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 fre-
quent 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 per-
formance isolation refers to variability in performance of an application due to
the behavior of other applications co-running with it. This performance depen-
dency 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 un-
der 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 sched-
uler 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
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] toolset (mentioned in chapter 5 at page 87). AKULA is a recently developed toolset 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.