom 8450 processors is shown in figure 5.1a and figure 5.1b respectively.



(a) AMD Phenom 9650 processor.



(b) AMD Phenom 8450 processor.

Figure 5.1: Last level (L3) cache sharing on AMD Phenom 9650 and AMD Phenom 8450 processors, used for generating data to test regression models built for performance prediction.

The experimental platforms used for generating the train-set and test-set data differ in terms of the organization of processor memory hierarchy on them. The organization of processor caches differs mainly on following aspects:

- Number of levels of caches

- Size of the caches
- Associativity of caches
- Number of cores sharing the last level caches

Moreover the caches present at different levels (i.e. L1, L2 etc.) are inclusive on Intel quad-core Xeon X5482 and Intel Core2 6300 processors [55], while on AMD Phenom 8450 and Phenom 9650 processors they are exclusive [126].

The next section describes the solo-run and concurrent-run experiments for collecting the data.

5.2 Experiments

Each instance of train-set as well as test-set data comprised of the solo-run program attributes and class variable “concurrent-run performance”.

The solo-run program attributes were calculated from hardware performance counter data collected for complete run of each SPEC cpu2006 benchmark program [74] [78] in solo-run experiment.

In concurrent-run experiment, the programs from SPEC cpu2006 benchmark suite were run on the cores sharing last level cache. The values of class variable “concurrent-run performance” were calculated from the hardware performance counter data collected in this experiment. On Intel Xeon X5482 processor two processor cores share the last level(L2) cache, hence with 55 programs, we have 55×55 (i.e. 3025) pairs of programs to run on it.

In concurrent-run experiment on Intel Xeon X5482 and Intel Core2 6300 processor based platforms, two programs were run simultaneously, as there are two cores on the processor sharing the last level (L2) caches. One program was run on a core along with other on the core sharing the resources with the first. The performance counter data for complete run of the first program were collected. As the programs have different run-times, the other program was re-run if it finished before the completion of the first program.

In concurrent-run experiment on AMD Phenom 9650 processor based platform, four programs were run simultaneously; because on AMD Phenom 9650 processor, four cores share the last level (L3) cache. The performance counter data for complete run of the first program were collected. As the programs have different run-times, each of the other three programs was re-run if it finished before the completion of the first program. In the similar fashion on AMD Phenom 8450 processor based platform, three programs were run simultaneously, as three cores on it share the last level (L3) cache.

5.3 Generation of Solo-run Program Attributes & Class Variable

This section describes the solo-run program attributes to predict the concurrent-run performance on multicores. We begin with our first proposal of the solo-run program attributes and its shortcomings followed by the modified proposal.

The model developed in our first study [127], used solo-run attributes of a program as well as its co-running program to predict its concurrent-run (paired-run) performance. The study was performed on Intel Xeon X5482 processor based experimental platform, on which two cores share the last level (L2) cache. For example, consider programs A and B co-scheduled on two cores, which share the resources. In the concurrent-run both programs interfere with each other due to usage of the resources shared by the cores. The model f to predict the concurrent-run performance of program A , can be expressed in the form of the equation 5.1:

$$f((a_{1A}, a_{2A}, \dots, a_{8A}), (a_{1B}, a_{2B}, \dots, a_{8B})) \rightarrow \text{concurrent_run_CPI}_A \quad (5.1)$$

In equation 5.1, $\text{concurrent_run_CPI}_A$ is paired-run (concurrent-run) performance of program A in terms of cycles per instructions, $(a_{1A}, a_{2A}, \dots, a_{8A})$ and $(a_{1B}, a_{2B}, \dots, a_{8B})$ are set of solo-run program attributes of program A and B respectively.

The list of the eight solo-run program attributes used by the model is given below:

- *LAST_LEVEL_CACHE_REFERENCES_PKI*:
Number of references to last level (L2) cache per kilo instructions.
- *L2_LINES_IN_SELF_ANY_no_prefetch_PKI*:
Number of misses occurring at last level (L2) cache per kilo instructions.
- *L2_LINES_IN_SELF_ANY_no_prefetch_PK_LAST_LEVEL_CACHE_REFERENCES*:
Number of misses occurring at last level (L2) cache per kilo references. This gives a rough estimate of cache re-referencing property of the program at last level (L2) cache.
- *L1D_REPL_PK_L1D_ALL_CACHE_REF*:
Number of misses at first level (L1) data cache per kilo references. This gives a rough estimate of cache re-referencing property of the program at L1 data cache.

- *L1L_MISSES_PK_L1L_READS*:
Number of misses at first level (L1) instruction cache per kilo reads. This gives a rough estimate of cache re-referencing property of the program at L1 instruction cache.
- *DTLB_MISSES_ANY_PKI*:
Number of misses at data translation look aside buffer per kilo instructions. This gives a rough estimate towards spread of data access of the program.
- *ITLB_MISSES_PKI*:
Number of misses at instruction translation look aside buffer per kilo instructions. This gives a rough estimate towards spread of instruction access of the program.
- *CPI*:
Cycles per instruction. It encapsulates the effect of overall resource utilization behavior of the program.

The values of the class variable i.e. concurrent-run performance (in terms of CPI) were generated from hardware performance counter data collected during concurrent-run experiment described in section 5.2. Each instance of the training data is comprised of 8+8 i.e. 16 solo-run program attributes for two programs and the paired-run (concurrent-run) performance (in terms of CPI) of first program (which ran to completion in the concurrent-run experiment).

In case of two co-running programs sharing the resources, the model (trained by the afore-mentioned program attributes, model shown in equation 5.1) needed solo-run attributes of both programs as inputs for performance prediction. Thus a model trained for case of two cores sharing the resources, can not be used for predicting the performance, when the number of cores sharing the resources are more than two. For example, on Intel Xeon E5630, Intel Core i5 and Intel Core i7 processors [55], four cores share the last level cache. On AMD Phenom 8450 processor, three cores present on the chip share the last level cache; and on AMD Phenom 9650 processor, four cores share the last level cache [56] [126].

Such cases require new model, which should consider the solo-run program attributes of all the co-running programs. This requires us to again perform the solo-run and concurrent-run experiments on the new platform to collect the data to train the machine learning algorithms for building the model for each of the new scenario (i.e. three cores sharing the resources or four cores sharing the resources).

We proposed new solo-run program attributes [128] to overcome this limitation. Here we use fractional values of the hardware performance counter data to compute these new solo-run program attributes. The new solo-run attributes contain the fraction of the hardware performance counter data for the program under consideration with the similar data of its co-running programs. Hence the model trained once for two cores sharing the resources, can be used for making performance predictions when the number of cores sharing the resources are more than two. We validated this aspect by testing the model (built using data from Intel Xeon X5482 processor) on test-data from AMD Phenom 9650 and AMD Phenom 8450 processors. The results on transferability of the trained model on other processors are provided in section 5.5.

The new program attributes to train the model are calculated from solo-run hardware performance counter data of the program as well as its co-runners. The performance events which can be collected on the experimental platforms differ, as the performance monitoring unit of a processor is specific to its micro-architecture. The events available on the processors used in the study are described in respective processor manuals from Intel [55] and AMD [126]. We collected the performance events which are specific to utilization of the memory hierarchy resources shared among the cores. The performance event data collected during the experiments are mentioned below:

- *INSTRUCTIONS_RETIRED*:

Number of instructions completed. The performance event used for collecting this data is named INSTRUCTIONS_RETIRED on Intel Xeon X5482 and Intel Core2 6300 processors. On AMD Phenom 9650 and AMD Phenom 8450 processors, the performance event used for collecting this data is named RETIRED_INSTRUCTIONS.

- *UNHALTED_CORE_CYCLES*:

Number of core cycles completed. The performance event used for collecting this data is named UNHALTED_CORE_CYCLES on Intel Xeon X5482 and Intel Core2 6300 processors. On AMD Phenom 9650 and AMD Phenom 8450 processors, the performance event used for collecting this data is named CPU_CLK_UNHALTED.

- *LAST_LEVEL_CACHE_REFERENCES*:

Number of references to last level cache. The performance event used for collecting this data is named LAST_LEVEL_CACHE_REFERENCES on Intel Xeon X5482 and Intel Core2 6300 processors. On AMD Phenom 9650 and AMD Phenom 8450 processors, we can not count core specific

events for last level (L3) cache [126] [129] [130]. Though the processor manual [126] specifies core select masks for getting the core specific events from last level (L3) cache, the prescribed masks do not work in practice [129] and there is an official erratum from AMD [130] regarding this issue. Due to this we used event data for level-2 (L2) cache for generating the solo-run program attributes. Hence on AMD Phenom 9650 and AMD Phenom 8450 processors, the performance event used for collecting this data is named REQUESTS_TO_L2_ALL, which basically collects the number of references to level-2 (L2) cache.

- *LAST_LEVEL_CACHE_MISS:*

Number of misses and prefetches occurring at last level cache. On Intel Xeon X5482 and Intel Core2 6300 processors, the performance event used for collecting this data is named L2_LINES_IN_SELF_ANY. Due to non-functioning of the core specific event mask for last level (L3) cache on AMD Phenom 9650 and AMD Phenom 8450 processors (discussed in previous paragraph), we used event data for level-2 (L2) cache for generating the solo-run program attributes. On AMD Phenom 9650 and AMD Phenom 8450 processors, the performance event used for collecting this data is named L2_CACHE_MISS_ALL, which collects the number of misses and prefetches at level-2 (L2) cache.

The list of the solo-run program attributes calculated from the aforementioned performance event data collected in solo-run experiment, is given below:

- *LAST_LEVEL_CACHE_REFERENCES_PKI:*

Number of references to last level cache per kilo instructions.

- *LAST_LEVEL_CACHE_IN_PKI:*

Number of misses and prefetches occurring at last level cache per kilo instructions.

- *LAST_LEVEL_CACHE_IN_PK_LAST_LEVEL_CACHE_REFERENCES:*

Number of misses and prefetches occurring at last level cache per kilo references. This gives a rough estimate of cache re-referencing property of the program at last level cache.

- *Fr_LAST_LEVEL_CACHE_REFERENCES_PKI:*

Fraction of number of references to last level cache per kilo instructions of the program as compared to its co-running programs.

- *Fr_LAST_LEVEL_CACHE_IN_PKI*:
Fraction of number of misses and prefetches to last level cache per kilo instructions of the program as compared to its co-running programs.
- *Fr_CPI*:
Fraction of solo-run cycles per instruction of the program as compared to its co-running programs. It encapsulates the effect of overall resource utilization behavior of the program as compared to its co-running programs.
- *solo_run_CPI*:
Cycles per instruction of the program for its solo run. It encapsulates the effect of overall resource utilization behavior of the program.

Above mentioned seven solo-run program attributes were used to predict the performance of the program in terms of *concurrent_run_CPI* i.e. cycles per instruction of the program for its concurrent run. Hence the new model to predict the concurrent-run performance of program *A*, can be expressed in the form of equation 5.2:

$$f_{new}(a_{1A}, a_{2A}, a_{3A}, a_{4A}, a_{5A}, a_{6A}, a_{7A}) \rightarrow \text{concurrent_run_CPI}_A \quad (5.2)$$

In equation 5.2, $a_{1A}, a_{2A}, \dots, a_{7A}$ are seven solo-run attributes for program *A*. Due to use of new program attributes especially the fractional attributes, the model shown by equation 5.2 always needs seven attributes for a program to predict its concurrent-run performance. The previous model shown in equation 5.1 on page 78, takes solo-run attributes of two applications to predict the concurrent-run performance (for the case of two applications co-running on two cores sharing the resources). The model shown in equation 5.1 needs to be rebuilt to consider the cases with different numbers of cores sharing the memory hierarchy resources.

An instance of train-set data could be shown as an eight-tuple having solo-run program attributes as first seven members and class variable *concurrent_run_CPI* as the last member. A sample of train-set data for performance prediction collected from Intel Xeon X5482 processor is shown in table 5.3.

5.4 Results on Prediction Accuracy for Predicting Concurrent-run Performance

We performed 10 times 10-fold cross validation [124] using weka-3.6.2 experimenter [112]. Total number of instances in the data set were 3025, which

Table 5.3: Sample of train-set data for performance prediction collected from Intel Xeon X5482 processor. First seven columns list the solo-run program attributes and the last column shows class variable as follows: Column 1: *LAST_LEVEL_CACHE_REFERENCES_PKI*, Column 2: *LAST_LEVEL_CACHE_IN_PKI*, Column 3: *LAST_LEVEL_CACHE_IN_PK_LAST_LEVEL_CACHE_REFERENCES*, Column 4: *Fr_LAST_LEVEL_CACHE_REFERENCES_PKI*, Column 5: *Fr_LAST_LEVEL_CACHE_IN_PKI*, Column 6: *Fr_CPI*, Column 7: *solo_run_CPI*, Column 8: *concurrent_run_CPI*.

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8
9.074	0.629	69.306	0.315	0.061	0.284	0.715	0.815
19.730	9.763	494.810	0.500	0.500	0.500	1.802	2.706
17.455	3.296	188.855	0.469	0.252	0.389	1.147	1.488
17.228	7.771	451.054	0.466	0.443	0.467	1.581	2.214
16.152	8.092	500.992	0.450	0.453	0.449	1.470	1.954
17.985	9.153	508.912	0.477	0.484	0.472	1.609	2.167
18.835	9.176	487.156	0.488	0.484	0.472	1.608	2.207
21.546	14.805	687.139	0.522	0.603	0.545	2.160	2.903
17.671	10.092	571.093	0.472	0.508	0.496	1.772	2.463
17.456	1.562	89.505	0.469	0.138	0.351	0.974	1.201
136.232	42.801	314.173	0.873	0.814	0.766	5.890	7.829
16.701	0.135	8.111	0.458	0.014	0.269	0.664	0.695
15.376	0.506	32.901	0.438	0.049	0.364	1.030	1.108
13.738	0.422	30.688	0.410	0.041	0.357	0.998	1.070
13.264	2.176	164.061	0.402	0.182	0.368	1.051	1.145
15.161	0.700	46.145	0.435	0.067	0.367	1.046	1.120
12.342	0.393	31.850	0.385	0.039	0.354	0.989	1.047
5.670	0.176	31.060	0.223	0.018	0.317	0.837	0.852
3.903	0.000	0.027	0.165	0.000	0.319	0.842	0.850
3.246	0.402	123.955	0.141	0.040	0.330	0.887	0.935

were collected from Intel Xeon X5482 processor. The evaluation results for the different algorithms used in this study are shown in table 5.4.

Table 5.4: Prediction accuracy results from 10 times 10-fold cross validation for predicting concurrent-run performance on Intel Xeon X5482 processor.

Algorithms	<i>Prediction Accuracy Metrics</i>				
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>	<i>RRSE %</i>	<i>RAE %</i>
LR	0.97	0.18	0.31	24.99	21.06
ANN	0.99	0.14	0.20	16.41	15.82
IBK	0.98	0.11	0.22	17.63	12.80
K*	0.99	0.08	0.20	15.72	9.73
M5'	0.99	0.09	0.17	13.95	10.66
SVM	0.97	0.15	0.33	26.82	17.52

The prediction accuracy metrics [112] obtained as shown in table 5.4, indicate that KStar (K*), Model Trees (M5') and K-nearest neighbors classifier (IBK) algorithms produce lower values of *Mean Absolute Error (MAE)*.

The time taken to build the models (for predicting concurrent-run performance) using train-set having 3025 instances of data is shown in table 5.5.

Table 5.5: Time taken to build regression models using train-set having 3025 instances from Intel Xeon X5482 processor for predicting concurrent-run performance.

Algorithms	Time (in seconds) taken to build model
LR	0.07
ANN	30.98
M5'	17.52
SVM	159.15

5.5 Assessing Transferability of Model for Predicting Concurrent-run Performance

We performed solo-run and concurrent-run experiments on other platforms based on Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors. The test-set data containing the program attributes and class variable i.e. concurrent-run performance were generated from the hardware performance counter data collected during the experiments. We tested the regression model

(trained by data from Intel Xeon X5482 processor) using test-set data from Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors.

5.5.1 Differences in Last Level Cache Organization on Processors used to Generate Train-set and Test-set Data

The processors used to generate the train-set data (i.e. Intel Xeon X5482) and test-set data (i.e. Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors) have differences in their memory hierarchy with respect to last level cache as mentioned in table 5.6.

Table 5.6: Last level cache related data of Intel Xeon X5482, Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors.

Processor	Cache levels	No of cores sharing last level cache	Size (MB) shared among the cores	Ways of associativity	Relation with upper level caches
Xeon X5482	2	2	6	24	Inclusive
Core2 6300	2	2	2	8	Inclusive
Phenom 9650	3	4	2	32	Exclusive
Phenom 8450	3	3	2	32	Exclusive

The size of train-set data collected for the pairs formed by SPEC cpu2006 benchmark programs on Intel Xeon X5482 processor was 55×55 (i.e. 3025). The number of instances of test-set data generated from Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors were 418, 164 and 153 respectively. These numbers of test instances were generated based on the availability of the platforms for performing experiments. We used prediction accuracy metrics to assess the transferability of trained regression models. The assessment for transferability of trained regression models across different cache organizations was performed for the three best performing algorithms in 10-fold cross validation i.e. KStar (K*), Model Trees (M5') and K-nearest neighbors classifier (IBK).

5.5.2 Prediction Accuracy Metrics of Trained Regression Models on Test Data-set

The prediction accuracy results for the test data from Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450 processors are shown in table 5.7, table 5.8 and table 5.9 respectively.

Table 5.7: Prediction accuracy of the trained model on test data from Intel Core2 6300 processor for predicting concurrent-run performance.

Algorithms	<i>Prediction Accuracy Metrics</i>		
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>
IBK	0.93	0.29	0.51
K*	0.95	0.20	0.33
M5'	0.93	0.17	0.35

Table 5.8: Prediction accuracy of the trained model on test data from AMD Phenom 9650 processor for predicting concurrent-run performance.

Algorithms	<i>Prediction Accuracy Metrics</i>		
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>
IBK	0.95	0.31	0.45
K*	0.89	0.25	0.72
M5'	0.98	0.23	0.30

Table 5.9: Prediction accuracy of the trained model on test data from AMD Phenom 8450 processor for predicting concurrent-run performance.

Algorithms	<i>Prediction Accuracy Metrics</i>		
	<i>C</i>	<i>MAE</i>	<i>RMSE</i>
IBK	0.94	0.26	0.41
K*	0.95	0.20	0.35
M5'	0.94	0.26	0.41

The values of prediction accuracy metrics obtained for the test data shown in table 5.7, table 5.8 and table 5.9 indicate that the regression model trained by data from Intel Xeon X5482 processor are reasonable for performing performance predictions on other processors i.e. Intel Core2 6300, AMD Phenom 9650 and AMD Phenom 8450.

The Model Trees (M5') algorithm produces least values of *Mean Absolute Error (MAE)* on two platforms. The next section describes application of the trained model for simulation of multicores.

5.6 Application of Machine Learning Based Performance Prediction for Multicore Simulation

The sharing of the resources by cores on multicore processors has spawned research in the domain of thread scheduling. It is the objective of recent research in thread scheduling to be aware of shared resource requirements of the running programs (threads). To this end, AKULA [54] is a recently developed tool-set that provides a platform for experimenting and developing thread scheduling algorithms on multicore processors.

AKULA tool-set has been developed to aid the researchers in narrowing down and rapidly exploring the design space of scheduling algorithm for multicore processors. AKULA provides user friendly APIs (Application Programming Interfaces) to prototype thread placement algorithms at user level and also provides facility to rapidly test the algorithms using multicore simulation provided by its *bootstrap* module. We propose application of machine learning based performance prediction to improve the multicore simulation provided by *bootstrap* module of AKULA tool-set. The next section provides overview of the AKULA tool-set along with description of the *bootstrap* module.

5.6.1 Overview of the AKULA Tool-set

The AKULA tool-set [54] helps the algorithm developers in quickly converting an idea for the scheduling algorithm into a working scheduling algorithm, which can then be rapidly evaluated. AKULA consists of three modules: *profiler*, *bootstrap* and *wrapper* modules. The framework provided by AKULA tool-set [54] is shown in figure 5.2. The framework facilitates prototyping of scheduling algorithms for multicores in user space. If the evaluation of a proposed scheduling algorithm gives good results then the developers can move to implementing it in kernel.

The *profiler* module is provided for collecting the performance data for workload consisting of a pair of programs. The performance data is collected for solo-run and concurrent-run (paired-run) of the programs on a real multi-

Framework provided by AKULA tool-set

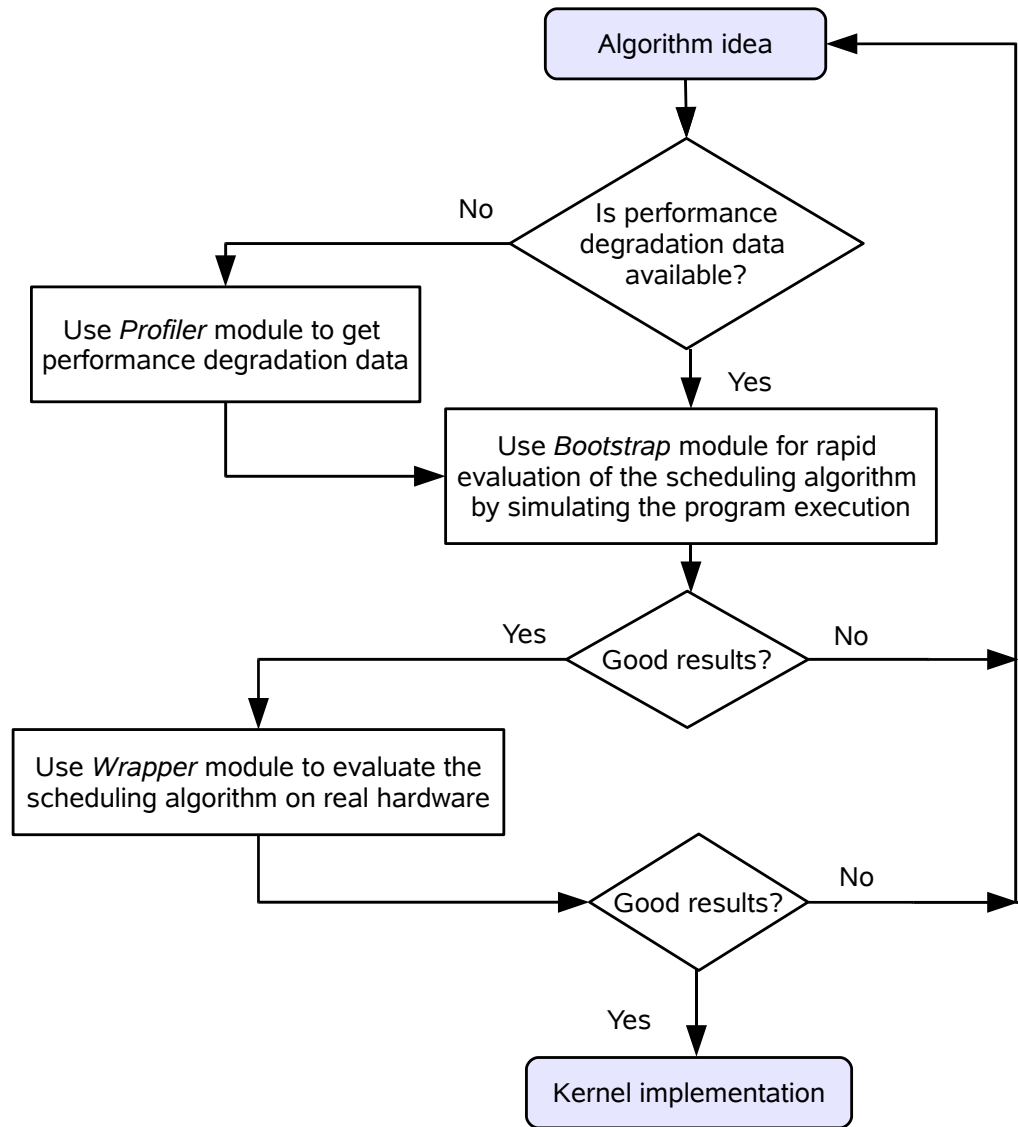


Figure 5.2: Framework provided by AKULA [54] tool-set for rapid prototyping and evaluation of scheduling algorithms for multicores in user space.

core system. The concurrent-run involves co-running the pair of programs on processor cores, which share the resources such as last level caches and bus.

Bootstrap module simulates the multicore system for rapid evaluation of proposed scheduling algorithms with the workloads for which the performance data have already been collected by the *profiler*.

If the rapid evaluation of a scheduling algorithm by *bootstrap* module shows fulfillment of the objective (e.g. improvement in performance), then the scheduling algorithm can be further tested on real multicore based systems using *wrapper* module. Otherwise the algorithm developer can further improvise the proposed algorithm or work on a different idea.

The key components of the *bootstrap* data are the solo-run and concurrent-run (paired-run) performance data. Solo-run data include execution time of the application measured on the real system, when the application runs alone without interference from the applications co-running on other core sharing the resources. The concurrent-run data is kept in the form of degradation matrix, which contains for a set of target applications, performance degradation values when each application is co-scheduled with every other application on the core sharing the resources. For example, in case of four programs A, B, C and D which run on the system where two cores share the resources, the performance data collected could be shown as table 5.10 and table 5.11.

Table 5.10: Example solo-run performance data used by AKULA *bootstrap* module for multicore simulation.

Program name	Solo-run Time (sec.)
A	100
B	150
C	175
D	200

Table 5.11: Example performance degradation matrix for concurrent-run, used by AKULA *bootstrap* module for multicore simulation.

Program name	Alone	With A	With B	With C	With D
A	1.00	0.75	0.50	0.98	0.99
B	1.00	0.60	0.30	0.95	0.97
C	1.00	0.99	0.98	1.00	1.00
D	1.00	1.00	1.00	1.00	1.00

The *bootstrap* module performs the simulated run of the threads in a given schedule by using the collected performance data (also referred as *bootstrap* data). Please note that we use terms processes and threads interchangeably for the purpose of scheduling. For every scheduling time interval (called tick) it calculates the progress of the threads using *bootstrap* data and the formula shown in equation 5.3.

$$Progress(A) = \frac{tick}{Solo(A)} \times degrad(A, Neighbour(A)) \quad (5.3)$$

In equation 5.3, $Progress(A)$ refers to the fraction of total work that a thread A completes in a given scheduling interval. The *tick* refers to the length of the scheduling time interval. $Solo(A)$ is the solo-run completion time of the thread A and $degrad(A, Neighbour(A))$ is the degradation that a thread A experiences due to sharing of the resources by co-running thread $Neighbour(A)$ on other core. Both $Solo(A)$ and $degrad(A, Neighbour(A))$ are obtained from previously collected *bootstrap* data. Example *bootstrap* data are shown in table 5.10 and table 5.11.

When a thread executes solo, the progress made by it is equal to length of the scheduling clock interval (i.e. *tick*) divided by the time to execute the entire thread. When the other co-running threads compete for the resources shared by the cores, the progress made by the thread needs to be scaled by the performance degradation observed by it in concurrent-run (paired-run) as shown in equation 5.3.

Since time needed to calculate the progress of a thread over a *tick* using above methodology is usually much shorter than the *tick* itself, this methodology allows very fast evaluations of the scheduling algorithms by simulated run of the threads.

5.6.2 Improving Multicore Simulation in AKULA

The model built using methodology proposed in this chapter can be used by *bootstrap* module of AKULA tool-set [54]. The trained model would replace the *bootstrap* data i.e. the performance degradation matrix used by *bootstrap* module of AKULA. The advantages gained by using machine learning trained model for multicore simulation in AKULA will be as follows:

1. The current *bootstrap* method in AKULA [54], performs the multicore simulation using data from performance degradation matrix, which works

for pair of applications for which such data have already been collected. It does not provide the facility of predicting the performance degradation for pairs of applications for which such data have not been collected.

The machine learning trained model has the ability to predict performance degradation for unknown pairs of applications, which is reflected in terms of results on prediction accuracy metrics of the trained model mentioned in section 5.4.

2. The current *bootstrap* method in AKULA [54], uses performance data (the solo performance and degradation matrix) for run of applications on a particular platform. Thus the simulation resulting from the use of such performance data, basically is representative of the platform from which the data were collected.

The machine learning trained model has transferability to other platforms also. The transferability aspect of the trained model has been discussed in section 5.5. It has been observed that model trained with data from Intel Xeon X5482 processor is reasonably transferable to other processor i.e. Intel Core2 6300. The two processors represent two different scenarios especially in terms of their last level cache organization. The transferability aspect could be attributed to generalization capability of the model trained by machine learning.

3. The AKULA [54] *bootstrap* works using the data from performance degradation matrix for pairs of applications. This method works for the case where the pair of applications is co-scheduled on two processor cores, which share the resources. The method becomes unwieldy when the number of processor cores sharing the resources is more than two. The dimensions of the performance degradation matrix grow, with the increase in number of cores sharing the resources. The architectural trends for multicore processors indicate towards growth in number of cores sharing the resources. For example, on Intel Core i5 and Intel Core i7 processors [55], four cores share the last level cache. On AMD Phenom 8450 processor three cores present on the chip share the last level cache, and on AMD Phenom 9650 processor four cores share the last level cache [56]. On Sun Ultrasparc T1 (Niagara1) eight cores each with four threads i.e. total 32 threads share the last level cache [15], and on Ultrasparc T2 (Niagara2) processor there are eight cores each with eight threads i.e. total 64 threads share the last level cache [131].

The machine learning trained model uses fractional solo-run programs attributes, which enables the offline trained model to predict the performance degradation even in the case when the number of cores sharing the memory hierarchy resources is more than two. We observed that the models (trained by the data from Intel Xeon X5482 processor) are reasonably transferable on other two processors viz. AMD Phenom 8450 and Phenom 9650. On Intel Xeon X5482 processor the last level cache is shared by two cores, while the number of cores sharing the last level cache on AMD Phenom 8450 and Phenom 9650 processors are three and four respectively.

4. The methodology proposed in AKULA [54], collects *bootstrap* data for one minute run of the application. The real life applications have quite larger runtimes, and also the applications have different phase behaviors during their entire run [110]. The shared resource requirements of applications change with the phases. The machine learning trained model having the prediction capability would be more amenable to take phase behaviors of the applications into account.

The machine learning trained model gives concurrent-run performance in terms of cycles per instruction (*concurrent_run_CPI*) by taking the solo-run attributes as inputs. Thus the calculation of the progress a thread A makes in a *tick*, (mentioned for AKULA in equation 5.3 on page 90) can now be done as shown in equation 5.4.

$$Progress(A) = \frac{tick}{Solo(A)} \times \frac{solo_run_CPI_A}{concurrent_run_CPI_A} \quad (5.4)$$

In equation 5.4, the *solo_run_CPI_A* is solo-run performance of application A , which is collected in solo-run experiment and is also used as one of the solo-run program attributes. The *concurrent_run_CPI_A* is concurrent-run performance of application A , predicted by the offline-trained model. The ratio of the two as mentioned in equation 5.4, basically is the degradation in performance of application A , when sharing the resources with another application/s co-running on other core/s of the processor.

5.7 Conclusions

This chapter described the methodology to build model to predict concurrent-run performance of programs running on multicore processors. The model takes

solo-run program attributes as inputs and predicts the concurrent-run performance in terms of cycles per instruction. The work presented could have potential application for creation of workloads for performance studies on multicore processors. The chapter also described one potential use of the offline-trained model to improve the multicore simulation done by *bootstrap* module of recently developed AKULA tool-set. The machine learning trained model will extend the *bootstrap* module's ability to predict degradation in performance due to co-runner interference, where previous performance data is not available for pairing /co-scheduling of applications. The approach proposed in chapter also allows greater scalability for simulation to consider variable number of processor cores sharing the resources.

Chapter 6

Conclusions and Future Work

The thesis contributes to the modeling of the multicore processor based computing systems, along with the development of methodologies to synthesize the models. The example use-cases of the synthesized models were described with prospective applications. The methodologies use machine learning for building the models.

The discipline of machine learning provides various techniques to automatically learn the complex behaviors. The models trained by machine learning can be used for making intelligent decisions. Programs co-running on different cores of a multicore processor, have complex interactions due to usage of shared memory hierarchy resources. The thesis describes an approach for modeling the performance implications of such interactions by applying machine learning techniques.

Section 6.1 provides summary of the contributions followed by conclusions and future work described in section 6.2 and section 6.3 respectively.

6.1 Summary of Contributions

Multicore has become the dominant processor architecture at present. The future generations of the processors will have greater degree of sharing of resources among the execution cores, due to continual growth in number of cores present on a single-chip.

The interference among the programs running on the processor cores, which share the resources remains a cause of concern due to performance degradations caused by it. Researchers have been proposing various solutions for this problem, yet it will take time for those solutions to appear in commodity production grade system. It is due to the reason that among majority of the proposed solutions, some of the solutions require extra support from the processor hardware

while others use complex models.

In this thesis, we proposed the use of machine learning techniques for synthesizing the models for program memory behavior characterization and performance prediction on multicore processor based systems. The work was performed on the existing commodity multicore processor based platforms. The main objective to use machine learning was to gather the knowledge about the interactions between the processor architecture and applications. Our focus of study was the interactions of the applications with memory hierarchy resources, which are shared among the cores of the multicore processors.

In summary, the thesis makes contributions towards advancement of performance oriented research on multicore processor based computing systems. The main contributions could be described by the following methodologies and associated prototypes developed as part of the work:

- ***Methodology to characterize program memory behavior on multicore processors:*** We used machine learning techniques to characterize the program behavior with respect to utilization of memory hierarchy resources, which are shared among the cores of the multicore processors. We proposed solo-run last level cache stress as a metric to characterize program memory-behavior on multicore processors. We also proposed the program attributes, which are used as inputs to predict the solo-run last level cache stress.
- ***Meta-scheduler for multicore processors:*** Subsequently we implemented a proof of concept meta-scheduler for multicore processors to demonstrate the application of the trained model. The off-line trained model was utilized in on-line manner to improve the process scheduling on multicore processors, so that the interference between programs co-running on the cores sharing the resources is mitigated. We observed performance improvement up to 76% for 4-cores and 54% for 8-cores with the meta-scheduler as compared to default operating system process scheduler. The approximate average cost of the model was (observed in the experiments) about 0.00075% of the total time (i.e. 7-8 cycles per million cycles).
- ***Methodology for performance prediction on multicore processors:*** We proposed the solo-run performance attributes of the programs, which are used by the model to predict the concurrent-run performance. The model is built by training the machine learning algorithms. Such performance prediction models could be useful for simulation studies of

process scheduling algorithms for multicore processors as well as creation of workloads for design and performance studies of memory hierarchy on multicores.

- ***Application of machine learning based performance prediction for multicore simulation:*** We proposed one of the potential use of the model, which was trained by machine learning to predict performance on multicores. The offline-trained model can be used to improve the multicore simulation provided by *bootstrap* module of AKULA [54] tool-set. AKULA [54] is a recently developed tool-set that provides a simulation platform for rapid prototyping and evaluation of thread scheduling algorithms on multicore processors. The machine learning trained model will enable the *bootstrap* module to predict the performance degradation for new workload schedules, while the current implementation works for workload pairings for which the performance degradation data is previously stored. The model also allows greater scalability to simulation for variable number of processor cores sharing the resources. As the offline-trained model will also be useful for performance prediction when the number of cores sharing the resources (e.g. last level caches) is more than two.

The methodologies and meta-scheduler developed as part of the work do not require any modifications in the existing hardware platforms as compared to some of the previous works. The meta-scheduler runs in user mode and guides the process scheduler of underlying operating system (linux on our experimental platform). It does not require any modifications in the operating system process scheduler. The approach presented in the thesis does not require modifications or recompilation of the application programs.

6.2 Conclusions

The work has demonstrated that the data driven methods such as machine learning techniques are helpful in performing performance studies on upcoming multicore computing systems. The three key lessons learnt during the work are:

- The machine learning techniques can be effectively used to build models for performance critical program characteristics on emerging multicore processors.

- Hardware Performance Monitoring Units (PMU) on the processors and the hardware performance counters provided by them can play an important role in on-line optimizations.
- Systems software mechanisms must continually evolve as the hardware evolves, in order to fully exploit the beneficial characteristics and minimize the performance bottlenecks of the upcoming systems.

This thesis shall serve as one of the preliminary proof-of-concept that techniques and algorithms from the machine learning and data mining domains can be applied to emerging multicore based systems to learn relevant knowledge about the resource utilization behavior and associated performance implications for programs running over those systems. It describes the methodology for data collection from real-world platforms and subsequent model building by applying machine learning techniques on that data. It also mentions example applications of the trained models. The methodology developed in the thesis shall be applicable to computer systems based on emerging multicore processors, as the upcoming multicores might witness increased sharing of resources among the processor cores. The lessons learnt in the context of a problem could be helpful in providing insights for applying machine learning to other related problems as well.

6.3 Future Work

We hope that the methodologies and mechanisms developed as part of the work create further research possibilities. Fellow researchers can use these initial mechanisms and ideas as a starting point for their own work and further use, improve and extend the base mechanisms as well as spawn or inspire new related ideas.

The work done in the thesis opens up some more directions for future work. In general, additional experiments could be planned on some more multicore based platforms to strengthen the work. Some of the directions for further work are mentioned as follows:

1. **Improving the models:** The models developed in the thesis consider program attributes as inputs. One of the model predicts the program memory behavior in terms of solo-run last level cache stress. The other model predicts the concurrent-run performance on multicores. There can be additional program attributes, which can improve the models. To

explore such possibility some additional experiments could be planned with variety of systems.

We used default values of the parameters for machine learning algorithms in weka [112] [113] machine learning workbench. The default settings for the parameters for algorithms work well for a wide variety of problems [114]. Further tuning of the parameters for the machine learning algorithms could be considered for the problems taken up in the thesis. In future, use of emerging state of the art machine learning techniques could also be investigated for building the models.

2. **Extending the meta-scheduler to additional systems:** The developed meta-scheduler could be extended to some more platforms. Current implementation of the meta-scheduler considers processor topology and solo-run last level cache stress of the programs for making its decisions. Experimentation with additional systems could provide possibility to consider some more metrics and factors for improving the process schedule.
3. **Extending the methodologies for emerging systems:** The methodologies for synthesizing the models could be extended to some more platforms. The focus of study in this thesis has been on memory hierarchy resources, which are shared among the cores of a multicore processor. The multicore processors considered in the study were homogeneous multicores. In future, performance study could be done over heterogeneous multicores as well as multithreaded-multicore processors. Implications associated with additional performance critical resources could be considered for further explorations.
4. **Exploring additional applications of the proposed methodologies:** We proposed methodologies to build models for program memory behavior characterization and performance prediction. Exploration for the additional applications of the methodologies could be done. Such studies will be helpful in further extensions and improvements of the methodologies. The performance prediction techniques could be used for creation of workloads for performance studies on memory hierarchies of the multicores as well as for developing the simulation tools.
5. **Consideration of additional workloads:** The work done in the thesis considered multiprogrammed workloads using applications from SPEC cpu2006 benchmark suite [74] [78]. Though the SPEC cpu2006 benchmark programs are realistic applications, widely used by research com-

munity as well as industry; workloads from other domains also could be considered. Such studies may be helpful in identifying additional issues.

6. **Consideration of program phase behavior of workloads:** The methodologies presented in the thesis for building the models, used performance counter data collected from complete run of the programs. The methodologies could be extended to consider the phase behavior [110] of the programs. The meta-scheduler described in the thesis does performance monitoring and optimizations in periodic manner (every 500 milliseconds). It could be made aware of the program phases, so that it gets activated accordingly.

7. **Standardization of hardware performance counters subsystem:** Performance counters are special registers provided by performance monitoring unit of the processors. They were originally created to help computer architects evaluate their designs. Their usage was primarily introspective, hence they had the lowest priority during development of processor architecture.

The work done in the thesis involves extensive use of the event data collected from hardware performance counters of the processors. However, the performance events available on the systems differ for different generations / families as well as make of the processors. There is a need for further research to come up with certain minimal set of events, which should be available on all the processors. It also entails standardization of the way to configure and measure the performance events as well as standardization of the software interface / API for doing the same. We used perfmon2 [122] interface for accessing the hardware performance counters. The linux kernel perf_events project [132] seems a step towards this direction.

8. **Research on performance counters:** This is bit orthogonal to our previous suggestion for standardization of hardware performance counter subsystem. Additional performance counters could be envisaged for providing the information on various performance aspects of the emerging platforms. These counters could provide information related with performance events of the system hardware as well as the software stack which runs over the hardware. For example, the information provided by these counters could be useful to improve policies implemented in various subsystems such as swapping, page replacement, disk scheduling etc.

in operating systems. The additional performance counters could have hard as well as soft implementations. The hard implementation refers to implementation at hardware level such as on the processor. The soft implementation refers to implementation at the software level such as inside the operating system. Recently launched linux kernel perf_events project [132] has considerations for counters at both hardware and software levels.

Apart from this, further research could also involve extensions and improvement of state of the art machine learning techniques for utilization in further optimizations of computing systems.

References

- [1] O. Herescu, B. Olszewski, and H. Hua, “Performance workloads characterization on POWER5 with simultaneous multi threading support,” in *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-8) in Conjunction with the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, pp. 1–9, February 2005.
- [2] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, “Performance of multithreaded chip multiprocessors and implications for operating system design,” in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 26–26, USENIX Association, 2005.
- [3] F. Faggin, J. Hoff, M.E., S. Mazor, and M. Shima, “The history of the 4004,” *Micro, IEEE*, vol. 16, no. 6, pp. 10–20, 1996.
- [4] G. Moore, “Progress in digital integrated electronics,” in *Electron Devices Meeting, 1975 International*, vol. 21, pp. 11–13, 1975.
- [5] G. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.
- [6] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobbs Journal*, vol. 30, no. 3, pp. 202–210, 2005.
- [7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, third ed., 2003.
- [8] D. Burger, J. R. Goodman, and A. Kägi, “Memory bandwidth limitations of future microprocessors,” in *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, (New York, NY, USA), pp. 78–89, ACM, 1996.

- [9] J. Huh, D. Burger, and S. W. Keckler, “Exploring the design space of future CMPs,” in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT ’01, (Washington, DC, USA), pp. 199–210, IEEE Computer Society, 2001.
- [10] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: challenges in and avenues for CMP scaling,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA ’09, (New York, NY, USA), pp. 371–382, ACM, 2009.
- [11] J. Laudon, “Performance/watt: the new server focus,” *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 5–13, 2005.
- [12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The tera computer system,” in *Proceedings of the 4th international conference on Supercomputing*, ICS ’90, (New York, NY, USA), pp. 1–6, ACM, 1990.
- [13] K. Kurihara, D. Chaiken, and A. Agarwal, “Latency tolerance through multithreading in large-scale multiprocessors,” in *Proceedings of International Symposium on Shared Memory Multiprocessing*, pp. 91–101, IPS Press, 1991.
- [14] J. Laudon, A. Gupta, and M. Horowitz, “Interleaving: A multithreading technique targeting multiprocessors and workstations,” in *In Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 308–318, 1994.
- [15] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [16] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, “Simultaneous multithreading: A platform for next-generation processors,” *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.
- [17] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A single-chip multiprocessor,” *Computer*, vol. 30, no. 9, pp. 79–85, 1997.
- [18] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “Processor power reduction via single-ISA heterogeneous multi-core architectures,” *IEEE Comput. Archit. Lett.*, vol. 2, no. 1, pp. 2–5, 2003.

- [19] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 503–519, 2007.
- [20] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, 2005.
- [21] D. Geer, "Industry trends: Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.
- [22] T. M. Conte and W. W. Hwu, "Benchmark characterization," *Computer*, vol. 24, no. 1, pp. 48–56, 1991.
- [23] L. K. John, P. Vasudevan, and J. Sabarinathan, "Workload characterization: Motivation, goals and methodology," in *Proceedings of the Workload Characterization: Methodology and Case Studies*, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 1998.
- [24] P. Bose, "Workload characterization: A key aspect of microarchitecture design," *IEEE Micro*, vol. 26, no. 2, pp. 5–6, 2006.
- [25] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, (New York, NY, USA), pp. 2–11, ACM, 1996.
- [26] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The stanford hydra cmp," *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.
- [27] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 282–293, 2000.
- [28] L. Codrescu, D. S. Wills, and J. Meindl, "Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications," *IEEE Trans. Comput.*, vol. 50, no. 1, pp. 67–82, 2001.
- [29] J. A. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 245–256, 2000.

- [30] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, 2002.
- [31] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, (Washington, DC, USA), pp. 26–34, IEEE Computer Society, 2003.
- [32] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, vol. 5, no. 1, pp. 1–13, 2001.
- [33] J. R. Bulpin and I. A. Pratt, "Multiprogramming performance of the pentium 4 with hyper-threading," in *Proceedings of the Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004) held at ISCA '04*, pp. 53–62, 2004.
- [34] J. R. Bulpin and I. A. Pratt, "Hyper-threading aware process scheduling heuristics," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2005.
- [35] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 system microarchitecture," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 505–521, 2005.
- [36] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. Res. Dev.*, vol. 46, no. 1, pp. 5–25, 2002.
- [37] Intel. Dual core era begins, PC makers start selling Intel-based PCs. Intel Press Release, Santa Clara, CA, Apr. 18, 2005.
- [38] AMD. AMD announces worlds first 64-bit, x86 multi-core processors for servers and workstations at second-anniversary celebration of AMD Opteron processor. AMD Press Release, New York, NY, Apr. 21, 2005.
- [39] Microsoft. The countdown begins: Xbox 360 to hit store shelves Nov. 22 in North America, Dec. 2 in Europe and Dec. 10 in Japan. Microsoft Press Release, Tokyo, Japan, Sept. 15, 2005.

- [40] Sun. Sun Microsystems marks new era in network computing with breakthrough CoolThreads technology - unveils high-performance, eco-responsible server line. Sun Press Release, New York, NY, Dec. 6, 2005.
- [41] Intel. New dual-core Intel Itanium 2 processor doubles performance, reduces power consumption: Aggressive growth in Itanium hardware and software solutions deliver mission critical computing freedom. Intel Press Release, Santa Clara, CA, Jul. 18, 2006.
- [42] IBM. IBM makes first Cell computer generally available: Cell-based BladeCenter system poised to boost performance for new applications in aerospace, oil & gas and medical industries. IBM Press Release, Armonk, NY, Sept. 12, 2006.
- [43] Sony. Playstation 3 launches on November 17, 2006 in North America. Sony Press Release, Los Angeles, CA, Nov. 17, 2006.
- [44] Intel. Intel ignites quad-core era: Worlds best microprocessor gets even better. Intel Press Release, Santa Clara, CA, Nov. 14, 2006.
- [45] Fujitsu. Fujitsu and Sun Microsystems set the standard for open systems computing with fastest, most reliable Solaris/SPARC servers: Co-developed SPARC Enterprise servers: To be comarketed by Fujitsu and Sun. Fujitsu Press Release, Tokyo, Japan, and Santa Clara, CA, Apr. 17, 2007.
- [46] IBM. IBM unleashes worlds fastest chip in powerful new computer: Processor doubles speed without adding to energy footprint, enabling customers to reduce electricity consumption by almost half; enough bandwidth to download entire iTunes catalog in 60 seconds. IBM Press Release, London, UK, May 21, 2007.
- [47] AMD. AMD introduces the worlds most advanced x86 processor, designed for the demanding datacenter. AMD Press Release, Sunnyvale, CA, Sept. 10, 2007.
- [48] Sun. Sun Microsystems and Fujitsu expand SPARC enterprise server line with first systems based on the UltraSPARC T2 processor. Sun Press Release, Las Vegas, NV, Oct. 7, 2007.
- [49] Intel. Intel high-end Xeon server processors raise performance bar. Intel Press Release, Santa Clara, CA, Sept. 15, 2008.

- [50] Fujitsu. Fujitsu and Sun unveil new entry-level server powered by the SPARC64 VII processor and the Solaris OS: New SPARC Enterprise M3000 server delivers enterprise performance and mission-critical RAS in ultra-dense footprint. Fujitsu Press Release, Tokyo, Japan, and Santa Clara, CA, Oct. 28, 2008.
- [51] AMD. New six-core AMD Opteron processor delivers up to thirty-four percent more performance per watt in exact same platform: Available five months ahead of schedule, Istanbul is an industry game-changer. AMD Press Release, Sunnyvale, CA, Jun. 1, 2009.
- [52] IBM. IBM unveils new POWER7 systems to manage increasingly data-intensive services: Unprecedented scale for emerging industry business models, from smart electrical grids to real-time analytics. IBM Press Release, Armonk, NY, Feb. 8, 2010.
- [53] Intel. New Intel Xeon processor pushes mission critical into the mainstream. Intel Press Release, Santa Clara, CA, Mar. 30, 2010.
- [54] 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*, PACT ’10, (New York, NY, USA), pp. 249–260, ACM, 2010.
- [55] Intel 64 and IA-32 Architectures Software Developer’s Manuals, <http://www.intel.com/products/processor/manuals>.
- [56] AMD-phenom-processor-model-numbers-feature-comparison. <http://www.amd.com/us/products/desktop/processors/phenom/Pages/AMD-phenom-processor-model-numbers-feature-comparison.aspx>.
- [57] Y. Xie and G. H. Loh, “Dynamic classification of program memory behaviors in CMPs,” in *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects in conjunction with the 35th International Symposium on Computer Architecture*, ISCA-35, pp. 1–9, 2008.
- [58] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 340–351, IEEE Computer Society, 2005.

- [59] N. Rafique, W. T. Lim, and M. Thottethodi, “Architectural support for operating system-driven cmp cache management,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT ’06, (New York, NY, USA), pp. 2–12, ACM, 2006.
- [60] G. E. Suh, S. Devadas, and L. Rudolph, “A new memory monitoring scheme for memory-aware scheduling and partitioning,” in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA ’02, (Washington, DC, USA), pp. 117–128, IEEE Computer Society, 2002.
- [61] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, “Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT ’06, (New York, NY, USA), pp. 13–22, ACM, 2006.
- [62] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 423–432, 2006.
- [63] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory,” *J. Supercomput.*, vol. 28, no. 1, pp. 7–26, 2004.
- [64] S. Kim, D. Chandra, and Y. Solihin, “Fair cache sharing and partitioning in a chip multiprocessor architecture,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’04, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2004.
- [65] J. Chang and G. S. Sohi, “Cooperative cache partitioning for chip multiprocessors,” in *Proceedings of the 21st annual international conference on Supercomputing*, ICS ’07, (New York, NY, USA), pp. 242–252, ACM, 2007.
- [66] J. Chang and G. S. Sohi, “Cooperative caching for chip multiprocessors,” in *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA ’06, (Washington, DC, USA), pp. 264–276, IEEE Computer Society, 2006.

- [67] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, “CacheScouts: Fine-grain monitoring of shared caches in cmp platforms,” in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT ’07, (Washington, DC, USA), pp. 339–352, IEEE Computer Society, 2007.
- [68] R. Iyer, “On modeling and analyzing cache hierarchies using CASPER,” in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pp. 182–187, Oct. 2003.
- [69] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir, “A case for integrated processor-cache partitioning in chip multiprocessors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, (New York, NY, USA), pp. 6:1–6:12, ACM, 2009.
- [70] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.
- [71] P. J. Denning, “Thrashing: its causes and prevention,” in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS ’68 (Fall, part I), (New York, NY, USA), pp. 915–922, ACM, 1968.
- [72] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, “Throughput-oriented scheduling on chip multithreading systems,” August 2004. Technical Report TR-17-04, Harvard University.
- [73] E. Berg and E. Hagersten, “Efficient data-locality analysis of long-running applications,” May 2004. Technical Report 2003-021, ISSN 1404-3203, Department of Information Technology, Uppsala University, Uppsala, Sweden.
- [74] Standard Performance Evaluation Corporation <http://www.spec.org>.
- [75] 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*, PACT ’07, (Washington, DC, USA), pp. 25–38, IEEE Computer Society, 2007.

- [76] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, “Using os observations to improve performance in multicore systems,” *Micro, IEEE*, vol. 28, no. 3, pp. 54–66, 2008.
- [77] M. T. Jones, “Inside the linux 2.6 completely fair scheduler.” <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler>.
- [78] C. D. Spradling, “SPEC cpu2006 benchmark tools,” *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.
- [79] M. Banikazemi, D. Poff, and B. Abali, “PAM: a novel performance/power aware meta-scheduler for multi-core systems,” in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC ’08, (Piscataway, NJ, USA), pp. 39:1–39:12, IEEE Press, 2008.
- [80] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS ’10, (New York, NY, USA), pp. 129–142, ACM, 2010.
- [81] 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*, PACT ’08, (New York, NY, USA), pp. 220–229, ACM, 2008.
- [82] J. Edmonds, “Maximum matching and a polyhedron with 0,1 vertices,” *J. of Res. the Nat. Bureau of Standards*, vol. 69 B, no. 1-2, pp. 125–130, 1965.
- [83] K. Tian, Y. Jiang, and X. Shen, “A study on optimally co-scheduling jobs of different lengths on chip multiprocessors,” in *Proceedings of the 6th ACM conference on Computing frontiers*, CF ’09, (New York, NY, USA), pp. 41–50, ACM, 2009.
- [84] P. Hart, N. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.
- [85] A. Snaveley and D. Tullsen, “Symbiotic jobscheduling for a simultaneous multithreading processor,” in *In Eighth International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, pp. 234–244, 2000.

- [86] S. Parekh, S. Eggers, H. Levy, and J. Lo, “Thread-Sensitive Scheduling for SMT Processors,” 2000. www.cs.washington.edu/research/smt/papers/threadScheduling.pdf.
- [87] 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*, CGO ’10, (New York, NY, USA), pp. 257–265, ACM, 2010.
- [88] S. Cho and L. Jin, “Managing distributed, shared L2 caches through os-level page allocation,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 455–468, IEEE Computer Society, 2006.
- [89] D. Tam, R. Azimi, 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*, pp. 1–8, 2007.
- [90] E. Berg and E. Hagersten, “Fast data-locality profiling of native execution,” in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS ’05, (New York, NY, USA), pp. 169–180, ACM, 2005.
- [91] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, “RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations,” in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS ’09, (New York, NY, USA), pp. 121–132, ACM, 2009.
- [92] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multicore cache management,” in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys ’09, (New York, NY, USA), pp. 89–102, ACM, 2009.
- [93] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.
- [94] Machine learning - Wikipedia, the free encyclopedia.

- [95] R. M. Yoo, H. Lee, K. Chow, and H. hsin S. Lee, “Constructing a non-linear model with neural networks for workload characterization,” in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 150–159, 2006.
- [96] K. Kishore and A. Negi, “Characterizing process execution behaviour using machine learning techniques,” in *Proceedings of HiPC International Conference, DpROM’4 workshop*, pp. 1–6, 2004.
- [97] A. Negi and K. Kishore, “Applying machine learning techniques to improve linux process scheduling,” in *TENCON 2005 IEEE Region 10*, pp. 1–6, 2005.
- [98] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. Doshi, and S. Abraham, “Using model trees for computer architecture performance analysis of software applications,” in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pp. 116–125, 2007.
- [99] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, “A case for machine learning to optimize multicore performance,” in *Proceedings of the First USENIX conference on Hot topics in parallelism, HotPar’09*, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2009.
- [100] A. Fedorova, D. Vengerov, and D. Doucette, “Operating system scheduling on heterogeneous core systems,” in *Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures, at PACT 2007*, 2007.
- [101] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, “Efficiently exploring architectural design spaces via predictive modeling,” in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, ASPLOS-XII*, (New York, NY, USA), pp. 195–206, ACM, 2006.
- [102] R. Bitirgen, E. Ipek, and J. F. Martinez, “Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach,” in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41*, (Washington, DC, USA), pp. 318–329, IEEE Computer Society, 2008.

- [103] J. F. Martinez and E. Ipek, “Dynamic multicore resource management: A machine learning approach,” *IEEE Micro*, vol. 29, no. 5, pp. 8–17, 2009.
- [104] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero, “Explaining dynamic cache partitioning speed ups,” *Computer Architecture Letters*, vol. 6, no. 1, pp. 1–4, 2007.
- [105] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 367–378, 2008.
- [106] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’05, (New York, NY, USA), pp. 190–200, ACM, 2005.
- [107] K. Hoste and L. Eeckhout, “Microarchitecture-independent workload characterization,” *Micro, IEEE*, vol. 27, no. 3, pp. 63–72, 2007.
- [108] C. Xu, X. Chen, R. Dick, and Z. 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, 2010.
- [109] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, “Performance prediction based on inherent program similarity,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT ’06, (New York, NY, USA), pp. 114–122, ACM, 2006.
- [110] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, “Discovering and exploiting program phases,” *IEEE Micro*, vol. 23, no. 6, pp. 84–93, 2003.
- [111] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from berkeley,” Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

- [112] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, second ed., 2005.
- [113] Weka 3: Data Mining Software in Java.
<http://www.cs.waikato.ac.nz/ml/weka/>.
- [114] M. Hall, “On weka mailing list: [wekalist] global blend in kstar algorithm,” Thu Mar 10 03:07:03 NZDT 2011.
<https://list.scms.waikato.ac.nz/mailman/htdig/wekalist/2011-March/025132.html>.
- [115] D. W. Aha, D. Kibler, and M. K. Albert, “Instance-based learning algorithms,” *Machine Learning*, vol. 6, no. 1, pp. 37–66, 1991.
- [116] J. G. Cleary and L. E. Trigg, “K*: An instance-based learner using an entropic distance measure,” in *In Proceedings of the 12th International Conference on Machine Learning*, pp. 108–114, Morgan Kaufmann, 1995.
- [117] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Chapman & Hall, New York, NY, 1984.
- [118] Y. Wang and I. H. Witten, “Inducing model trees for continuous classes,” in *In Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pp. 128–137, 1997.
- [119] J. R. Quinlan, “Learning with continuous classes,” in *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence (AI92)*, pp. 343–348, World Scientific, 1992.
- [120] A. J. Smola and B. Schölkopf, “A tutorial on support vector regression,” *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, 2004.
- [121] S. Shevade, S. Keerthi, C. Bhattacharyya, and K. Murthy, “Improvements to the smo algorithm for svm regression,” *Neural Networks, IEEE Transactions on*, vol. 11, no. 5, pp. 1188–1193, 2000.
- [122] S. Eranian, “Perfmon2: The hardware-based performance monitoring interface for linux,” in *Proceedings of the 2006 Linux Symposium*, pp. 269–288, 2006.
- [123] R. Kohavi, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” in *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, (San Francisco, CA, USA), pp. 1137–1143, Morgan Kaufmann Publishers Inc., 1995.

- [124] R. R. Bouckaert and E. Frank, “Evaluating the replicability of significance tests for comparing learning algorithms,” in *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004*, pp. 3–12, Springer, 2004.
- [125] The Linux Kernel Archives. <http://kernel.org>.
- [126] Advanced Micro Devices, Inc. BIOS and Kernel Developers Guide (BKDG) For AMD Family 10h Processors, <http://developer.amd.com/documentation/guides/pages/default.aspx>.
- [127] J. K. Rai, A. Negi, R. Wankar, and K. D. Nayak, “Performance prediction on multi-core processors,” in *Proceedings of the 2010 International Conference on Computational Intelligence and Communication Networks, CICN '10*, (Washington, DC, USA), pp. 633–637, IEEE Computer Society, Nov 26–28 2010.
- [128] J. K. Rai, A. Negi, and R. Wankar, “Using machine learning techniques for performance prediction on multi-cores,” *International Journal of Grid and High Performance Computing (IJGHPC)*, vol. 3, no. 4, pp. 14–28, 2011.
- [129] S. Eranian, “Re: [perfmon2] CORE 0 SELECT Umask for L3 events on AMD F10h,” 30 May 2011. <http://permalink.gmane.org/gmane.comp.linux.perfmon2.devel/3014>.
- [130] Advanced Micro Devices Inc., “Revision Guide for AMD Family 10h Processors 41322 Rev. 3.82, Available online: support.amd.com/us/Processor-TechDocs/41322.pdf,” February 2011. L3 Cache Performance Events May Not Reliably Track Processor Core.
- [131] “Niagara2: A highly threaded server-on-a-chip.” <http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf>.
- [132] Performance Counters for Linux Wiki. <https://perf.wiki.kernel.org>.
- [133] perfmon2: The hardware-based performance monitoring interface for linux. <http://perfmon2.sourceforge.net>.
- [134] Pfmmon user guide. http://perfmon2.sourceforge.net/pfmmon_usersguide.html.

Appendix A

Perfmon2 Interface

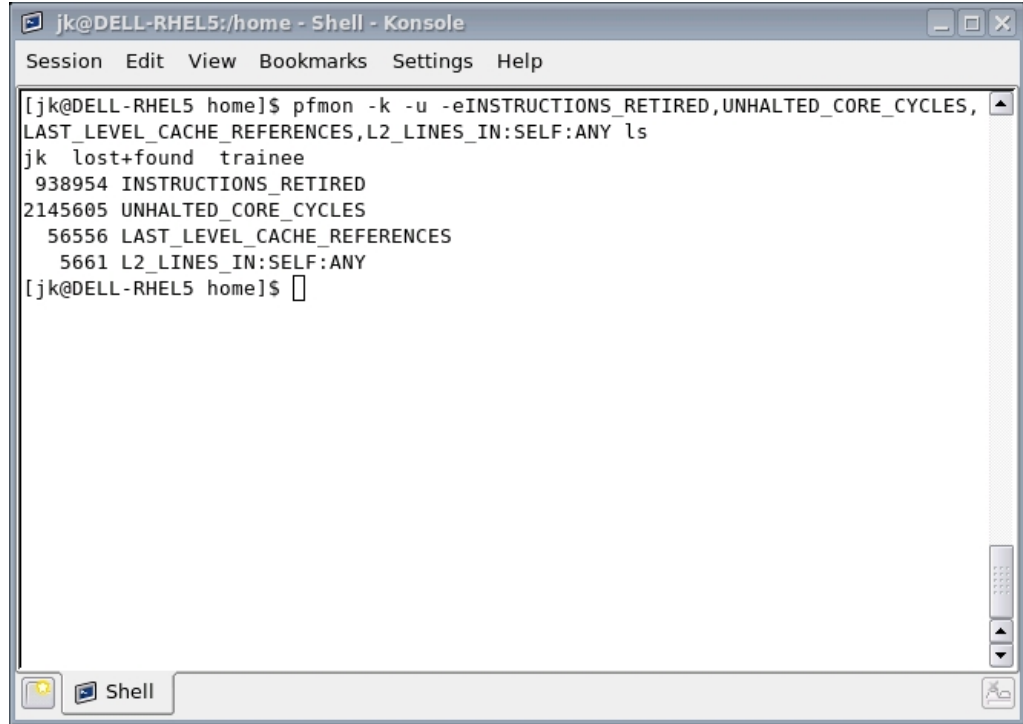
Perfmon2 [122] is an interface developed for monitoring the performance of programs running on computing systems. This interface provides APIs and tool to access the hardware performance counters of the processors from user-mode. The perfmon2 interface has been developed for linux kernel version 2.6, and is available on the web [133] for download. The interface is uniform across all hardware platforms, i.e. it offers the same level of software functionality on each platform. The nature of the captured data depends solely on the capabilities of the underlying hardware. Performance monitoring using hardware performance counters of the processor does not require modification or instrumentation or recompilation of the program to be monitored.

The perfmon2 distribution comes with three software components as mentioned below, which can be downloaded from the perfmon2 project site on web [133].

- Patch of perfmon2 interface for linux kernel: This patch is to be applied on linux kernel source. The linux source could be downloaded from linux kernel project site on web [125]. After applying the patch the linux kernel need to be compiled and installed on the system. Before compiling the kernel, the hardware performance monitoring related options need to be enabled via `make menuconfig` command.
- Library for perfmon2, libpfm: This is a user space library. Once the system is booted with perfmon2 enabled kernel as mentioned in previous step, the library can be built and installed on it.
- Pfm tool for monitoring: After installing the libpfm as above, the Pfm tool can be built and installed on the system.

The output of Pfm utility on Intel quad-core Xeon X5482 processor based

experimental platform is shown in figure A.1. The operating system on the

A screenshot of a terminal window titled "jk@DELL-RHEL5:/home - Shell - Konsole". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal shows a command prompt "[jk@DELL-RHEL5 home]\$ pfmon -k -u -eINSTRUCTIONS_RETIRED,UNHALTED_CORE_CYCLES, LAST_LEVEL_CACHE_REFERENCES,L2_LINES_IN:SELF:ANY ls". The output of the command is displayed as follows:

```
jk lost+found trainee
938954 INSTRUCTIONS_RETIRED
2145605 UNHALTED_CORE_CYCLES
56556 LAST_LEVEL_CACHE_REFERENCES
5661 L2_LINES_IN:SELF:ANY
[jk@DELL-RHEL5 home]$
```

The terminal window also features a status bar at the bottom with a "Shell" icon and a scroll bar on the right side.

Figure A.1: Output of Pfmon session on Intel quad-core Xeon X5482 processor based experimental platform. The output shows counts of four events for command `ls`.

platform is linux-2.6.30 kernel with perfmon2 interface [122]. The output is shown for command `ls`. The output shows counts of four events listed below:

- INSTRUCTIONS_RETIRED: It gives number of instructions retired.
- UNHALTED_CORE_CYCLES: It gives number of cycles consumed.
- LAST_LEVEL_CACHE_REFERENCES: It gives number of references to last level (L2) cache.
- L2_LINES_IN_SELF_ANY: It gives number of misses and prefetches to last level (L2) cache.

The events which can be measured on a processor are described in processor manuals [55]. The information about the syntax of command-lines for using the Pfmon tool is available on perfmon2 project site in the form of pfmon user-guide [134].

Appendix B

WEKA Machine Learning Workbench

WEKA (Waikato Environment for Knowledge Analysis) [112] is a suite of machine learning algorithms. It was developed at the University of Waikato, New Zealand. The algorithms can either be applied directly to a data-set or called from Java code. WEKA contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization.

The WEKA distribution can be downloaded from the WEKA project site on web [113]. It provides mainly four types of interfaces – Explorer, Experimenter, KnowledgeFlow and Commandline. The WEKA manual provided with the distribution describes the utility of the interfaces.

The example output of WEKA Explorer is shown in figure B.1. The output shows prediction accuracy results of M5P (i.e. M5') algorithm for predicting solo-run last level cache stress. The train-data-set has been generated from hardware performance counter data collected from Intel Xeon X5482 processor based experimental platform (specifications described in table 3.3 on page 45)). The test-data-set has been generated from hardware performance counter data collected from Intel Core2 6300 processor based experimental platform (specifications described in table 3.4 on page 45).

The **Attributes:** 5 shown in Classifier output panel of the figure B.1, refers to the five-tuple data instance i.e. four attributes plus the class variable solo-run last level cache stress (mentioned at page 51).

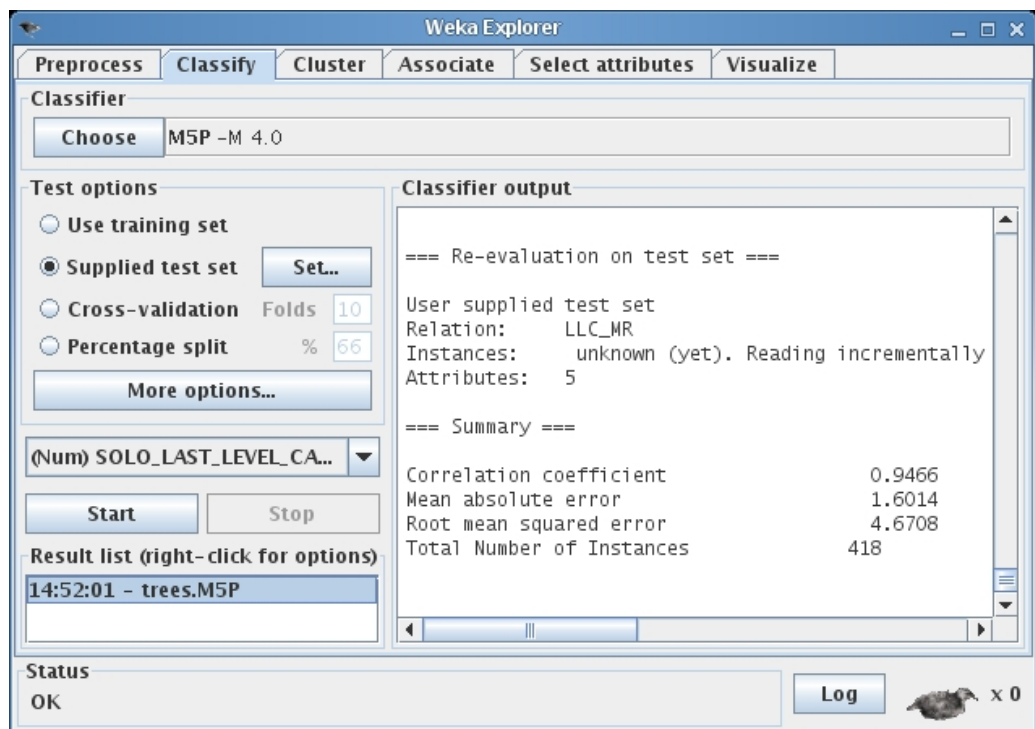


Figure B.1: Output of WEKA Explorer. The output shows prediction accuracy results of M5P algorithm for predicting solo-run last level cache stress. The train-data-set has been generated from hardware performance counter data collected from Intel Xeon X5482 processor based experimental platform. The test-data-set has been generated from hardware performance counter data collected from Intel Core2 6300 processor based experimental platform.

Synopsis
of the PhD thesis on

**Empirical Performance Studies on
Implications of Shared Memory Hierarchy
Resources in Multicore Computing Systems**

Submitted by
Jitendra Kumar Rai
Reg. No. 06MCPC04

for the degree of
Doctor of Philosophy
in
Computer Science

Under the Guidance of
Dr. Atul Negi
Dr. Rajeev Wankar



Submitted to the
Department of Computer & Information Sciences
University of Hyderabad
Hyderabad - 500046 Andhra Pradesh INDIA
October 2012

Abstract

Multicore processor architectures evolved and became dominant in recent years. It is mainly due to the diminishing returns from the efforts to increase the performance with single execution core on the chip. However multicore architectures involve sharing of resources especially the memory hierarchy resources such as processor caches, prefetchers and bus among the cores. This sharing may cause performance degradation of the programs co-running on multicore processors as compared to their solo run performance.

In the literature attempts to solve this problem proposed analytical models, which are very complex and difficult for use in real systems. Many of the previous works proposed specialized support from processor hardware, which may take time to become available in future generations of multicore processors.

It is observed that programs running simultaneously on different cores of a multicore processor, have complex interactions due to the use of shared memory hierarchy resources. In this work, we take up a unique approach for modeling the performance implications of such interactions by applying machine learning techniques. Machine learning techniques focus on methods that learn to recognize complex patterns from data. Here we give emphasis to empirical model building.

The work demonstrates the application of machine learning techniques to build models for characterization and performance prediction on multicore processor based systems. By characterization, we mean to characterize program memory behavior especially with respect to utilization of shared caches on multicore processors. The model developed for the program memory behavior characterization is later-on used for improving the process scheduling (i.e. CPU scheduling) on multicore processors. We also developed methodology to build a model that predicts performance on multicore processors. Such methodologies can further be utilized in performance oriented research on multicores. We describe the application of the built model for improving the multicore simulation in the AKULA tool-set. AKULA has been recently developed by researchers for rapid prototyping and evaluation of scheduling algorithms for multicore processors.

The efficacy of the developed methodologies to build the models, was validated using existing commodity multicore processor based systems. We observed that machine learning techniques are helpful in performance related studies on multicore processor based computing systems. We observed performance improvement up to 76% for 4-cores and 54% for 8-cores as compared to default linux kernel process scheduler on our experimental multicore platform by improving process scheduling. The process scheduling was improved by utilizing the model developed using machine learning techniques, so that the interference among the co-running programs due to usage of resources shared

among the cores is mitigated. The approximate average cost of the model was about 0.00075% of the total time (i.e. 7-8 cycles per million cycles), as observed in the experiments. Thus here we demonstrate that machine learning methods have been effective in building program performance models on existing commodity multicore based platforms.

1 Introduction

Over the past four decades, the microprocessor industry has seen consistent gains in application performance as a result of multiplicative effect of growth in transistor count and higher clock frequencies. The growth in transistor count has been the result of Moore's law. Gordon Moore predicted in the year 1975, that the number of transistors on an integrated circuit would double every two years [1]. The three main areas, which contributed to performance gains in the past are clock speed, execution optimization and cache [2]. Increasing the processor clock speed resulted in getting more cycles, which increased the speed at which the CPU performed the work. Optimizing execution flow is about doing more work per cycle. It includes having more powerful instructions as well as various other optimizations like pipelining, branch prediction, making the pipelines deep, having superscalar architectures and using out of order processing [3]. The performance scaling in the single core processors largely through increasing clock speed and by reliance on a single thread of control to find instruction-level parallelism has almost touched its limit. As the chip geometries shrink and clock speeds rise, the transistor leakage current increases, which leads to excessive power consumption and heat. The advantages of increased clock speed are also negated by memory latency, as the memory access speeds are not scaling on par with processor clock speeds. There is large and growing mismatch between the processor (CPU) and off-chip main memory in terms of speed as well as bandwidth. Many researchers have referred to this problem as "memory / bandwidth wall" in their works [4] [5] [6].

Another paradigm emerged to improve utilization of CPU resources by leveraging on thread and process level parallelism. A single physical processor can have one or multiple cores and each core can have one or multiple hardware threads. A single core processor with multiple hardware threads is called multithreaded processor while a multicore processor having multiple hardware threads per core is also called multicore-multithreaded or chip-multithreaded processor. The multicore or chip-multiprocessing architectures provide a way to scale the performance, while keeping the heat dissipation and power consumption under limit [7].

There have been several research projects to explore multi-core architecture like

Hydra [8], Piranha [9] and Atlas [10]. These projects explored various issues such as microarchitectural design, compiler support and speculative execution of user-level applications. In the past, performance studies have been done on simulators to analyze the operating system behavior in the presence of multiple hardware threads on a single processor [11]. Intel introduced hyperthreading (HT) on Xeon processor [12], which is an implementation of simultaneous multi-threading (SMT). On hyperthreaded Xeon processor, a single core supported two hardware threads. Hyperthreading makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors. Initial performance analysis have been done by Tuck and Tullsen [13] over hyperthreaded Intel Pentium 4 processor [14]. The focus of the study was to understand its performance and the underlying reasons behind that performance. Bulpin and Pratt [15] measured the multiprogramming performance of Intel Pentium 4 processor with hyperthreading and confirmed the findings of previous study done by Tuck and Tullsen [13]. They observed the mutual effect of processes simultaneously executing on the Intel Pentium 4 processor and found that many performance results can be explained by considering cache miss rates and resource requirement heterogeneity of those processes. The general rule of thumb derived from the study was that threads with high cache miss rates can have a detrimental effect on simultaneously executing threads. In a later work Bulpin and Pratt [16] proposed process scheduling heuristics for hyper-threaded processor, so that pathological combinations of workloads that can give a poor system-throughput could be avoided.

Among studies on multicore processors, Herescu et al. [17] performed performance workload characterization on platform based on IBM POWER5 [18] processor. IBM POWER5 is a dual core processor with simultaneous multithreading support. Each core on the processor supports two hardware threads. They observed some performance impacts due to shared resources, such as caches, translation look-aside buffers (TLBs) and branch prediction hardware. Fedorova et al. [19] also made observation that the sharing of memory hierarchy on the processor such as last level caches among the cores may cause the co-running programs to suffer with performance degradations.

Multicore processors have introduced sharing of resources among multiple execution cores present on the processor chip. Resources shared among the execution cores include on-chip caches, hardware prefetchers and system bus. In contrast, in the previous generation single-core processors the aforementioned resources were private for the execution core. Previous performance studies, mainly by Bulpin and Pratt [15], Herescu et al. [17] and Fedorova et al. [19] identified processor memory hierarchy resources shared

among the cores / threads of the processors as one of the performance critical resource. Sharing of the resources among the cores of the multicore processors causes co-running programs to interfere with each-other. The mutual interference among the co-running programs may cause them to suffer with performance degradations. Thus the multicore architecture poses additional performance issues that need to be addressed for effective utilization of the systems.

At present the multicore processors have become the driving engine for variety of computing systems such as desktops, servers, game consoles as well as embedded systems. The eminent ubiquity of the multicore processors indicates that multicore is going to be the dominant architecture of future. It makes imperative for us to look into performance issues arising due to interactions of multicore based systems and software. In next section we mention the research objectives and the problem statement for the thesis.

2 Research Objectives and Problem Statement

We aim to come up with the models, which can address the performance issues caused due to sharing of memory hierarchy resources among the cores of the multicore processors. The processor memory hierarchy resources include caches, prefetchers and memory bus.

The major objectives of the thesis are to:

- Investigate the performance issues due to sharing of the processor memory hierarchy resources (such as processor caches, prefetchers and bus) among the cores of the multicore processors.
- Review the existing models for memory behavior of programs and associated performance aspects of the multicore processors.
- Propose models for program memory behavior and associated performance aspects of the multicore processors.
- Suggest prospective applications of the proposed models towards solving the performance issues of multicore processors.

The thesis addresses the performance issues of multicore processor based computing systems with focus on program behavior with respect to utilization of the memory hierarchy resources, which are shared among the cores of the processor. The following are the key concerns addressed in the thesis:

- How to develop a model to characterize the program memory behavior on multicore processors.
- How to devise a mechanism for application of model developed in previous step so that the interference among co-running programs due to usage of memory hierarchy resources shared among the cores is mitigated.
- How to develop a model for predicting the performance degradations caused due to sharing of the processor memory hierarchy resources among the co-running programs.
- A prospective application of the model developed for predicting the performance degradations.

Overall the work aims at modeling the performance aspects of multicore processor based computing systems and prospective applications of the developed models. In order to model these performance aspects, memory hierarchy resources shared among the cores of the processors have been considered.

3 Research Contributions

The domain of machine learning provides various techniques which can be used to build systems for characterizing complex phenomena. Programs simultaneously running on different cores of a multicore processor, have complex interactions due to usage of shared memory hierarchy resources. The thesis presents a unique approach for modeling the performance implications of such interactions by applying machine learning techniques.

The contributions that stem from our research are demonstrated by the publications generated from the thesis. The main contributions from the thesis could be summarized as follows:

- **Methodology to build model to characterize program memory behavior on multicore processors:** We proposed the methodology to build model to characterize program memory behavior on multicore processors. The methodology includes proposal of program attributes, using which the program memory behavior can be predicted in terms of solo-run last level cache stress. It involves use of machine learning techniques to capture the knowledge about processor memory hierarchy resource utilization behavior of running programs. The off-line trained model could be used later for guiding the system policies to mitigate the interfer-

ence among the co-running programs due to usage of memory hierarchy resources shared among the cores.

- **Meta-scheduler for multicore processors:** We implemented a proof of concept meta-scheduler as an example application of the model developed for program memory behavior characterization. The meta-scheduler runs in user space and guides the process (CPU) scheduling decisions made by underlying operating system process scheduler so that the interference among the programs co-running on multicores is mitigated. We observed performance improvement up to 76% for 4-cores and 54% for 8-cores as compared to default linux kernel process scheduler on our Intel quad-core Xeon X5482 processor based platform by improving process scheduling. The approximate average cost of the model was about 0.00075% of the total time (i.e. 7-8 cycles per million cycles), as observed in the experiments. The meta-scheduler does not require modifications in process scheduler of the operating system running on the platform.
- **Methodology to build model for performance prediction on multicore processors:** We also proposed the methodology to build model to predict the performance on multicore processors. The model takes the proposed solo-run program attributes as inputs and predicts the concurrent-run performance of the programs on multicores. The concurrent-run involves interference among the program and other programs co-running on cores, which share the memory hierarchy resources with the first program. Such models and techniques could be used for further performance oriented research on multicores as well as simulation of multicores.
- **Application of performance prediction model for simulation of multicore processors:** We also propose the prospective application of the model, which was developed for performance prediction in previous step. We describe the use of the model for simulation of multicores in AKULA tool-set [20], which was recently developed by Zhuravlev et al. for rapid prototyping and evaluation of scheduling algorithms for multicore processors. The use of machine learning based model adds the performance predictability to enable the multicore simulation for workload combinations for which the concurrent-run performance data is not available. The approach proposed in the thesis also supports simulation for processors having variable number of processor cores sharing the resources. For example, the number of cores sharing last level caches are four on Intel Xeon E5630 [21], three on AMD Phenom 8450 [22] and four on AMD Phenom 9650 [22] processors.

The models and the prototype meta-scheduler were developed and demonstrated on the existing commodity multicore processor based systems. It may take some time for some of the previous works done on simulators (Xie and Loh [23], Chandra et al. [24], Rafique et al. [25], Suh et al. [26] and Hsu et al. [27]) to become applicable on real systems. We used machine learning techniques to develop the models by capturing the knowledge about the interactions of the applications and the processor architecture. The focus of studies have been on interactions of programs related with processor memory hierarchy resources, which are shared among the cores present on multicore processors. The use of machine learning requires training the algorithms to synthesize the models, on the other hand some of the analytical models proposed in previous works [24] [19] are fairly involved for application in real systems. The methodologies and mechanisms presented in the thesis do not require specialized support from hardware as proposed in some of the previous works by Xie and Loh [23], Chandra et al. [24], Rafique et al. [25], Suh et al. [26], Hsu et al. [27] and Qureshi and Patt [28]. The work also does not require any modifications or recompilations of the applications. We provide experimental evidence that the developed methodologies using machine learning techniques can be utilized to achieve performance gains. The methodologies developed as part of our work create further research possibilities. Fellow researchers can use these methodologies as initial point, for using, improving and extending the ideas.

4 Organization of the Thesis

The thesis is organized into six chapters as mentioned below. Thesis organization is also shown in figure 1.

- **Chapter 1: Introduction.** The chapter gives an overview of evolution of processor architecture and associated performance studies, which provided the motivation behind the work carried out in the thesis. It also mentions the research objectives of the work.
- **Chapter 2: Review of Performance Studies and Related Background.** This chapter provides the review of the previous studies related with performance issues of multicore based computing systems, with specific emphasis on sharing of processor memory hierarchy resources among the cores present on the processors. It also gives an overview of machine learning and some of its applications in computer systems research.
- **Chapter 3: Characterization of Program Memory Behavior.** This chap-

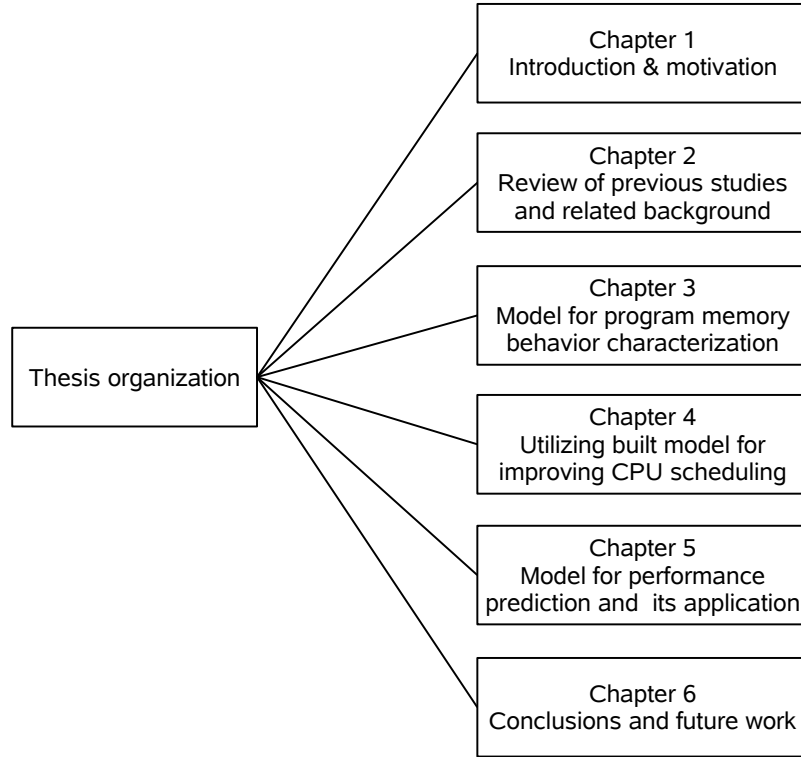


Figure 1: Thesis organization.

ter presents the methodology for characterizing the program memory behavior on multicore processors. The chapter also provides brief description of machine learning algorithms as well as experimental platforms used in the study. It describes the program attributes and the experimental method to gather the data from a multicore processor based platform to generate the training data-set. It also describes the prediction accuracy results of the trained model and its transferability over other experimental multicore platform.

- **Chapter 4: Improving Process Scheduling.** This chapter describes a meta-scheduler as an example application of the model built for program memory behavior characterization. It also provides the results on improvement in performance, observed with the meta-scheduler along with details of experimental setup.
- **Chapter 5: Performance Prediction.** This chapter describes the methodology to build model for performance prediction on multicore processors along with experimental setup and results. It includes the proposed solo-run program attributes, which the model takes as inputs to predict the concurrent-run performance. It also describes the prospective application of the model for simulation of multicores.

- **Chapter 6: Conclusions and Future Work.** This chapter summarizes the major contributions of the work and mentions the observations. It also highlights the future research directions.

Overall the thesis covers the modeling of the performance aspects of multicore processors with focus on sharing of processor memory hierarchy resources among the cores present on the multicore processors. The models were built by training the machine learning algorithms. The applicative aspects of the research reported in the thesis are reflected in the prospective use of the developed models for – (i) improving process scheduling on multicores and (ii) simulation of multicores.

5 Research Publications

Journals:

- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Characterizing L2 Cache behavior of Programs on Multi-core Processors: Regression Models and Their Transferability”, *International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)*, ISSN: 2150-7988 vol. 2 (2010), pp. 212–221. <http://www.mirlabs.org/ijcism>.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “A Machine Learning based Meta-Scheduler for Multi-core Processors”, *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, ISSN: 1947-9220 vol. 1, no. 4, (2010), IGI-Global, USA, 2010, pp. 46–59.

Also appeared in *Machine Learning: Concepts, Methodologies, Tools and Applications*, ed. Information Resources Management Association, USA, 2012, pp. 522–534, doi:10.4018/978-1-60960-818-7.ch311.

- J. K. Rai, A. Negi and R. Wankar, “Using Machine Learning Techniques for Performance Prediction on Multi-cores”, *International Journal of Grid and High Performance Computing (IJGHPC)*, ISSN: 1938-0259 vol. 3, no. 4, (2011), IGI-Global, USA, 2011, pp. 14–28.

Also in *Applications and Developments in Grid, Cloud, and High Performance Computing*, ed. Emmanuel Udoh, IGI-Global, USA, 2013, pp. 259–273, doi: 10.4018/978-1-4666-2065-0.ch017.

Conferences:

- J. K. Rai, A. Negi, R. Wankar and K.D. Nayak, “Using Machine Learning to Characterize L2 Cache behavior of Programs on Multicore Processors”, in *Proceedings of 2009 International Conference on Artificial Intelligence and Pattern Recognition (AIPR-09)*, Orlando, Florida, USA July 13-16, 2009, pp. 301–306.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “On Prediction Accuracy of Machine Learning Algorithms for Characterizing Shared L2 Cache behavior of Programs on Multicore Processors”, in *Proceedings of The 2009 IEEE International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN2009)*, Indore, India, July 23-25, 2009, pp. 213–219.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Characterizing L2 Cache behavior of Programs on Multi-core Processors: Regression Models and their Transferability”, in *Proceedings of The Eighth IEEE International conference on Computer Information Systems and Industrial Management Applications (CISIM 2009), held in conjunction with World Congress on Nature Biologically Inspired Computing 2009, (NaBIC 2009)*, Coimbatore, India, Dec. 09-11, 2009, pp. 1673–1676.
- J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, “Performance Prediction on Multi-core Processors”, in *Proceedings of The IEEE 2010 International Conference on Computational Intelligence, Communication Systems and Networks, (CICN 2010)*, Bhopal, India, Nov. 26-28, 2010, pp. 633–637.
- J. K. Rai, A. Negi and R. Wankar, “Machine Learning Based Performance Prediction for Multi-core Simulation”, in *Proceedings of The 5th Multi-Disciplinary International Workshop on Artificial Intelligence MIWAI 2011*, Hyderabad, India, Dec. 07-09, 2011, LNAI 7080, Springer-Verlag Berlin Heidelberg, pp. 236–247.

References

- [1] G. Moore, “Progress in digital integrated electronics,” in *Electron Devices Meeting, 1975 International*, vol. 21, pp. 11–13, 1975.
- [2] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobbs Journal*, vol. 30, no. 3, pp. 202–210, 2005.

- [3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, third ed., 2003.
- [4] D. Burger, J. R. Goodman, and A. Kägi, “Memory bandwidth limitations of future microprocessors,” in *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA ’96, (New York, NY, USA), pp. 78–89, ACM, 1996.
- [5] J. Huh, D. Burger, and S. W. Keckler, “Exploring the design space of future CMPs,” in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT ’01, (Washington, DC, USA), pp. 199–210, IEEE Computer Society, 2001.
- [6] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: challenges in and avenues for CMP scaling,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA ’09, (New York, NY, USA), pp. 371–382, ACM, 2009.
- [7] J. Laudon, “Performance/watt: the new server focus,” *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 5–13, 2005.
- [8] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, “The stanford hydra cmp,” *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.
- [9] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: a scalable architecture based on single-chip multiprocessing,” *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 282–293, 2000.
- [10] L. Codrescu, D. S. Wills, and J. Meindl, “Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications,” *IEEE Trans. Comput.*, vol. 50, no. 1, pp. 67–82, 2001.
- [11] J. A. Redstone, S. J. Eggers, and H. M. Levy, “An analysis of operating system behavior on a simultaneous multithreaded architecture,” *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 245–256, 2000.
- [12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, “Hyper-threading technology architecture and microarchitecture,” *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, 2002.

- [13] N. Tuck and D. M. Tullsen, “Initial observations of the simultaneous multithreading pentium 4 processor,” in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’03, (Washington, DC, USA), pp. 26–34, IEEE Computer Society, 2003.
- [14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousel, “The microarchitecture of the pentium 4 processor,” *Intel Technology Journal*, vol. 5, no. 1, pp. 1–13, 2001.
- [15] J. R. Bulpin and I. A. Pratt, “Multiprogramming performance of the pentium 4 with hyper-threading,” in *Proceedings of the Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004) held at ISCA’04*, pp. 53–62, 2004.
- [16] J. R. Bulpin and I. A. Pratt, “Hyper-threading aware process scheduling heuristics,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC ’05, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2005.
- [17] O. Herescu, B. Olszewski, and H. Hua, “Performance workloads characterization on POWER5 with simultaneous multi threading support,” in *Eighth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-8) in Conjunction with the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, pp. 1–9, February 2005.
- [18] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, “POWER5 system microarchitecture,” *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 505–521, 2005.
- [19] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, “Performance of multi-threaded chip multiprocessors and implications for operating system design,” in *ATEC ’05: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 26–26, USENIX Association, 2005.
- [20] 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*, PACT ’10, (New York, NY, USA), pp. 249–260, ACM, 2010.
- [21] Intel 64 and IA-32 Architectures Software Developer’s Manuals, <http://www.intel.com/products/processor/manuals>.

- [22] “AMD-phenom-processor-model-numbers-feature-comparison.”
<http://www.amd.com/us/products/desktop/processors/phenom/Pages/AMD-phenom-processor-model-numbers-feature-comparison.aspx>.
- [23] Y. Xie and G. H. Loh, “Dynamic classification of program memory behaviors in CMPs,” in *CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects in conjunction with the 35th International Symposium on Computer Architecture*, ISCA-35, pp. 1–9, 2008.
- [24] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 340–351, IEEE Computer Society, 2005.
- [25] N. Rafique, W. T. Lim, and M. Thottethodi, “Architectural support for operating system-driven cmp cache management,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT ’06, (New York, NY, USA), pp. 2–12, ACM, 2006.
- [26] G. E. Suh, S. Devadas, and L. Rudolph, “A new memory monitoring scheme for memory-aware scheduling and partitioning,” in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA ’02, (Washington, DC, USA), pp. 117–128, IEEE Computer Society, 2002.
- [27] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, “Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT ’06, (New York, NY, USA), pp. 13–22, ACM, 2006.
- [28] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 423–432, 2006.