THREAD MAPPING USING SYSTEM-LEVEL MODEL FOR SHARED MEMORY MULTICORES

by

Reshmi Mitra

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2015

Approved by:

_____
Dr. Bharat Joshi

_____
Dr. Ryan Adams

_____
Dr. Arindam Mukherjee

_____
Dr. Arun Ravindran

_____
Dr. Alexander Gordon

ABSTRACT

RESHMI MITRA. Thread mapping using system-level model for shared memory multicores (Under the direction of DR. BHARAT S. JOSHI)

Exploring thread-to-core mapping options for a parallel application on a multicore architecture is computationally very expensive. For the same algorithm, the mapping strategy (MS) with the best response time may change with data size and thread counts. The primary challenge is to design a fast, accurate and automatic framework for exploring these MSs for large data-intensive applications. This is to ensure that the users can explore the design space within reasonable machine hours, without thorough understanding on how the code interacts with the platform. Response time is related to the cycles per instructions retired (CPI), taking into account both active and sleep states of the pipeline. This work establishes a hybrid approach, based on Markov Chain Model (MCM) and Model Tree (MT) for system-level steady state CPI prediction. It is designed for shared memory multicore processors with coarse-grained multithreading. The thread status is represented by the MCM states. The program characteristics are modeled as the transition probabilities, representing the system moving between active and suspended thread states. The MT model extrapolates these probabilities for the actual application size (AS) from the smaller AS performance. This aspect of the framework, along with, the use of mathematical expressions for the actual AS performance information, results in a tremendous reduction in the CPI prediction time. The framework is validated using an electromagnetics application. The average performance prediction error for steady state CPI results with 12 different MSs is less than 1%. The total run time of model is of the order of minutes, whereas the actual application execution time is in terms of days.

ACKNOWLEDGEMENTS

I wish to thank and acknowledge the contributions of several people who guided, encouraged and contributed to this dissertation. First, I wish to thank my advisor, Dr. Bharat Joshi, for guiding me through the dissertation work with his innovative and valuable ideas. I appreciate his patience, kindness and research experience, which proved invaluable to me. This work is a product of the endless discussions with him. I want to acknowledge Dr. Ryan Adams for sharing his electromagnetics expertise. I will also thank Dr. Adams, Dr. Arun Ravindran and Dr. Arindam Mukherjee for consenting to be part of the dissertation committee. Their useful suggestions and comments especially during the earlier parts were extremely helpful for completing this research.

A special note of thanks goes to Intel and “Intel Education Network” for their generous donation of the Clovertown machines and VTune tool. A sincere gratitude is extended to Dr. Kaustav Das for lending his expertise on Machine Learning. I will also thank Mr. Ram Narayan, my mentor in AMD, for his guidance and words of encouragement. A special mention goes to Ms. Jerri Price and Mrs. Stephanie LaClair Brilliante and for their kindness and support.

A special acknowledgement goes to my sister Chayanika, my late grandfather Dr. Shyam Shashtri and my uncle Dr. Vijay Babbar for their love and support. Lastly, and most importantly, I wish to thank my parents, Mrs. Supriya Mitra and Mr. Samaresh Chandra Mitra, for their absolute confidence in me. They raised me, supported me, taught me, and loved me. To them I dedicate this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANN	Artificial Neural Networks
AS	Application Size
BPU	Branch Prediction Unit
CPI	Cycles per instruction
DTLB	Data translation lookaside buffer
FDTD	Finite Difference Time Domain
FFT	Fast Fourier Transform
HPC	High Performance Computing
I/O	Input/Output
MCM	Markov Chain Model
MOB	Memory ordering buffer
MS	Mapping strategy
MSW	MagnetoStatic Wave
MT	Model Tree
OS	Operating Systems
PEC	Per-event-contribution
ROB	Reorder buffer
RS	Reservation station
SML	Statistical machine learning

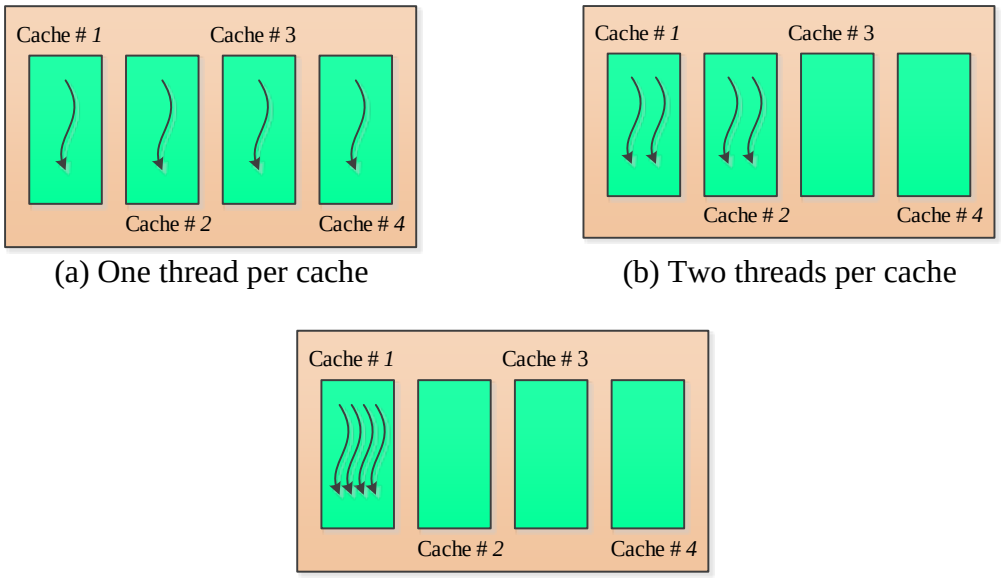
CHAPTER 1: INTRODUCTION

1.1 Problem Statement

1.1.1 Problem Description

A primary challenge for parallel programming in multicore machines is to optimize the thread count and blocking them to a particular core. Consider the example in Figure 1.1 for sample mapping strategies for a 4-thread 4-cache quad-core machine. This diagram illustrates the plausible options just with four threads. The key question here is which of these options are suitable for minimizing the application response time: (a) threads mapped together to facilitate inter-thread communication, or (b) threads mapped on independent cores to provide access to individual cache. In reality, the best possible thread-to-core binding choice is interdependent on multiple factors such as detailed knowledge about the architecture, application and their interaction. Hence, the selection process is not that straight-forward.

In fact, the study on the related work shows that exploring thread-to-core mapping strategies for any parallel application on multicore machines is very expensive both in terms of resources and man-power [11], [20], [38]. Recent research in the High Performance Computing (HPC) [41] has emphasized the need for efficient frameworks to improve the programmers' productivity. However, some significant disadvantages of the existing performance models are: (1) lack of efficient framework for top-level design exploration, (2) dependence on detailed system information, (3) low design reusability, and (4) large computation time to generate training data.



(a) One thread per cache

(b) Two threads per cache

(c) Four threads per cache

Figure 1.1: Sample thread-to-core mapping strategies for four threads on a quad-core machine

There are three main portability and scalability barriers for this optimization problem of selecting a suitable mapping strategy (MS). The first barrier is the large number of possible MS combinations to be explored, as multicore architectures proliferate. All the measurements and performance trends from other platforms are rendered useless with the move to a different architecture.

Secondly, the parallel programs have very domain-specific data and control structures, which regulate the architecture-application interactions. It is very difficult to infer accurate performance trends, without measurements from running the code on the platform. It also means that simply writing parallel programs based on data or task parallelism are not sufficient for extracting good performance. In short, the optimum MS may vary with data size and thread counts, because of the trade-offs between computation, communication, and memory accesses.

The third reason is that the Operating System (OS) controlled thread binding may seem inefficient without adding a priori information about the application behavior. Although the focus of this work is on user-controlled thread-to-core binding schemes, but, it is important to review the fundamentals of the OS controlled strategies.

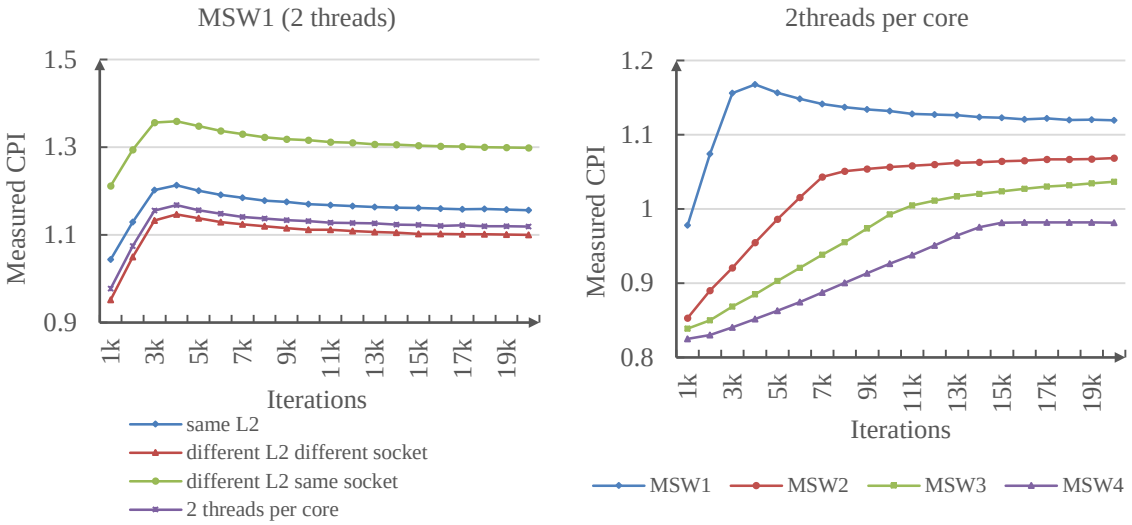
The native Linux scheduler is based on ranking user processes and threads according to their priority. It works on the principle of maximizing the throughput and fairness of service. The processes running for a long time have their priorities dynamically decreased [63]. The large parallel applications are categorized as batch processes, and often run in the background. Because of their low requirement for the user interaction and dynamic priority scheme of the OS, these programs are often penalized by the scheduler. Sometimes this renders the OS-controlled thread binding as extremely inefficient.

Summarizing the above arguments, it is extremely time-consuming for the application development community to predict the performance of a parallel program without understanding the details of the platform. The key idea here is to relieve them from a thorough understanding of this application-architecture interaction. This is to ensure that they can test out a large number of implementations in reasonable machine hours with fairly accurate results. Thus, the proposed solution is aimed towards reducing the application response time.

For the given parallel application, the response time (or, wall clock time) is proportional to the cycles per instruction (CPI). The number of instructions remains almost consistent all throughout the MSs. Thus, CPI can be a convenient metric for identifying the implementation(s) with lowest wall clock time, given that the number of

threads remains same. It is important to add that the measurements includes both pipeline states i.e. when the application is running and pipeline waiting during halt instruction.

1.1.2 CPI Characterization Curves



(a) Measured CPI with iteration variation for different mapping strategies for the same application size

(b) Measured CPI with iteration variation for different application sizes for the same mapping strategy

Figure 1.2: Measured CPI variation curves

Before formalizing the problem statement, it is imperative to examine some sample measured CPI versus iteration characteristic curves. In the first Figure 1.2(a), an electromagnetics application (MagnetoStatic Wave – abbreviated as MSW) is plotted for four different MSs with constant data size and thread count. For the same application, Figure 1.2(b) explores CPI variation with four different data sizes for the same MS. The smallest one is labelled as MSW1. The next higher one with twice the number of row and column elements is referred as MSW2 and so forth.

It is evident from the graphs that the measured CPI with increasing iteration follows an initial rise and then, eventually reaching saturation. This characteristic is a property of the underlying application behavior and can vary with changing MS and data size. For

this particular electromagnetics code, the actual value of steady state CPI may vary anywhere from 15-25% of the initial measurements. Also, there can be up to 60% increase in the actual application size (AS) steady state CPI with respect to smallest AS values. In fact, the characteristic figures consistently show poor scalability. And hence, there is no straightforward method to infer the values from the initial few (thousand) iterations or smaller AS measurements.

Summarizing the above ideas, the primary criteria for the proposed model design are:

a) Fast: Application performance information from “shorter” runs.

b) Accuracy: Reasonably precise with correct performance trends.

c) Portable: Performance prediction without detailed architecture features or code inspection.

1.1.3 Objective

The goal of this dissertation is to design a fast, accurate and portable performance model framework for exploring various thread-to-core MSs for parallel programs. The main metric of steady state CPI (related to response time) will allow the engineer to explore these MS possibilities within reasonable machine hours. This framework is directed towards large data-intensive, iterative parallel applications for shared memory multicores.

1.2 Proposed Model Framework

The proposed model framework consists of two main parts: application-based timing model and thread-to-core throughput prediction model.

The first part is studying the impact of the data partitioning and task reordering schemes on the performance without any empirical measurements. This is for developing

an execution timing cost function based on computation, communication and synchronization. The parallelizing strategy recommended here will be optimized in the next model.

The second part of the framework is for the thread-to-core steady state CPI prediction. It is a hybrid combination of the Markov Chain Model (MCM) and Model Tree (MT). The individual approaches, MCM and MT, in isolation have some significant disadvantages for performance prediction. Analytical models (such as MCM) are typically easy to modify and have low run-times. However, they may run the risk of larger error margins, as compared to statistical machine learning (SML) based techniques (such as MT). On the other hand, MT performs best with higher volume of training data, which is computationally very expensive to generate.

Hence, the driving idea here is to design a framework with reasonable accuracy, which can simultaneously capture the advantages of the both model types - fast, portable and scalable. Based on the comments presented so far, the metrics used in framework evaluation are listed below:

- Predicted steady state CPI for all MSs for the actual AS within permissible error limits.
- Minimum computation time for generating training set.
- Successful performance extraction without any real measurement data.

The basic idea in the proposed solution uses the small-scale performance for predicting the large-scale application behavior. This is implemented by capturing the thread stall patterns from hardware event counter data of the small AS using MCM. It translates the detailed, cycle-accurate information into a design space which has a much

reduced parameter count. The MT, in turn, takes in this input and extrapolates the behavior for the actual AS. The final step in the framework comprises of converting these thread stall information back to CPI values.

The architecture under study is a hierarchical, shared memory, multiple issue multicore processor, with coarse-grained multithreading. The model is validated for Intel Xeon Clovertown (X5365) using measurements from VTune (Intel's cycle accurate performance analyzer) for the electromagnetics application (MSW).

1.3 Research Contribution

The primary contributions of this dissertation are as follows:

a) Identifying a computationally efficient approach for parallel program performance prediction. This solution should be relevant to a large class of data-intensive iterative parallel applications running on shared memory multicore machines.

b) Laying down the details of a fast, accurate and portable framework based on the above approach. The thread-to-core model predicts the steady state CPI and points towards a suitable MS. It is in the form of hybrid combination of MCM and MT. It is based on extrapolating the effect of thread stall patterns for the actual AS. The inputs come from the cycle-accurate measurements of smaller data size of the same program.

c) Exploring data partitioning strategies using application-specific timing model for the electromagnetics-based algorithm. This consists of representing the execution time (including computation, memory access, inter-processor communication and synchronization time) without any empirical measurements.

d) Model validation on Intel Xeon Clovertown (X5365) using the same electromagnetics application (with data size and core scaling) with respect to

VTune measurements. As demonstrated in the results section, the model effectively predicts CPI with an average error of 0.168% with standard deviation of 3.866%. The total run time for model is of the order of minutes, whereas the actual application response time is in terms of days.

e) Using the thread-to-core model to explore parallel application pairs that can be executed together on the same machine, without mutual interference of overall performance.

In summary, the designed framework is to ensure that the parallel application development community can make informed MS choices without thorough understanding on the application-architecture interactions.

1.4 Dissertation Outline

There are six main chapters in the dissertation. Related work about modeling is presented in Chapter 2. Chapter 3 provides detailed information on the microarchitecture of the dual quad core Intel Clovertown machine. The entire design of the proposed model framework is explained in Chapter 4. The model results and discussions are presented in chapter 5. Conclusions are presented in Chapter 6 with the summary of the main contributions, and the future work. The appendix covers additional material on the MCM solver and the measurement methodology.

CHAPTER 2: RELATED WORK

This chapter covers topics related to performance prediction of parallel programs running on shared memory multicores. In general, analytical models are known for their fast predictions as shown in the survey paper [40]. Typically, there is a significant trade-off between prediction accuracy and model design time. With the emphasis on better error margins, models end up capturing extensive information on architecture and application features. Thus, the analysis is tightly connected with detailed knowledge of the platform and program interaction as presented in [26]. This labor-intensive process is detrimental for model reusability for exploring large number of application flavors. Hence, it may become an expensive method for top-level design exploration.

Table 2.1: Comparison criteria for different approaches

Main Criteria	Analytical Model	Statistical Machine Learning (SML)
Accuracy	Medium - high	Very high
Cost of training data	Low	High
Model reusability	High	Low
Understanding architecture-application interaction	High	Low

Usually, the performance trends are much closely connected with the key parameter measurements such as cache misses, page walks, division operations and other stall-

causing long latency events. Hence, models based on empirical measurements are relatively more accurate than the purely analytical models as shown in [11] and [31]. However, generating such large volume of measurement data is computationally expensive for large applications. The key ideas are summarized in Table 2.1.

The three main categories of the most popular class of performance analysis models covered in this chapter are - Amdahl's Law ([1] to [11]), Queuing Theory ([12] to [22]), and SML ([31] to [37]). For each technique examined here, the survey typically begins with revisiting the multiprocessor models for a better perspective. However, the current architectural features and profilers have paved the way for the new frameworks. Hence, after drawing sufficient parallels with the older work, the study moves on to the modern multicore performance analysis models.

A review of the related work is presented below:

2.1 Amdahl's Law

Amdahl's Law [1] is one of the most popular models to predict performance of parallel applications. In essence, it is used for determining program speed-up, where its value is limited by the parallelizable fraction of the application. However, the model and some of its extensions [5] do not address the problem of performance loss due to resource contention.

This section begins with the studies indicated in [2], [3], [4], which relate the speedup only to the inherent parallelism of the program and express the performance loss due to constraining the number of cores. These directed acyclic graphs based simple models, while useful, are impractical for understanding current multicore systems, where resource

contention can greatly hamper the net performance. In addition, the error margins (up to 33%) do not match up to the current acceptable limits.

The first significant multicore extension to the Amdahl's Law is proposed in [5]. This work was one of the first to show the importance of entire chip's performance over pipeline efficiencies. Compared to the above model, a more optimistic result on the multicore scalability is presented in [6] with the most significant suggested deterrent being the memory wall. Critical sections in the parallelizable fractions for the original multicore model [5] are introduced in [7]. The multiprocessor performance model is linked to the CPU and cache area in [8] without addressing the interference between resource contentions. A comprehensive model of [5], [7], [8], [9] is presented in [10], by including sequential-to-parallel synchronization and inter-core communication.

Although the above models in [5], [6], [7], [8], [10] are suitable for exploring a range of multicore topologies, it may prove insufficient for studying a specific application and especially, thread-to-core MS. One of the most important reasons is the lack of a clear-cut approach to obtain the model inputs while exploring different flavors of the application. This makes these models unsuitable to capture the application behavior and its impact on the underlying architecture. In addition, these works ([5], [6], [7], [8]) fail to address the problem due to multiple threads competing for the same resources.

One of the first (Amdahl's Law based) models to study parallel program performance for shared memory multicores is presented in [11]. This model explores inherent program parallelism using Amdahl's law with resource contention. The speed-up expression is similar to the one developed in [2]. In the model proposed in [11], the input parameters are dependent upon two major measurements. The first is the OS run-queue on a single

baseline run. The second one is the M/M/1 queuing model with inputs from the hardware counters using two to three runs. However, the main reason for performance loss is restricted to the memory behavior. This means that it may not be sufficient to account for compute-intensive application performance (e.g. floating-point operations).

2.2 Queuing Theory

This section covers techniques based on analytical models using probability matrices for performance prediction. Each component of the platform (mainly, processor and memory) is assigned a particular type of queue based on their length and waiting times. The idea is to determine the throughput of the entire system by solving the queue at steady state.

This section begins with superscalar uni-processor performance modeling in [12], continues with multithreaded multiprocessors [13], [14], [15], and concludes with multicores [18], [22]. One of the first versions of the uni-processor queuing model [12] represents machine parallelism and program parallelism as linked MCM. This preliminary model is restricted to very few key architectural parameters and does not include the effect of cache misses or bus transactions.

In comparison to the above model, a more integrated approach is presented in [13] and [14]. In these works, closed queues are used to denote the processor and the memory. Their model is extended further in [15], where the effects of cache and main memory are explored separately. Their model also accounts for prefetching and synchronization delays. However, these models treat the effect of stalling-causing events as independent queues. This may become an incomplete representation, since it does not account for their inter-dependent effects.

There is another set of queuing models which extensively cover just the pipeline parallelism for multithreaded systems. They represent both the multiprocessors ([16], [17], [18]) and multicores ([19]). Multicores thread-level queuing models in [20], [21] study the impact of thread-to-core mapping choices with respect to packet processing in communication processors. Their target is the phase before actual programming and the metrics under consideration is throughput, delay and loss. However, their analysis excludes impact of instruction or microarchitecture details on the application behavior.

The focus of all these models ([13], [14], [15], [16], [17], [18], [19], [20]) is restricted to understanding the architecture rather than optimizing a given application. Their goal is to provide a top-level analysis for the early stages of the processor design. This means that these models do not define a clear-cut methodology for performance estimation for a particular implementation of a given application.

One of the earliest works on multicores for parallel program performance is presented in [22]. The developed model is for fine-grained multithreaded single-issue multicore processor. The main thread stalling event is restricted to memory-misses and floating point instructions. However, this model's accuracy is dependent on the measurements based on their own micro-benchmarks. Thus, their approach has portability issues with respect to other pipelines (e.g. Intel Core in the dissertation). We have extended their work on coarse-grained multithreaded, multi-issue processor. Secondly, we have consolidated the measurement techniques using cycle-accurate values. Also, by the use of training data we have managed to bring down the error margins from 8% to less than 1%.

2.3 Other Analytical Models

This section covers models on performance prediction of program parallelism belonging to miscellaneous category. It includes topics on parameterized models ([23], [26], [27]), application traces ([24], [25]) and task graph analysis ([29], [30]).

The application-specific analytical model [23] is parametric, with input information in the form of basic machine performance numbers (latency, MFLOPS rate, and bandwidth) and application characteristics (problem size, decomposition method, etc.). Although this approach produces extremely accurate results, but these highly customized, labor intensive models require a thorough understanding of the application and its implementation.

Simple benchmark probes create machine profiles and a separate tool generates application signatures in [24] and [25]. Their approach is to gather a trace for each point in the parameter space for generating the application signatures. The convolution method is used to map them onto the machine profile. Depending on trace sampling rates, their predictions achieve error rates between 4.6% and 8.4%. Full traces obviously perform best, but such trace generation can slow application execution by almost three orders of magnitude.

Parallel application prediction using a semi-automatic parameterized model is designed in [26]. The application behavior is based on properties of the architecture, program binary, and other application inputs. However, they cannot account for some important architectural parameters, such as cache associativity in their memory reuse modeling. The approach in [27] relies on low-level hardware detail, which makes it a non-trivial process to develop the model for other architecture variants.

The model proposed in [28] predicts highly iterative parallel program execution time without code inspection or detailed simulations. The performance on a target system is derived using the combination of two different components. The first one is the known program performance on a reference system. The second is the relative performance between the two systems from a very short run of the application. Hence, it is dependent upon using partial results for different problem sizes and degrees of parallelization, which renders this model less accurate.

Tools based on task graph analysis ([29], [30]) help programmers identify optimization opportunities that are useful for application scalability. Compiler-synthesized static task graph generator is presented in [29]. It determines the sequential computations (tasks), parallel structure of the program (task precedencies, explicit computations) and the control flow that determines the parallel structure.

In comparison, a fairly accurate, but somewhat restricted model is proposed in [30]. This is a hybrid approach of combining analytical modeling with a performance simulation language called PAMELA. It automatically generates a formal description of parallel implementation to enable code and mapping decisions. However, these models ([29], [30]) based on task graph analysis are intentionally excluded from this review. This is because of their requirement of in-depth knowledge of the application. Rather than identifying the parallelization possibilities, our attempt is to explore the different implementations of the same parallel program with minimum possible information.

2.4 Statistical Machine Learning

SML is a relatively new approach for exploring the parameter space for performance prediction. These models feed empirical performance data to methods such as Artificial

Neural Networks (ANN), Linear Regression or other Machine Learning models for performance prediction. Typically, these methods have very high accuracy and are independent of any code analysis or architecture simulation. However, they are heavily dependent upon a very large training data set. This means that the input has a relatively high computational cost and low reusability. With a large number of performance influencing parameters, another significant drawback for these methods is the model design time and the proper choice of the training set.

The model in [31] uses ANN is used for predicting performance of High Performance Computing (HPC) application SMG 2000 (OpenMP based) with an average error of less than 10%. However, they do not consider statistical techniques for preliminary data analysis or regression methods. In comparison to [31], the profiling cost for building the model is reduced in [32] by pre-processing. They compare the effectiveness of piecewise polynomial regression and ANNs for predicting performance in the context of varying input parameters. Their findings suggest that prediction accuracies between the two approaches are comparable, but each approach is advantageous in different contexts. However, they report that the training process is significantly simplified through the use of ANNs.

Online learning based model in [33] is used to study the parallel program scaling behavior (processor count). Using linear regression techniques, they predict large scale performance behavior based on the extrapolation of small scale performance results. However, their focus is on the architecture exploration without any comments on the thread-to-core mapping. A machine learning approach (ANN and Support Vector Machine) using offline profiling is established in [34]. Their focus is on predicting the

number of OpenMP threads and the scheduling policy on a homogeneous architecture. However, their methodology is closely linked to the platform and is tested on an application with a single loop. ANN and regression models are used in [35], [36] and [37] to dynamically control number of threads for achieving higher predicted power/performance efficiency.

2.5 Summary

A review on the parallel application performance prediction on shared memory machines is presented in this chapter. Topics on Amdahl's Law, Queuing Theory, application-specific models and task graph analysis are covered as part of the analytical models. This chapter begins with a comparison of analytical modeling and statistical measurement based methods. Two main arguments are presented here. The first one is on the appropriateness of these methods for application optimization. The second point is the cost associated with generating the training data for the methods based on empirical measurements. This section concludes with the various SML algorithms used for analyzing parallel applications.

CHAPTER 3: MACHINE MICROARCHITECTURE

The framework is validated for a dual quad core machine with four processor cores per socket. It is a dual quad core machine with two processor cores on a single die. The two cores share a second level cache which improves performance by reducing the necessity to save the same data in multiple locations. A top level diagram is presented in Figure 3.1. This section is devoted to a brief description about the microarchitecture details including pipeline functionality and the cache architecture.

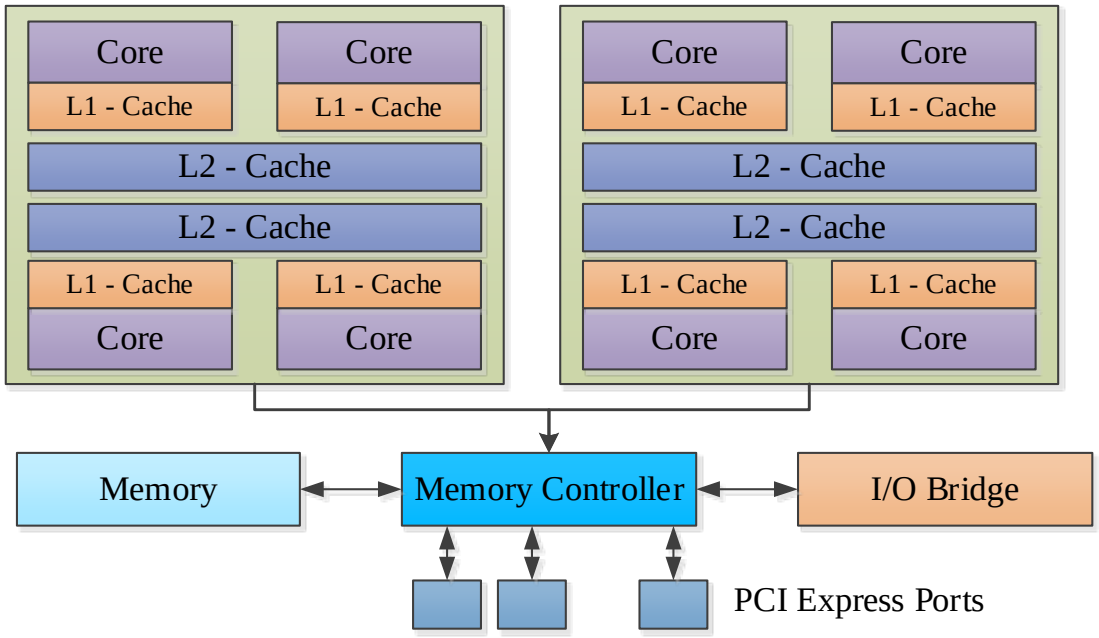


Figure 3.1: Block diagram representation of dual quad core “Clovertown”

The pipeline of the Intel Core micro-architecture consists of three major components [34]:

a) An in-order issue front end which fetches instruction streams from memory, with four instruction decoders to supply decoded instruction (micro-ops) to the out-of-order execution core.

b) An out-of-order superscalar execution core that can issue up to six micro-ops per cycle and reorder micro-ops to execute as soon as sources are ready and execution resources are available.

c) An in-order retirement unit that ensures the results of execution of micro-ops is processed and architectural states are updated according to the original program order.

The front end of the pipeline consists of the Branch Prediction Unit (BPU), Instruction Fetch Unit and, the Instruction Queue and Decode Unit. BPU enable speculative execution by predicting various branch types: conditional, indirect, direct, call and return using dedicated hardware for each type. In order to maintain a constant bandwidth irrespective of irregularities in instruction stream, Instruction Fetch Unit serves three primary purposes of prefetching, predecoding and buffering instructions, and caching frequently-used instructions. Instruction Queue and Decode Unit decodes up to four instructions, with the flexibility to fuse multiple micro-ops derived from the same macro-op thereby reducing the number of micro-ops that need to be executed.

The execution core is the collective name for renamer, reorder buffer (ROB), and reservation station (RS). In this process, the out of order core performs the following steps:

a) Allocates resources to micro-ops by moving micro-ops from the front end to the ROB and RS.

b) Binds the micro-op to an appropriate issue port in the RS.

c) Renames sources and destinations of micro-ops, enabling out of order execution.

d) Provides data to the micro-op when the data is either an immediate value or a register value that has already been calculated.

Table 3.1: Hierarchical shared memory multicore specification

Processor name	Intel Xeon X5365 (Clovertown)
Microarchitecture	Intel Core
Number of cores	4
Clock speed	3 GHz
Min. feature size	65 nm
Instruction set	x86
FSB	1333 MHz
Pipeline	14 stages, In-order issue, Out-of-order Superscalar execution, In-order retirement
L2 cache	8 MB, 64B/line, 16-way, Write back, 14 cycle latency, shared
L1 I-cache	32 KB, 8-way
L1 D-cache	32 KB, 64B/line, 8-way, Write back, 3 cycle latency, private
ALU	1 cycle execution
Branch predictor	32 byte, 16-entry Return Stack Buffer, private
TDP	120 W
Performance	38 GFlops in Linpack benchmark

As shown in the Figure 3.2, the memory architecture contains an instruction cache and a first level data cache in each core. The two cores share a 4 MByte level-2 cache. All caches are writeback and non-inclusive. The data translation lookaside buffer (DTLB) works with two levels of hierarchy, with each level supporting 4 KByte pages or more. The entries of the inner level (DTLB0) are used for loads. The entries in the outer level (DTLB1) support store operations and loads that missed DTLB0. The third most significant component of the memory in each core is the memory ordering buffer (MOB). It supports speculative issue of loads and stores, and ensures retired loads and stores have correct data upon retirement. The memory performance is improved by data prefetching in L1 cache and L2 cache, store forwarding, memory disambiguation among others.

As shown subsequently in the model evaluation section(s), most of the architectural features mentioned here are used as pointers for measurement purposes for the throughput model.

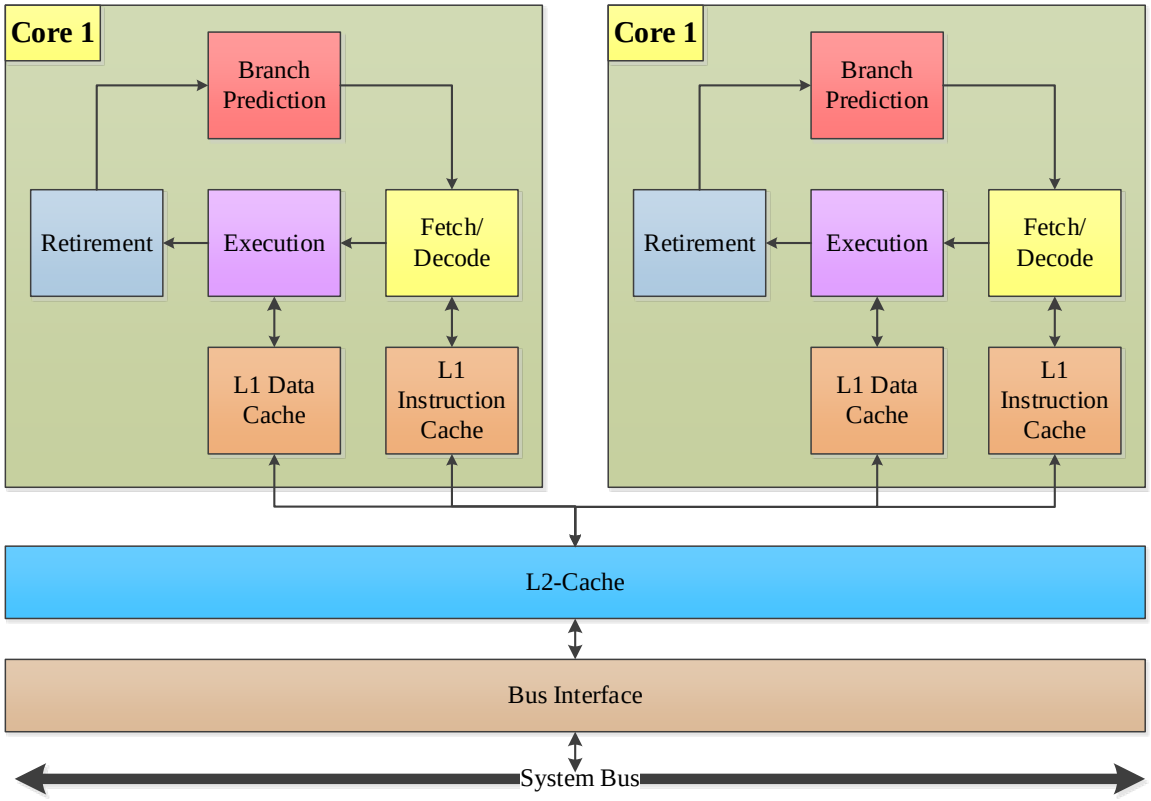


Figure 3.2: Memory architecture representing shared Level-2 cache

CHAPTER 4: THREAD-TO-CORE PERFORMANCE PREDICTION FRAMEWORK

This chapter provides a detailed discussion on the proposed model framework and the measurement methodology. It is divided into six main parts. In the first section, an introduction on the framework is presented. The throughput predicting MCM is the most important component of this framework. Hence, the next section specifies the step-by-step MCM design procedure. The pre-processing methodology for the MCM is illustrated in Section 4.3. The second step of the framework is an extrapolator based on the MT learning technique. This is presented in the fourth section. This includes the basic input data characteristics, background on MT and step-by-step algorithm description. The individual elements (MCM and MT) for the model framework are combined together in the fifth part of this chapter. The chapter is summarized in the last section.

4.1 Framework Overview

This dissertation develops and validates a fast, accurate and portable methodology for estimating steady state CPI and exploring thread-to-core MS. It is directed towards large data-intensive parallel applications running on multicores. This section introduces the top-level ideas of the MS exploration framework.

There are five main parts in Section 4.1. It begins with the key ideas on design of the proposed solution. The first two sections cover the main part of this design process (a) identifying major components of the problem, and (b) selecting a suitable approach for each of these parts. The main steps of the solution framework are highlighted in Section 4.1.3. The fourth section provides a list of characteristics that qualifies the parallel

application for this framework. The final section is a chapter outline for the benefit of the reader.

4.1.1 Identifying Problem Components

This section begins with emphasizing the effect of cycle-accurate empirical measurements. The next step is exploring ways to reduce the computation time for generating this input (performance) information. The discussion then moves on to the criticism of each approach and identifying the most relevant solution. Based on this selection, the problem is then broken down into smaller components.

The importance of empirical measurements from hardware performance counters to track application behavior is established in the beginning of Chapter 2. This data provides crucial insights on how the program interacts with the platform and its performance bottlenecks. However, it is expensive to generate these measurements for the actual AS for a long duration, especially, till the point the application performance reaches steady state. There are two main solutions to address this large response time issue and reduce the number of computations. These options involve collecting performance data from either one of the following cases:

a) Shorter runs for actual AS, or

b) Extended runs for smaller AS

The first option is covered in [28] within the context of cross-platform performance of repetitive scientific applications. However, the main disadvantage of these partial program runs is that there is no direct way to determine the smallest number of iterations that are adequate to represent the steady state behavior. It is more of a hit-and-trial method, with the performance analyst regularly interrupting the application to compare

the trends with the previous iterations. For example, in Figure 1.2(b) the actual AS (MSW4) data set shows steady state behavior at 15,000 iterations, but the smaller AS data (MSW1) shows steady state at 9,000 iterations. Hence, the resultant method involves significant user intervention and runs the risk of low automation.

On the contrary, some applications (e.g. Vortex and Wave of SPEC95 benchmark suite) may demonstrate regular periodic behavior all throughout the execution. In such cases, a purely statistical approach is sufficient to extrapolate the behavior from the initial iterations. This approach is widely covered in [64] for the basic block distribution analysis. However, the present framework addresses applications with repetitive operations, which may not demonstrate such periodic behavior.

The second option of using performance data for extended runs with smaller AS is examined further in the next few paragraphs. This approach is commonly used in preliminary exploration phase of application development, and there is no available record of related work citing this course of action for exploring parallel applications. However, as pointed out in the subsequent arguments, the advantages of this approach have catapulted its use in the present framework.

Depending on the data requirement, this response time is typically much less than the one for actual AS. Thus, it allows the performance analyst to examine the behavior till steady state for multiple sizes of data, without incurring the complete machine hour cost of the actual AS. After identifying the key thread stall patterns, it is much easier to transfer them to the actual AS.

However, the main drawback with this approach is identifying a training data that is rich enough to capture all the performance stalls in the actual AS. In spite of this issue

being a difficult problem, it can be addressed by a two-step solution. The first one is developing a detailed list of criteria for the selection of training data set. The second step is forcing the generated models to undergo a vigorous validation process, before applying it on the unknown data.

These guidelines for selecting training data, essentially, ensure similar execution patterns with respect to the real data size. This list (in the first step) includes exactly same concurrency, tasks per iterations, memory accesses, and boundary term sharing patterns as the actual AS. Thus, by maintaining the application execution intact in the training set during the data collection process, this approach requires low user-intervention. This is unlike the previous option with low scope for automation. Hence, the current thread-to-core model framework is designed based on the second approach of using extended runs for smaller AS.

Purely SML methods are known to be robust with low error margins in their predictions. However, they are dependent upon large training data for their results. Their second significant disadvantage is due to the poor scale-ups of the performance events. Without detailed information about the application behavior, it is difficult to anticipate how to draw the focus on a particular class of events (e.g. computations or memory accesses). This makes the data exploration a time-intensive ad hoc process. Hence, these methods are considered unsuitable for direct extrapolation from small AS to the actual data size.

In summary, it is useful to break down the problem into two parts. The first part of the framework handles the measurements from small AS and extracts the main thread stall patterns. This step should accept all the detailed cycle-accurate measurements. It should,

then, transform it into a very small number of parameters reflective of just the basic thread activity, irrespective of the cause for performance loss. The second part can, then, extrapolate this parameter variation from the small AS to the actual AS data.

4.1.2 Hybrid Approach for Proposed Solution

Continuing the design methodology of the proposed framework, the previous section establishes two key ideas in this process. The first significant comment is on breaking down the problem into (two) different components. Secondly, it explores the disadvantages of using a purely SML based approach, especially for the first part. To counter the disadvantages mentioned earlier, MCM is selected for this first step in the framework. It essentially translates the cycle-accurate empirical measurements into system-level performance terms with respect to thread stalls. The thread is treated as the primary element in the current MCM to represent the parallel behavior.

Summarizing the main ideas discussed henceforth in this section. It begins with the key features of MCM. It then proceeds with the basics of MCM design for building the framework. The next paragraphs explore the hybrid nature of the proposed solution and especially highlighting the exchange of MCM and SML based technique. This section concludes with the design strategies that have helped in reducing the response time.

There are three main benefits of using the MCM for the first part of the framework. Firstly, it goes beyond the black-box model treatment of the SML based techniques. Also, it is known to be comparatively easier to modify and thus, provides better design reusability. Thirdly, the performance analyst need not undergo data exploration during pre-processing stages to identify the key stall events (e.g. compute- or memory-

intensive). This step was a major drawback with SML, because the performance trends may vary with the application and reduces the extent of automation.

It is important to note here that MCM is used for predicting the state of system, while undergoing transitions within pre-defined states. However, it is unappealing in the context of the second part of the problem, which is extrapolating behavior with the change in data sizes. Under such conditions, it is very hard to perceive the system states in advance. Thus, as explained later, a SML based approach is used to overcome this unknown state issue. But considering its disadvantages pointed out in the previous section, its (prediction) scope is limited to the MCM characteristic curve.

Exploring the framework design further, the system-level information, such as, the number of cores and the memory hierarchy, are represented by the MCM states. Multithreading is shown as the state transitions. Hence, the model is generic enough to be easily extended to similar architectures. The value of transition probability terms is determined from the program characteristics. They are representative of the common long latency events such as cache misses, division operation, among others.

The designed MCM reduces the stall patterns to thread performance terms, namely, probability of thread to get suspended and probability of thread maintaining that suspended state. Application performance loss can occur in either of the two cases: (a) thread suspending frequently and recovering quickly, or (b) thread suspending rarely, but recovering after a long latency. Hence, the MCM gives rise to a characteristic curve, which maps a particular probability combination with the given system performance.

SML based technique is used for the second part of the proposed solution. Its role is to extrapolate these thread probability values for the actual AS. Thus, the SML based

technique operates on the much smaller MCM characteristic curve. This is a limited data space, in comparison to exploring the combination of large number of individual events to track performance loss.

Since only MCM has the key for the probability values to CPI transformation, it is again involved as the third step of the framework. Thus, the SML output probability value of actual AS is fed back to the MCM to determine the unknown CPI. This comprises a brief overview on the hybrid nature of the proposed solution.

Summarizing the above discussion, the improvement in the prediction time is achieved, because of two main reasons. First, the program characteristics are adequately modeled from the performance information from considerably smaller AS, in comparison with the actual data size. This significantly brings down the required number of computations.

Secondly, the architecture-application interaction for the actual AS is represented by the mathematical expressions (in MCM) and hence, avoids the use of cycle-accurate information. Because of these two primary advantages, the use of this framework can drastically reduce the design time for application optimization, by testing multiple parallel implementations in a short duration.

4.1.3 Thread-to-Core Model Outline

The main parts of the proposed solution are elaborated in this section. It begins with formulating the three key steps of the framework. It, then, introduces the MCM design within the context of constructing the model. It concludes with the comments on using SML based learning technique called MT as the extrapolator.

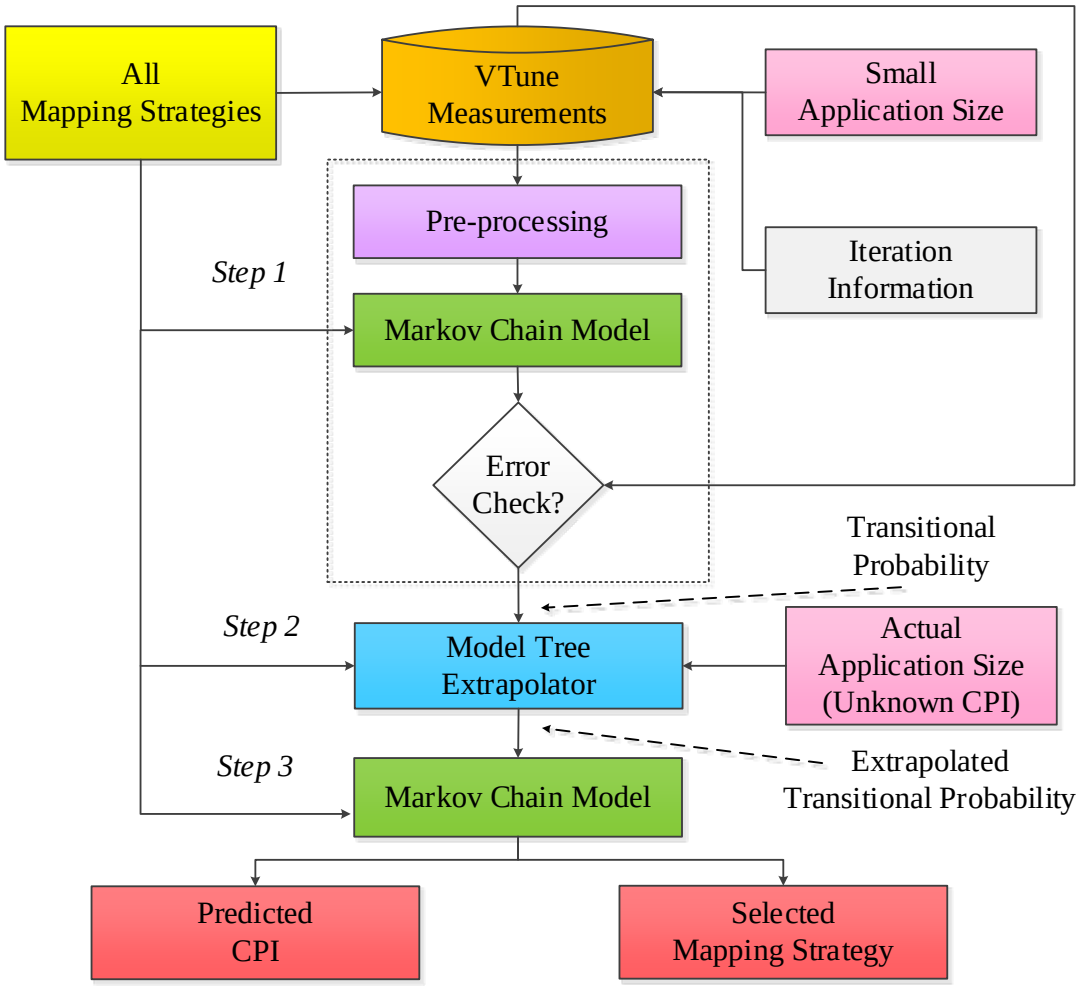


Figure 4.1: Thread-to-core performance prediction framework

A top-level overview of the entire model framework is shown in Figure 4.1. Based on the discussion in the previous section, this consists of the three primary parts as listed below:

a) Translating machine and application information into MCM states and the thread transitional probability values.

b) Extrapolating above thread transition probabilities for actual AS using MT.

c) Throughput prediction for actual AS with these extrapolated transition probabilities using MCM.

The most significant component in the model framework is the MCM. Hence, before diving into a much detailed process of constructing the model, it is important to understand an overview of the MCM. Typically, it has a fixed number of pre-defined states to depict the nature of the system. It transitions from one state to another in the state space with certain probability. In the current context, the primary objective of the MCM is to calculate the probability of (system) states under steady state conditions. The predicted CPI is inferred from that state probability information.

The first step in the framework begins with generating small AS- and MS-specific VTune measurements for the pre-processing steps. This includes separating instruction and cycle-related measurements, optimizations according to per event contribution values, among others. It, then, compares the measured and predicted CPI (from MCM) to select the best possible probability term that accurately represents the application behavior. In principle, this step transforms the cycle-accurate measurements into cycle-independent performance data, as represented by the thread transitional probability terms.

The MCM is again used in the third step of the framework (Figure 4.1). This MCM is the same model as before, with exactly the same states and the probability expressions. However, in this step, it receives input probability values from the MT extrapolator. These values denote the change in the system conditions, due to the larger data size application, running on the same machine. The output in this final step is the predicted steady state CPI for the actual AS.

In the second step in Figure 4.1, the AS- and MS-specific transition probability terms are, then, fed to the MT for extrapolation for the higher AS. Some other MCM-related information about thread binding and states is also passed to this model. The

extrapolation occurs on two levels: AS and iteration count. Model Trees has been used for this block of the framework. This part is actually a classifier and it constructs tree-based piecewise linear models as leaf nodes for the input information. It is readily available as part of M5´ module [52] in Weka [51].

A much simplified approach with the focus on just the individual performance events is not a practical idea, because of their non-linear scale-up characteristics, as discussed earlier in Section 4.1.1.1. Hence, a part of the design effort has been the development of a robust estimator for these probability terms.

4.1.4 Properties of Application

After a brief on the model design, it is important to understand the properties of application, which qualify them for the proposed solution. This framework is directed towards large parallel applications with all the following characteristics:

a) Homogeneous and heterogeneous multithreading workloads with aperiodic performance.
b) Rank order of MS remains consistent over the entire application execution.
c) Large number of same data operations occurring over long durations. These repetitive activities may include floating point computations, logical, string matching, among others. Additionally, the number of instructions should be high enough to justify the requirement for performance prediction over very large time periods.
d) Large memory requirement. This is with respect to per-iteration memory and may involve global sweep over data size much larger than the available cache. Also, the applications may have dependencies over space and time with low data reuse.

e) Constant number of threads throughout the application execution.

4.1.5 Framework Outline

The subsequent next two sections are assigned to the MCM design. This includes a step-by-step procedure for the hierarchical system-level model as shown in Section 4.2. The MCM for step 1 involves an elaborate sequence of pre-processing steps and is covered under Section 4.3. The MT input characteristics and learning technique basics, including building and pruning of the tree, are described in Section 4.4. The individual model elements of the entire framework are combined together in Section 4.5. This section provides in-depth discussion on the three primary steps shown in Figure 4.1.

4.2 Markov Chain Model

4.2.1 Introduction

The first two paragraphs review the fundamentals of MCM. It is a memory-less process and hence based on independent trials. The next state depends only on the current state, and not on the sequence of events that preceded it. The process moves successively within a group of fixed number of predefined states. For example, if the current state is s_i and the next state is s_j, then the system transitions from one state to another with a probability of p_{ij}. Alternately, the process can remain in the current state, and this occurs with a probability p_{ii}. The predefined states represent the state space. The state diagram is constructed with these states and the transitional probabilities.

An initial probability distribution specifies the conditions of the system in the beginning. On solving the transition matrix for the MCM, one can find the steady state probability of all the possible states. This numerically represents the feasibility of the system ending up from the first state to all other states, once it attains stable behavior at

steady state. MCM have been widely used in statistical modeling of various computing systems including telecommunication networks and multiprocessors.

After a brief review about the theory of MCM, it is important to discuss its design for the proposed framework. The main challenge in the current context is on how to define the states and the transition probability matrix which comprises the state space. Ideally, they should adequately represent the machine behavior and program characteristics, without focusing on a deeper understanding of the program or the platform.

Continuing on this reasoning with respect to the architecture, it is a well-known fact that the modern multicores derive their vastly improved performance from multiple cores, hierarchical shared memory and on-chip caches. Hence, the design of MCM reflects the memory hierarchy of the machine. The MCM states are based on the total number of threads supported by the particular cache level. It begins with the basic L1-cache level MCM. In this state machine, a thread can transition between active and suspension. Thus, the basic model is constructed using these two states.

This L1-cache level model is extended further to the L2-cache level model. The number of all possible ways a transition can occur is constrained by the number of software threads per L2-cache and, the current state of active threads. Multiple L2-level models constitute the system-level model. The L2-caches communicate with each other only through the main memory. Hence, the probability expressions for the system-level model are designed reflecting this concept. This represents the machine described in Chapter 2. However, the model hierarchy can be easily extended to more layers of cache.

The thread stalling activities affects the overall CPI and represents the program characteristics. They are included in the model as the thread transitional probability

values. They are calculated from the long latency events (frequency and average latency) from the empirical measurements. This, in turn, is calculated by the number of such instructions, frequency of their occurrence and, average cost per occurrence. For solving the transition matrix of the MCM, the initial state is assumed as all active threads. This is a realistic situation, since all threads are fully capable of processing the instructions in the beginning.

The model can be solved to predict the probability of system transitioning to all threads suspended (p_s) at steady state. The final throughput of the machine, representing the CPI is given by $1/(1 - p_s)$. A detailed step-wise description on finding steady-state probabilities and CPI using MCM is shown in Appendix A.

The main assumptions in the current MCM are:

a) Coarse-grain thread-level parallelism or TLP (architecture-related): This implies that thread switching occurs only during the stalls. Also, multiple threads mapped to a single core may not translate to any performance gain. This is a constraint for the MCM state transitions.

b) Single active-to-suspend transition per time stamp (MCM-related): The definition of MCM dictates that only one transition can occur in a single time stamp. This implies that the time interval between two events is so small that it can be safely assumed that only a single active thread can transition to the suspended state in a given time stamp. However, due to difference in the number of cycles of the long latency events, there can be multiple suspended threads returning to the active state in the same time stamp. This is again a constraint for the state transitions.

c) Constant number of threads throughout application execution (application-related): The model dictates that the number of thread remains constant for the application from start till the end of program execution. Varying thread count will translate to absorbing MCM states, which is beyond the scope of the current work.

A brief outline of this section on MCM is described henceforth in this paragraph. A basic thread-level model framework is designed in Section 4.2.2. This is extended further with the plausible state transitions for L2-level model in sections 4.2.3 and 4.2.4. A system-level representation is developed in Section 4.2.5. The next two sub-sections shows derivation on the expressions for the transition probabilities (probability of active-to-suspend - p and suspend-remaining-suspended - q). The last section 4.2.6 covers the alternative MCM model states.

4.2.2 Basic Model Hierarchy

Based on the discussions in Section 4.2.1 on the basics of MCM design, the multi-levelled thread transition diagram for the entire system is shown in Figure 4.2. The three main levels (L1-cache, L2-cache and main memory) represent the memory hierarchy. In this figure, the distinction in each level is based on the total number of threads (active or suspended) supported by that particular abstraction layer. The state transitions in each layer are derived from this information. The transition probability expressions represent the combinations of all possible number of active and suspended threads associated with that particular level. In the subsequent paragraphs, the transition diagram for each of the levels is explained in greater detail.

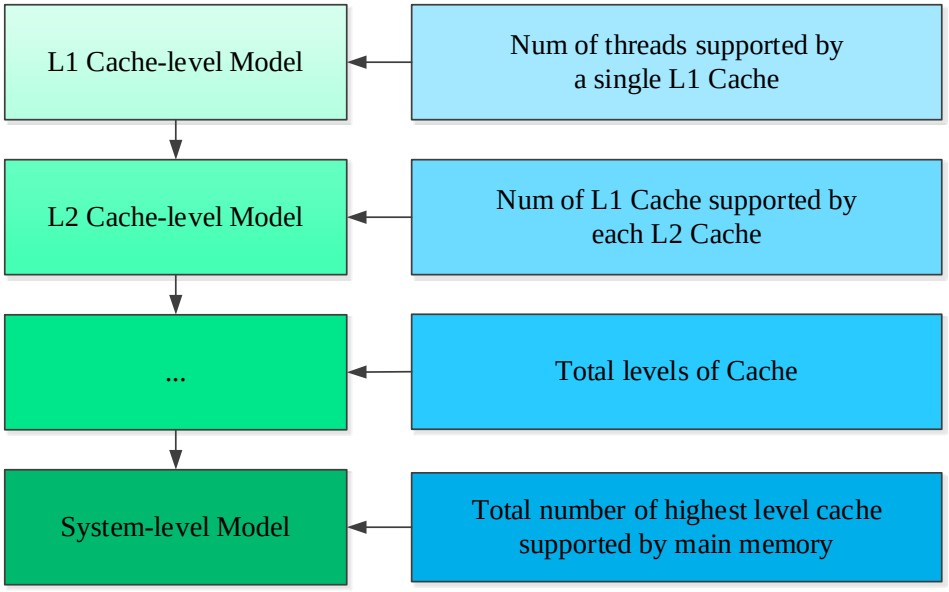


Figure 4.2: Markov chain model hierarchy

The first abstraction layer begins with the L1-cache level state machine as shown in Figure 4.3. As mentioned in Section 3, the best possible performance is derived with a single thread per core and each core has its own private L1-cache. In this state diagram, the system alternates between the active or suspended (no activity) states. Hence, the L1 cache-level thread transition diagram can be appropriately assigned these two basic states. Since there are exactly two (thread) states, which are completely independent of each other, this can be adequately represented as a Bernoulli trial.

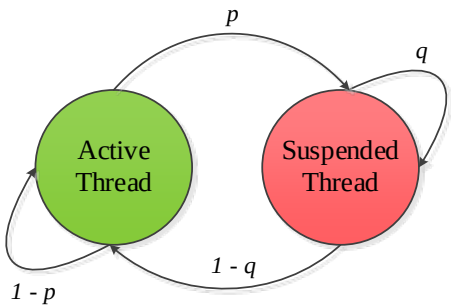


Figure 4.3: L1-cache level thread transition diagram showing thread transitional probability pair

The main thread transitional probability pair (Figure 4.3) are given as:

a) p - Probability of active thread getting suspended

b) q - Probability of suspended thread remaining suspended

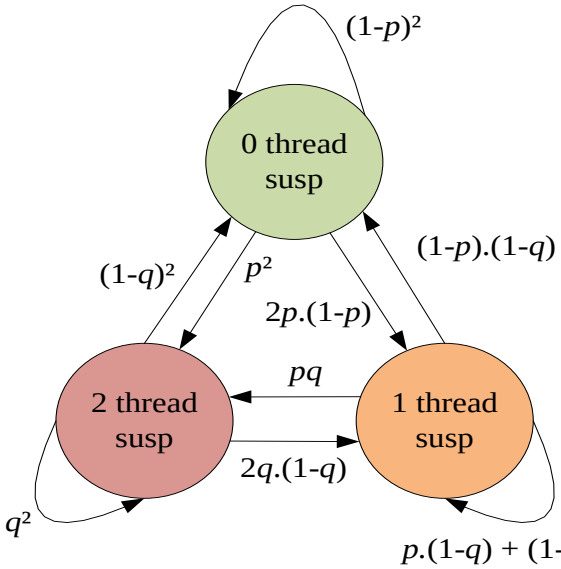


Figure 4.4: L2-cache-level thread transition diagram

A single L2-level model is the next tier of the entire system state space as represented in Figure 4.4. For the current machine, each L2-cache is shared by the two cores (Figure 3.1). Hence, the three possible states for each L2-cache, depending on all threads active, only one thread active and both threads active, are given as 0, 1 and 2, respectively. The transition probability expressions shown in the above figure are developed in the Section 4.2.4.

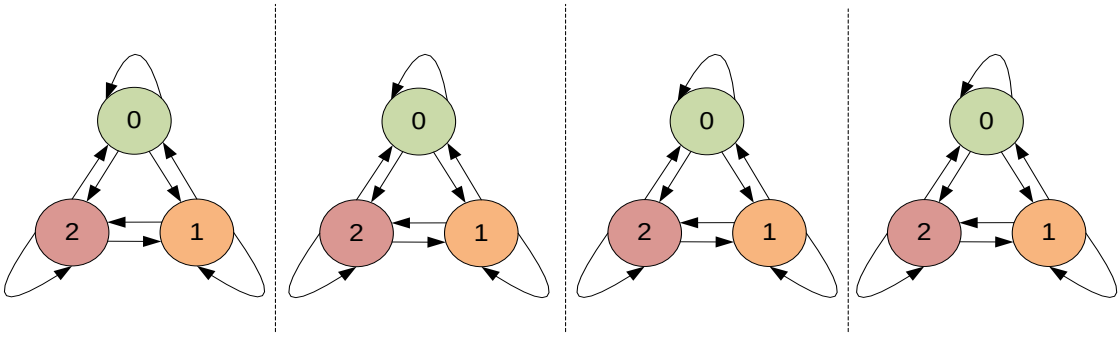


Figure 4.5: System-level thread transition diagram

The final system-level model with all the eight cores is shown in Figure 4.5. This is built from the four L2-cache models (Figure 4.4). Thus, the total number of states possible in the final model is 81. This is calculated from the number of possible states for each L2-cache to the power of number of such L2-cache present in a single machine = 3^4.

An alternate treatment to the total number of states is considering each thread state individually. For example, for the current case with eight threads, each can be either active or suspended. Hence, the total number of states becomes a slightly higher figure of 256 (number of possible states for each thread to the power of number of such threads), increasing the execution time for solving the model. And hence this approach has been dropped in favor of gaining prediction speed.

4.2.3 Examples for L2-level Markov Chain Model

In the previous section, the L1-cache-level MCM states are established. In this section, each and every possible transition for a hypothetical L2-cache is explored. This exercise is performed to develop the transition probability expressions in the next section. Assume an L2-cache with a set of maximum two active threads, similar to the current Clovertown machine. As an example, the transition of one active thread (and one suspended thread) to any one active thread is considered. This transition can occur by *any* of the two following cases:

a) Active thread remains active and suspended thread remains suspended, or

b) Active thread becomes suspended and suspended thread becomes active.

Since each of the above instances is independent of each other and *equally likely* to occur, the transition probability value is the *sum* of probabilities for both the possible cases.

Based on the decrease, increase, or no change in the number of "active" threads, these events are grouped into three categories as upstream, downstream, and constant, transitions respectively. The symbol (*x*, *y*) represents *x* active threads and *y* suspended threads. In general, a single L2-cache supporting *N* threads with *i* suspended threads is represented as (*N* - *i*, *i*). Consider a hypothetical single L2-cache supporting 8 threads each. Thus, a downstream transition from 2 active threads to 4 active threads, is represented as (2, 6) to (4, 4). This is illustrated in Figure 4.6.

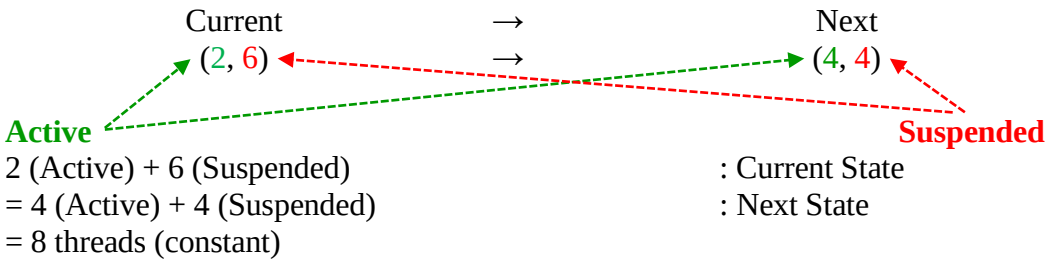


Figure 4.6: L2-level active and suspended thread notations

All possible transitions for this case are explained in detail in this section and summarized in Table 4.1. All possible next states are listed out in the columns marked as “Next State (i) to (iii)”. The suspended threads can either be active threads getting suspended (row 2), or, from suspended threads remaining suspended (row 3). The number of such possibilities depends on the total number of active (and suspended threads) in the next state. For better explanation, the set of active and suspended threads are treated separately to count all such occurrences. On reversing the treatment, i.e., on beginning with the set of suspended threads (instead of set of active threads), exactly same cases will emerge.

Table 4.1: Downstream transition example from (4, 4) to (2, 6)

Thread States	Current State	Next state *(i)*	Next state *(ii)*	Next state *(iii)*
Active threads	4	(2, 2)	(1, 3)	(0, 4)
Suspended threads	4	(0, 4)	(1, 3)	(2, 2)
Total	(4, 4)	(2, 6)	(2, 6)	(2, 6)

In the above example, following transitions for set of active thread *cannot* be considered, because they violate the maximum number of *active* threads in the next state:

a) (4, 0) to (4, 0)

b) (4, 0) to (3, 1)

An example of constant transition, of 6 suspended threads to 6 suspended threads is shown as follows:

Table 4.2: Constant transition example from (2, 6) to (2, 6)

Thread States	Initial State	Next state *(i)*	Next state *(ii)*	Next state *(iii)*
Active threads	2	(2, 0)	(1, 1)	(0, 2)
Suspended threads	6	(0, 6)	(1, 5)	(2, 4)
Total	(2, 6)	(2, 6)	(2, 6)	(2, 6)

In an upstream transition example of 4 suspended threads to 6 suspended threads, is illustrated as:

Table 4.3: Upstream transition example from (4, 4) to (2, 6)

Thread States	Initial State	Next state *(i)*	Next state *(ii)*	Next state *(iii)*
Active threads	2	(2, 0)	(1, 1)	(0, 2)
Suspended threads	6	(2, 4)	(3, 3)	(4, 2)
Total	(2, 6)	(4, 4)	(4, 4)	(4, 4)

4.2.4 Transition Probability Expressions for L2-level Markov Chain Model

The transition probability expression for a single L2-cache is denoted as *"transition (i, j, x)"* as shown in Figure 4.7. It represents the transition probability element from state-*i* to state-*j* for x^{th} L2-cache. In this section, the expressions for *"transition (i, j, x)"* for Constant, Upstream and Downstream are developed.

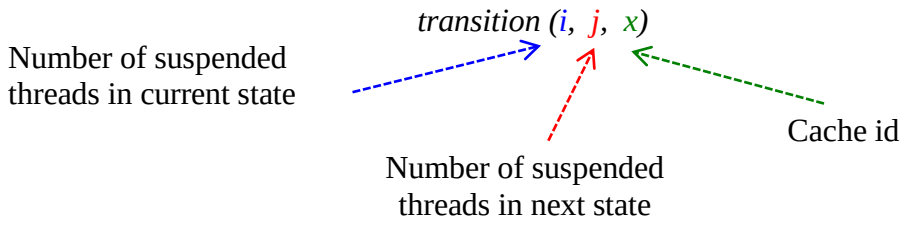


Figure 4.7: Representation for 1-tuple for a single L2-cache model

As explained earlier, the basic L1-cache level state model (Figure 4.3) is adequately represented as Bernoulli trials with binary values. Thus, the transition probability values for the different states are assigned based on the binomial distribution. The expressions for constant, upstream and downstream transitions are as follows:

Constant $i = j$

(N, 0) to *(N, 0)* with $i = 0$ (1a)

$$p_{ij} = (1-p)^N$$

(0, N) to *(0, N)* with $i = N$ (1b)

$$p_{ij} = q^N$$

(N-i, i) to *(N-i, i)* with $i \neq 0, N$ (1c)

$$p_{ij} = \sum_{m=0,}^{m=i} \sum_{n=i-m}^{n=i-m+1} \left[\binom{i}{m} q^m.(1-q)^{i-m}\right] \times \left[\binom{N-i}{n} p^n (1-p)^{N-i-n}\right]$$

Upstream $j > i$

(N, 0) to *(N-j, j)* with $i = 0$ (2a)

$$p_{ij} = \binom{N}{j} p^j (1-p)^{N-j}$$

(N-i, i) to *(0, N)* with $j = N$ (2b)

$$p_{ij} = q^i p^{N-i}$$

(N-i, i) to *(N-j, j)* with $i \neq 0, j \neq N$ (2c)

$$p_{ij} = \sum_{m=0,}^{m=i} \sum_{n=j-m}^{n=j-m+1} \left[\binom{i}{m} q^m . (1-q)^{i-m}\right] \times \left[\binom{N-i}{n} p^n (1-p)^{N-i-n}\right]$$

Downstream $i < j$

(N-i, i) to *(N, 0)* with $j = 0$ (3a)

$$p_{ij} = (1-q)^i (1-p)^{N-i}$$

(0, N) to *(N, 0)* with $i = N$ (3b)

$$p_{ij} = (1-q)^N$$

(N-i, i) to *(N-j, j)* with $i \neq N, j \neq 0$ (3c)

$$p_{ij} = \sum_{m=0,}^{m=j} \sum_{n=j-m}^{n=j-m+1} \left[\binom{i}{m} q^m . (1-q)^{i-m}\right] \times \left[\binom{N-i}{n} p^n (1-p)^{N-i-n}\right]$$

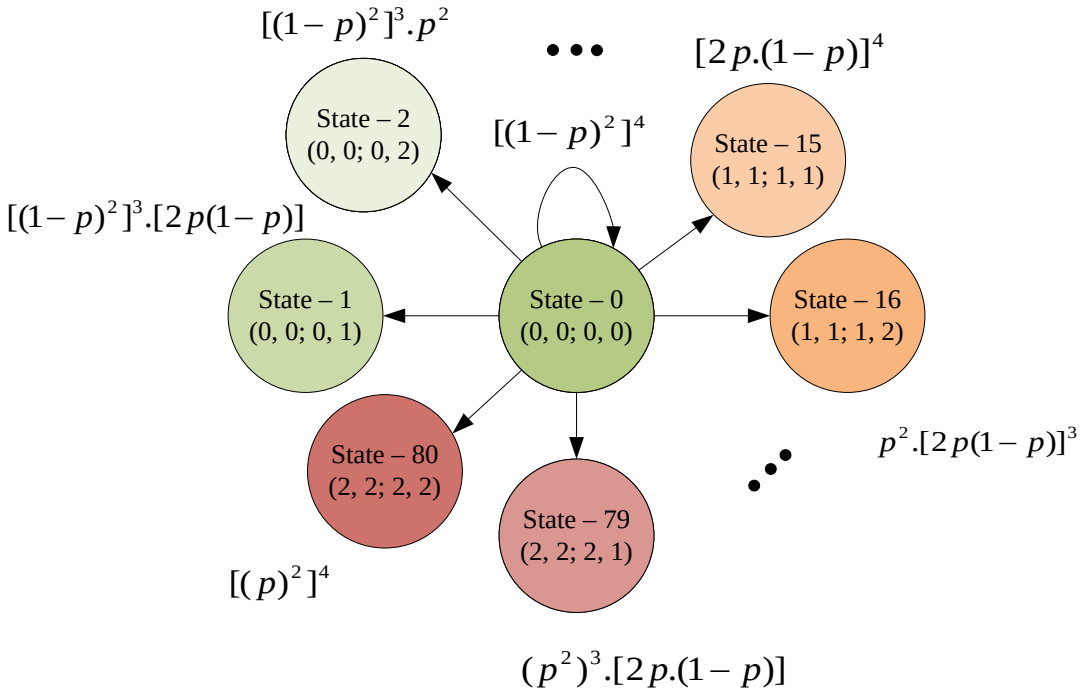
The expression (1c) is examined in detail here. The other expressions can be derived with a similar logic. In the set of *i* suspended threads associated to a single L2-cache, any *m* of them will continue to remain suspended, each with the probability of *q*. This means that the remaining $i - m$ suspended threads will become active (each with the probability of $1 - q$). This can happen in any $\binom{i}{m}$ number of ways. Thus, the transition probability value for the set of suspended threads is given by $\left[\binom{i}{m} q^m . (1-q)^{i-m}\right]$.

At the same time, in the set of the $N - i$ active threads, any n of them can get suspended, each with a probability of p. This also means that the remaining $N - i - n$ active threads will continue to remain active, each with the probability of $1 - p$. Again, this can happen in any $\binom{N-i}{n}$ of ways. Hence, this can be represented as $\left[\binom{N-i}{n} p^n (1-p)^{N-i-n}\right]$. Since, both events are occurring at the same time, hence product rule is applied. The total number of possible outcomes is given by the outer $\sum$ term. It is constrained by the total number of threads related to that particular cache.

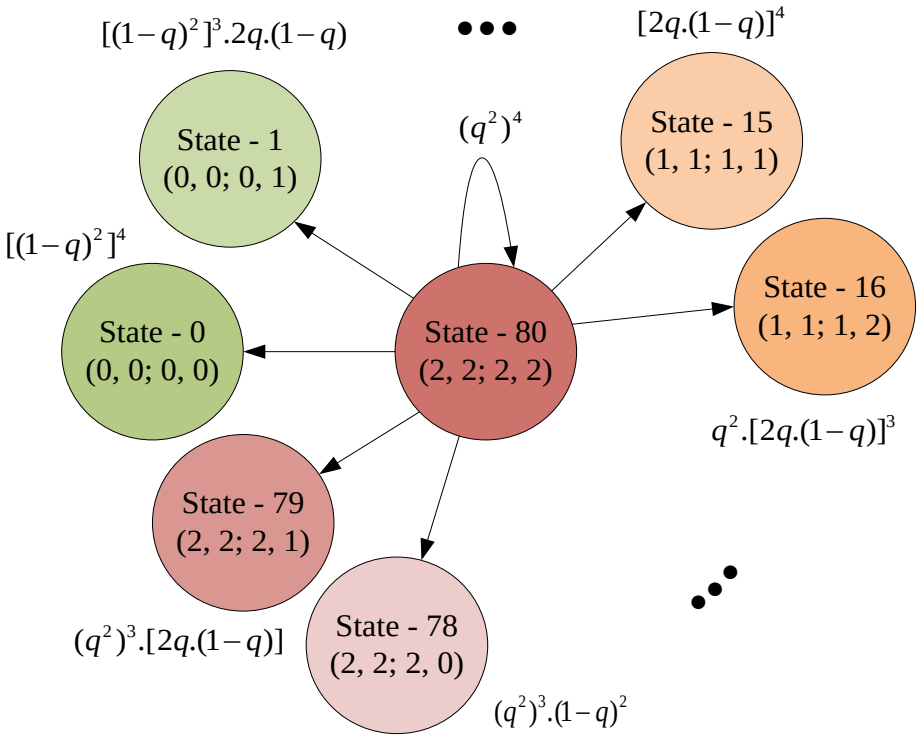
4.2.5 System-level Model

The final step is combining all the L2-cache-level probability expressions (Figure 4.4) in developing the system-level model (Figure 4.5). The states are represented as (i_1, i_2; i_3, i_4) a 4-tuple, for each of the L2-cache. Mathematically, each tuple is given by *transition* (i, j, x) as shown in Figure 4.7). As mentioned earlier, the total number of states possible is 81. These are assigned all possible combinations from all threads active (0, 0; 0, 0) to all threads suspended (2, 2; 2, 2). This also implies that the state 1 (0, 0; 0, 1) is different from state 3 (0, 0; 1, 0). Although, only one thread is suspended in both the cases, but the suspended thread belongs to a different L2-cache.

As mentioned earlier in Section 6.1, the performance prediction model is designed for a hierarchical shared memory machine. This means that communications between any two L2-caches takes place through the main memory and, hence, function independently.



(a) current state having all active threads



(b) current state having all suspended threads

Figure 4.8: System-level thread transition diagram

Mathematically, the system transition is represented as:

$$p_{ij} = \prod_{s=1}^{m} \text{transition } (i_s, j_s, s) \quad (4)$$

In current scenario, the value of *m* is four representing the maximum number of L2-cache in the system.

Using equation (4), two examples are shown in Figure 4.8.1. In the first Figure 4.8.2 the current state is all active threads i.e. (0, 0; 0, 0). Consider the next MCM state with all active threads. For each L2-cache (Figure 4.4), the probability of transition of all active threads remaining active is $(1 - p)^2$. Hence, for all system threads to continue to remain active, the transition probability is given by $[(1 - p)^2]^4$ using equation (4). The other transitions are worked out with a similar reasoning.

The m_i term for each event is calculated separately using equation (5). All the performance events used for calculating the above probability terms (both *p* and *q*) are enlisted in Section 5.7. These two transition probability values (*p* and *q*), along with cache and threading information is fed into the MCM solver. It assigns the number of states and their transition values from the input data to calculate the predicted CPI.

4.2.6 Expression for Probability of Active-to-Suspend

The method used here to find state transition probability of each thread, is first suggested in [6]. The current dissertation has been modified for Intel Clovertown versus, the previous, Sun Niagara processor. Hence, there are significant changes in the model design and performance measurement procedure.

The probability of an active thread to get suspended (Figure 4.3) is calculated from the ratio of the instructions causing a thread to get suspended, to the total number of retired instructions. It is shown by the following expression:

$$p = \sum_{i=1}^{N} \frac{I_i}{N_{total}} \tag{5}$$

where,

N = Total number of threads

I_i = Number of instructions causing the thread *i* to be suspended

N_{total} = Total number of instructions executed on the particular core

The numerator term is represented by the additional instructions for the multi-threaded versus single-threaded implementations. The detailed formula is shown below:

$$p = \sum_{i}^{r} \left(E_{i,m} - E_{i,s}\right) \Big/ Number\ of\ instructions\ retired \tag{6}$$

where,

r = Total number of events causing an active thread to stall

$E_{i,m}$ = Total number of occurrences for event-*i* for the multi-threaded implementation,

$E_{i,s}$ = Total number of occurrences for event-*i* for the single-threaded implementation

4.2.7 Expression for Probability of Suspend-remaining-Suspended

Calculation of probability of suspended thread remaining suspended (Figure 4.3) is designed as a two-step process. Step one is to find *q* for a single event type. Step two is to compute *q* for the multiple events with a varying frequency of occurrence. Beginning with the case of single event latency, it is assumed that the number of stall cycles for a particular stall causing event is *M*. Hence, at the onset of a stall event, for the next *M* – 1 cycles, the thread remains suspended and gets reactivated only in the last cycle. Thus, the

probability of reactivating a suspended thread on any given cycle is *1/M*. This follows that the probability of a suspended thread to remain suspended is:

$$q = 1 - \frac{1}{M} \tag{7}$$

Realistically, a multithreaded implementation can encounter stalls due to multiple events each influencing the overall performance in varying degrees. Some of the events are memory reads/writes, resource throttling in the pipeline, (conditional and unconditional) jumps etc. The average stall cycle not only depends on the latency of each event, but also on the number of their individual occurrences. For example, if event-1 with 10 stall cycles occurs 5 times, and event-2 with 2 stall cycles occur 25 times, the average latency term can be computed as a weighted mean:

$$\bar{M} = 10 \times \frac{10\times5}{10\times5+2\times25} + 2 \times \frac{2\times25}{10\times5+2\times25} = 6 \text{ cycles}$$

Mathematically, expressing this concept, the average stall cycles for all of the events is modeled using the formula:

$$\overline{M} = \sum_{i=1}^{r} w_i m_i \tag{8}$$

where,

r = number of events causing a suspended thread to remain stalled

w_i = weight of each event-i

m_i = latency of each event-i

The m_i term for each event is calculated separately using equation (20) as shown in Appendix B. The specific VTune events used for calculating the thread transitional probability pair is briefly explained in Appendix C.

4.2.8 Alternative MCM L1-level Model States

In this section, two L1-level model strategies are investigated. Both these design decisions were made to gain model execution performance.

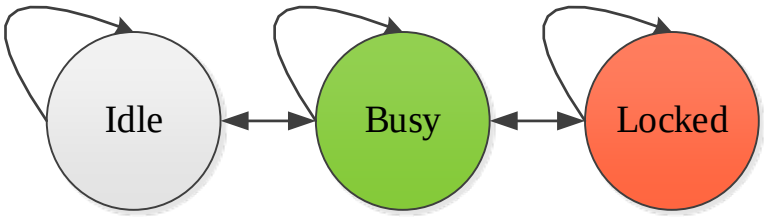


Figure 4.9: Alternate thread-level transition diagram

It begins with the tri-states for the basic L1-level model MCM. One way to improve model prediction accuracy is by modifying the thread states:

a) Idle or sleeping

b) Busy: resources available and thread doing useful work

c) Locked or suspended: resources unavailable and thread is “busy waiting”

The three states are shown in Figure 4.9. An actual locked thread cannot directly transition to an idle state, without performing assigned work with the newly available data or hardware. Similarly, the reverse situation is equally true about idle to busy transitions. Collectively, all such transitions will most likely improve the model prediction accuracy. Hence, the current approach of the binary states (Figure 4.3) of active and suspended (or no activity) might seem to be an over-simplification. However, this translates to lower state count at the system-level.

The increase in the number of states in the system-level model is explained in the following paragraphs. The three states for the current L2-level model (Figure 4.4) with the basic binary states for the L1-level model (Figure 4.3) can be listed as below:

a) all threads active

b) all threads suspended

c) only one active thread

Similar to the current L2-level model (Figure 4.4) a new L2-level model can be easily designed with the new L1-level model shown in Figure 4.9. The total number of states (after combining) for the new model can be listed as follows:

a) All threads are idle

b) All threads are busy

c) All threads are active

d) Any one thread is idle and another is busy

e) Any one thread is idle and another is locked

f) Any one thread is busy and another is locked

In comparison to the older model, the new state count increases by two times with the present treatment of three states in the basic MCM. Using equation (4), the total number of states in the new system-level model will increased by 16 times (6^4 versus 3^4). Hence, the present selection of binary states significantly improves the performance of the entire MCM solver.

The second alternative L1-level MCM is described henceforth in this paragraph. For more accurate representation, the lowest abstraction level should begin with the instruction-level, rather than the L1 cache-level model in Figure 4.2. A model capturing each instruction behavior should have better prediction results. However, there are two main disadvantages in such representation. First, there are up to six micro-ops that can be issued per clock cycle. Thus, the number of states in the system-level model will become

extremely high and, hence unmanageable. Second, tracking detailed instruction-level performance requires the measurement methodology to be much more expensive.

In both the above cases, there is a significant trade-off between speed versus accuracy. In other words, at the price of slightly better prediction accuracy, there will be a much greater loss in terms of the run-times for the MCM solver.

4.3 Pre-processing

The methodology for transition probability calculation using the performance measurements is presented in this section. This includes details on the various parts of the first framework step (Figure 4.1). This section is divided into four main segments. Motivation on using the multiple starting points during the pre-processing the cycle-accurate data is shown in the first section (4.3.1). The next section covers the details of (pre-processing) methodology including the computation of the starting vector. The calculations are explained using examples shown in Section 4.3.3. The last section presents an overview on the actual CPI used for error checking block in the first step in the framework.

4.3.1 Multiple Starting Points

In order to understand the motivation for the Multiple Starting Points, it is important to explore the MCM output properties. A sample predicted CPI characteristic curve for probability variation is shown in Figure 4.10. This 3-D graph is obtained by plotting the predicted CPI (from MCM) with respect to change in thread transitional probability pair value from 0 to 1, for different thread counts. Three primary conclusions can be derived from the visual inspection of this plot. This first significant observation is that the shape

of the plot is similar for all the different thread counts. The second comment is that the predicted CPI demonstrates performance loss with reduction in thread count.

The final remark is the performance loss (increasing CPI) with increasing thread suspension probability. This probability term refers to *both* the thread probabilities associated with active-to-suspend - *p* and suspend-remaining-suspended - *q*. This, in turn, means that the application can have low performance in any of the three cases - (a) very high *p*, average *q*, (b) average *p*, very high *q*, or, (c) average *p*, average *q*. In fact, there is a band of acceptable solution values (Figure 4.10), each corresponding to a different transition probability pair.

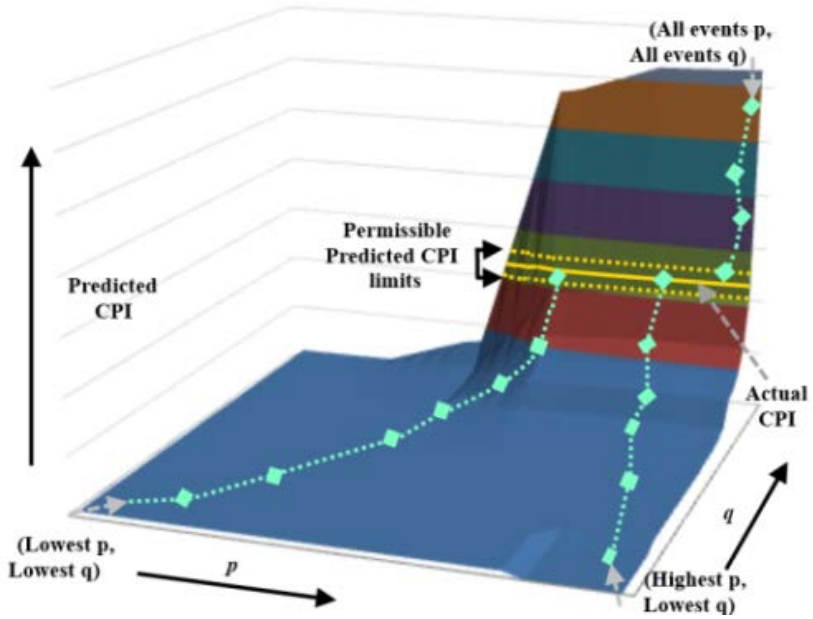


Figure 4.10: Multiple starting points

The idea about Multiple Starting Points is for optimizing the throughput prediction process. It is roughly based on multiple source optimization problems. The approach is to explore the “sources” one-by-one to identify the best possible result that most accurately represents the system behavior. In the current context as explained in the previous paragraph, it takes advantage of the multiple solutions for a single correct predicted CPI.

The pre-processing block searches for a solution pair with the shortest distance from the acceptable solution band. The shortest distance corresponds to the lowest number of steps to reach the correct transition probability pair. The primary idea here is to search for a single solution with acceptable error limits. This also implies that it is *not necessary* to identify all possible solutions.

The Multiple Starting Points are chosen as the combination of the boundary cases for per-event-contribution (PEC) for both *p* and *q*. A sub-set is shown in Figure 4.10 by the arrowheads. The three possible boundary cases are: the probabilities associated with - the highest PEC, the lowest PEC and all the events. This is referred as the *PEC Vector*. The Multiple Starting Points are the nine combinations of these three cases for *p* and *q* and is termed as the *Starting Vector*. Its components are listed as follows:

a) Lowest *p* PEC, lowest *q* PEC

b) Lowest *p* PEC, highest *q PEC*

c) Lowest *p* PEC, all *q* events

d) Highest *p* PEC, lowest *q* PEC

e) Highest *p* PEC, highest *q PEC*

f) Highest *p* PEC, all *q* events

g) All *p* events, lowest *q* PEC

h) All *p* events, *highest q PEC*

i) All *p* events, all *q* events

Using the MCM solver, predicted CPI is calculated for each of the starting points. The one with the minimum error has the shortest distance from the band of correct solution (Figure 10). Hence, this pair of *p* and *q* value is used for next step calculations.

4.3.2 Pre-processing Methodology

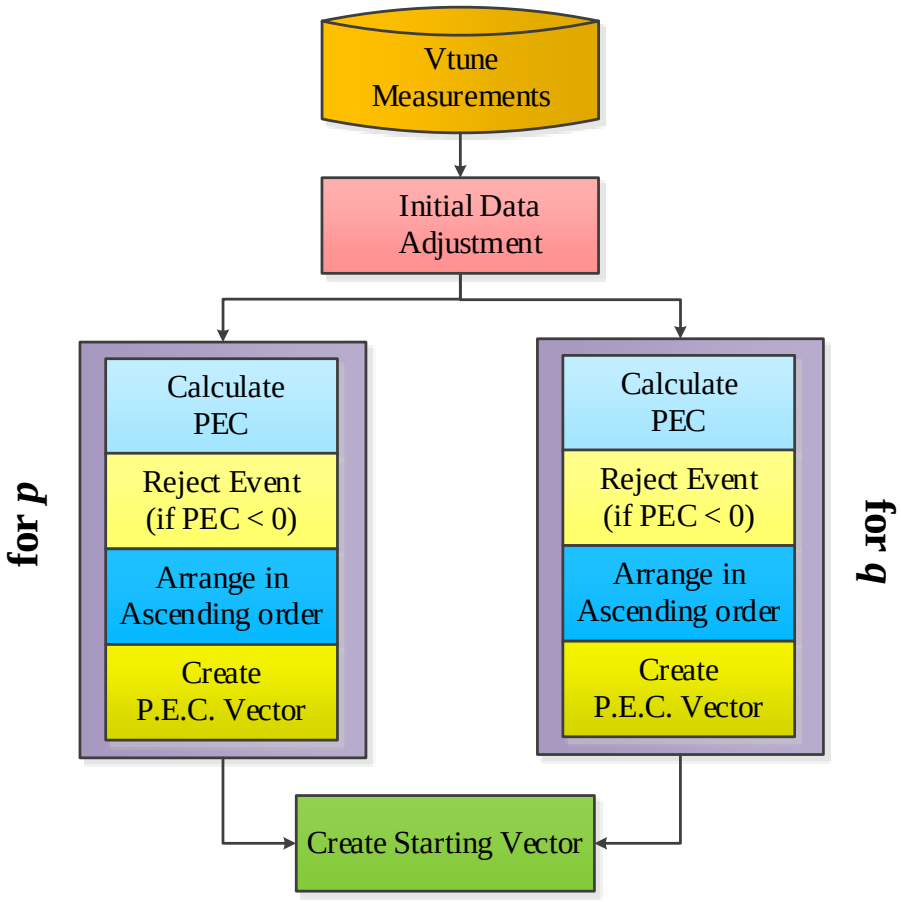


Figure 4.11: Pre-processing methodology

The main aim of the pre-processing block is computing the nine-element starting vector from the raw measured data. A detailed step-by-step algorithm is shown in Figure 4.11. The input to this block is the VTune application activity. It is in form of comma-separated values (CSV) files. The performance data is obtained from the average of three runs. This is, then, adjusted with the hardware performance monitoring unit interrupting frequency values of the associated events. More details about VTune and this step is covered in Appendix B. This is for calculating the actual number of occurrences. The basic formulae for the transition probabilities are discussed in the previous section. The

per-event-contribution (PEC) values (for both p and q) are calculated from the adjusted VTune measurements.

After arranging PEC values in ascending order, they are used to create the PEC vector for both p and q. A combination of the p and q PEC Vector is used for the Starting Vector. Out of the set of nine-element vector, the best starting point is selected from the predicted CPI (from the MCM solver) with the minimum error. This special starting point can, then, be used for a more detailed event search process to identify the (thread transitional probability) solution pair.

4.3.3 Starting Vector Example

The Starting Vector concept is explained using numerical examples in the following tables. First, PEC vector (lowest PEC, highest PEC, all events) for p is shown. Assume that the total instructions retired as 50:

Table 4.4: Per-event-contribution example for p

Performance Events	VTune multi-threaded value (measured)	VTune single-threaded value (measured)	PEC for p	Comments
A	20	10	$\frac{20-10}{50} = 0.2$	Lowest PEC
B	15	17	$\frac{15-17}{50} = -0.04$	Reject (PEC < 0)
C	30	15	$\frac{30-15}{50} = 0.3$	Highest PEC
p_{all}	$\frac{(20-10)+(30-15)}{50} = 0.5$			

Thus, the PEC vector for p is (0.2, 0.3, 0.5) from (lowest PEC, highest PEC, all events). In some cases, PEC for p can have a negative value. This means that a particular

performance event has a lower value as a multi-threaded implementation, in comparison to the single-threaded version. An example of this type of event can be reduction in misses, due to more cache available to each thread in parallel program. Hence, they are not responsible for thread stall and can be safely ignored. Next, the following example is a sample PEC vector for q.

Table 4.5: Per-event-contribution example for q

Performance Events	Total occurrences	Avg. latency per occurrence (cycles)	Total cycle time (cycles)	PEC for q	Comments
X	10	3	30	$1 - {}^{1}/_{3} = 0.33$	
Y	2	5	10	$1 - {}^{1}/_{5} = 0.8$	Lowest PEC
Z	15	10	150	$1 - {}^{1}/_{10} = 0.1$	Highest PEC
W	25	2	50	$1 - {}^{1}/_{2} = 0.5$	
Total			240		

Average latency is computed using the method described earlier is shown as follows:

$$M = \frac{(30\times3)+(10\times5)+(150\times10)+(50\times2)}{240} = 7.2$$

$$q_{all} = 1 - \frac{1}{7.25} = 0.862$$

Thus, the PEC vector for q is (0.8, 0.1, 0.862) from (lowest PEC, highest PEC, all events). The overall starting point vectors are: (0.2, 0.8), (0.2, 0.1), (0.2, 0. 862) and so on.

4.3.4 Actual CPI Measurement

The response time for the application is dependent upon the three main factors - clock frequency, CPI, and number of cores. The clock frequency is the property of the architecture. The CPI and number of cores are dependent upon the application. The CPI

is the average of the number of clocks cycles each instruction takes for retirement. The total number of clock cycles includes both the active and sleep instances. In order words, it is the measure of instances where the pipeline is actively processing an instruction and in a sleep state (halt instruction). In the actual measurement, the ratio of VTune events “CPU_CLK_UNHALTED.TOTAL_CYCLES” and “INST_RETIRED.ANY” is used to quantify this particular part of the framework.

4.4 Model Tree

The second step in the framework (as shown in Figure 4.1) is the thread transitional probability pair extrapolation. This is performed using model trees which are a SML learning technique. This is readily available as M5´ package [52] in Weka (Waikato Environment for Knowledge Analysis), a machine learning toolkit. Weka [51] is free software available under the GNU General Public License. It supports several standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection.

There are four main segments in this section. The extrapolator design begins with comparison between MT with other numeric prediction learning techniques. This is explained in Section 4.4.2. The training data characteristics are presented in Section 4.4.1. In Section 4.4.3, the main stages for the MT algorithm are elaborated. The two main parts of this learning technique includes building the decision tree and regression model. Section 4.6 presents details on the characteristics for selecting the training data and validating the model before applying to the application size with the unknown performance.

4.4.1 Background

Some popular learning techniques for predicting floating point values are neural nets, instance-based learning, standard regression, induction trees and regression trees, among others as presented in [52]. Neural nets and instance-based learning are much powerful models. However, they suffer from opacity and do not provide much insight into the structure of the data. In comparison to regression and tree-induction methods, MT accounts better for the bias-variance trade-off (Figure 4.12). Bias and variance are the two main components of the prediction errors in most data fitting processes. Bias error is associated with erroneous assumptions in the learning algorithm, whereas variance is in relation to the sensitivity of the model to the small fluctuations in the training set.

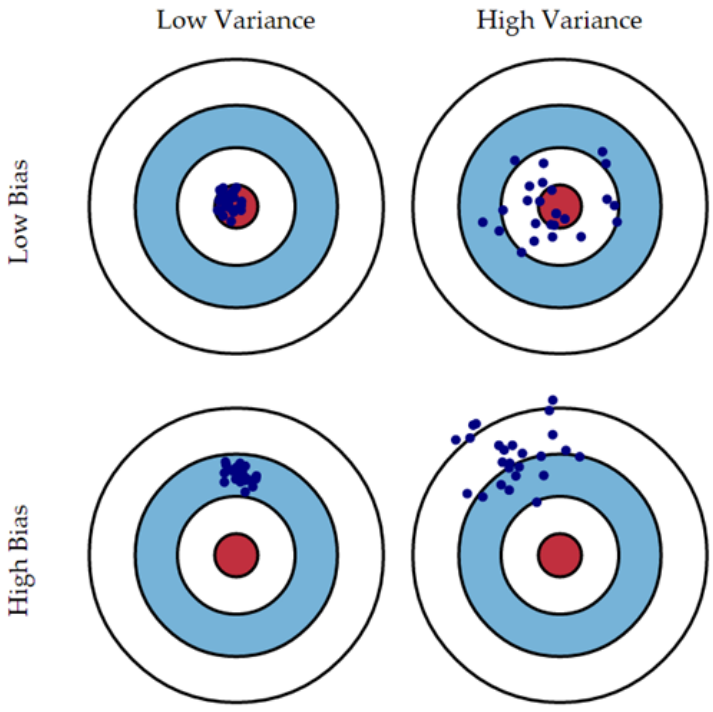


Figure 4.12: Graphical illustration of bias and variance [55]

At this point, it is important to understand the argument for selecting of using MT over logistic regression and induction trees. The differences between logistic and linear

regression are addressed as follows. The dependent variable in logistic regression can assume a fixed number of possible values. In comparison, the linear regression models have floating values as their dependent variable. In application, the latter of the two is used in regression settings while the latter is used for binary classification or multi-class classification.

Past research ([53] and [54]) has shown that both regression and tree-induction are superior methods. They are somewhat complementary and their relative performance depends on the size and the characteristics of the dataset. Logistic regression fits a simple (linear) model to the data, and the process of model fitting is quite stable. The method can be characterized as having a high bias but low variance. Tree induction, on the other hand, exhibits low bias but often high variance: it searches a less restricted space of models, allowing it to capture nonlinear patterns in the data, but making it less stable and prone to overfitting. This implies the model tends to fit according to the error in the training data.

In general, the linear models fit by logistic regression are preferable when the data is noisy or only few training examples are available, or when the data exhibits a linear structure. Tree induction is preferable on highly non-linear datasets, if enough training examples are available. The work that combines these two schemes (linear regression and tree induction) is termed as Model Trees, as presented in [50] and [54].

4.4.2 Basics of Model Tree Learning Technique

The model tree M5´ classifier is based on the pioneering work proposed in [50]. It constructs tree-based piecewise linear models as leaf nodes for multi-dimensional training data set. There are two main stages in building a model tree [52]. The first step

consists of building an ordinary decision tree by the divide-and-conquer method. A standard deviation based splitting criterion is used to create subsets for the set *T*. The variable *T* represents part of the training data to be explored for the particular node.

This procedure is applied recursively to the sub-sets with the goal of reducing an error function at each node. However, this division may produce overelaborate structures that must be pruned back. This consists of the second stage, in which the tree is pruned back by replacing a sub-tree with a regression surface. If this step is omitted and the target is taken to be the average target value of training examples that reach this leaf, then the tree is called a “regression tree”.

Most details of the original M5 algorithm [50] are not readily available and hence, it is reconstructed in [52]. M5´ also includes methods for dealing with numerical attributes and missing values, both of which are commonly occurring in the real world data sets. Due to these additions, M5´ performs better than the original M5 as demonstrated in the results in [52]. The subsequent paragraphs describe the main steps of M5´:

a) Building the Initial Tree

The splitting criterion is based on the standard deviation for each attribute that reaches that node. The error function associated with a particular node is calculated using the standard deviation reduction (SDR). The attribute which maximizes SDR is chosen for the subsequent sub-tree. The SDR is calculated using the following formula:

$$SDR = sd(T) - \sum_i \frac{|T_i|}{|T|} \times sd(T_i) \tag{9}$$

Where,

T : Set of examples that reach the node

$T_1, T_2,$: Sets that result from splitting the node according to the chosen

… attribute

b) Pruning the Tree

A linear model for each interior node of the unpruned tree is completed using standard regression techniques. Only attributes in sub-tree below the node are used for this purpose. The subsequent step is to simplify the resulting linear model by greedily dropping the terms. The goal is to reduce the expected error at each node for the actual versus predicted value in the test data. After finalizing the linear model for each node, the tree is pruned back from the leaves as long as the estimated error is reduced.

c) Smoothing

The final stage is a smoothing process which takes place whenever the model is used for prediction purposes. This step compensates for the sharp discontinuities that might inevitably occur between adjacent linear models at the leaves of the pruned tree. This may occur specially for models constructed from a small number of training instances. First the predicted value is computed using the leaf model. The next step is filter that predicted value along the path back to the root. In the process, each node undergoes a smoothing process by combining it with the value predicted by the linear model of that node.

4.4.3 Thread Transitional Probability Pair Characteristics

The Figure 4.13 and Figure 4.14 represent sample thread transitional probability curves with respect to iterations (in thousands). The plotted figures are from actual measurements for the smallest AS and all the MS. It is very evident that increase in the value of thread transitional probability pair (p and q) shows degradation in application

performance. The given characteristics can be correlated with actual timing measurements.

For example, on comparing the values for a particular MS (2T_2 - different L2-cache on same socket) for two threads, it is observed that (in Figure 4.13(a) and Figure 4.14(a)) the *p* values are the highest, but the value of *q* is the lowest. This means that threads tend to get stalled, but quickly come back to active state. In fact, the actual response time for 2T_2 is reported as a mid-range performing MS. An example (from these data points) is for the highest response time (worst performance) for 8T_3 (8 threads with 4 threads per L2-cache). It is seen that at steady state (in Figure 4.13(c) and Figure 4.14(c)), both *p* and *q* has the high values for this MS, in comparison to the other MSs with 8 threads.

As explained earlier in Figure 4.1, these serve as input data for the extrapolator in the second step in the framework. Hence, it is important to study the main characteristics of these thread transitional probability pair plots. They are listed as follows:

a) Basics

- Non-linear with continuous floating point data with respect to iterations. The probability values are in the range between 0 and 1, since probability can never be a negative value or greater than 1.
- The required output is extrapolation for AS for given iteration. This involves prediction outside of range of available training data information for a single attribute (Application Size).

b) Additional knowledge

- This is an instance of unsupervised learning. This means that no additional information is provided as part of the training data for the actual AS.

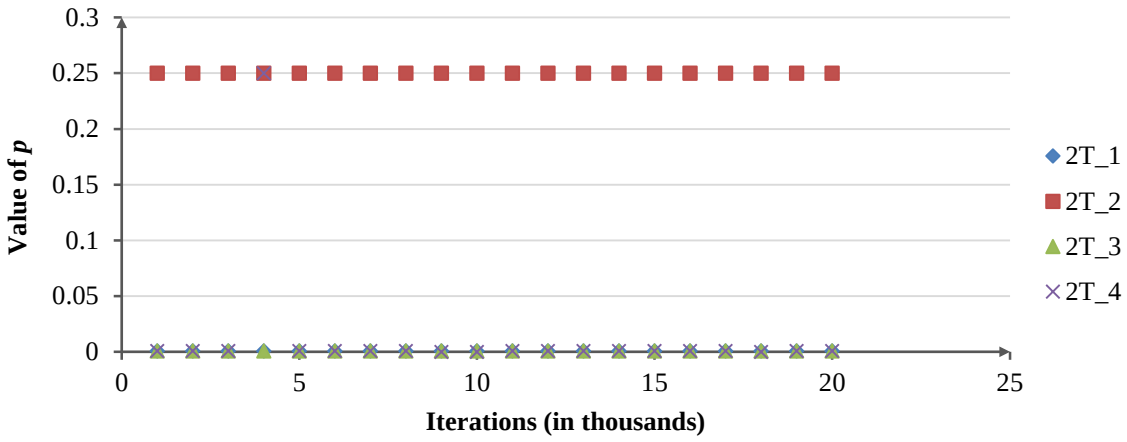
- Also, there is no other expert knowledge that can be built into the model. For example, two threads per core will always have a fixed behavior for the *p* versus iteration characteristic curve.

c) Balanced

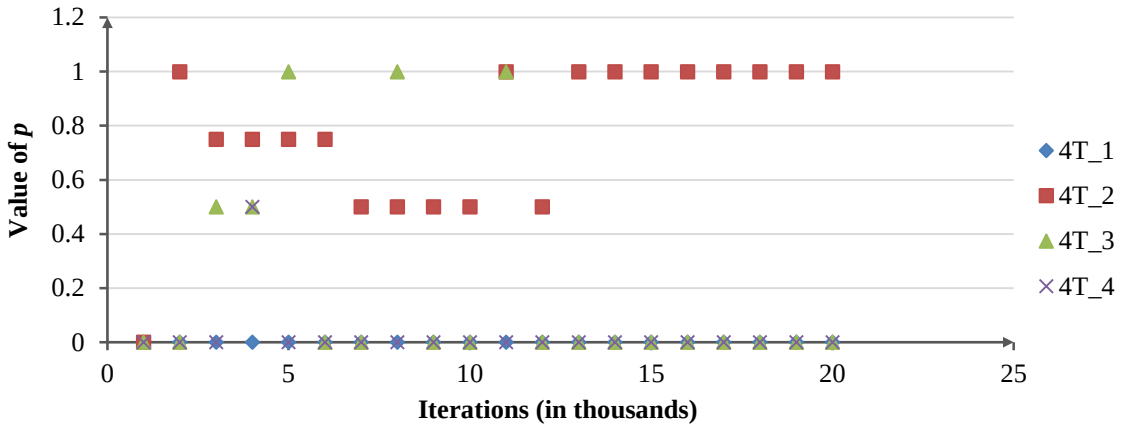
- The initial training model successfully calculates the probability values for all the iterations. Thus, there is equal amount of information available for all the MSs (different sub-classes) in the training set.
- Also, the training set class frequencies is the same as the operational conditions. This means that for an unknown *i-th* iteration value in some MS in the actual AS, there is information available for the same MS and same iteration in the training set data. This is in regards to the quality of information between training and actual test data.

d) Noisy data

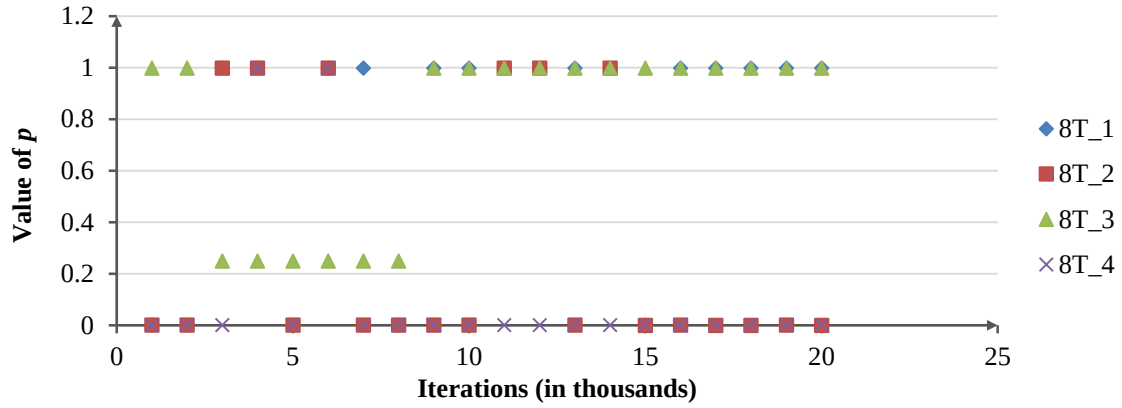
- Since, training data is generated from a clean input (VTune measurements) by the first MCM, the attribute labels are always correct.



(a) 2 threads

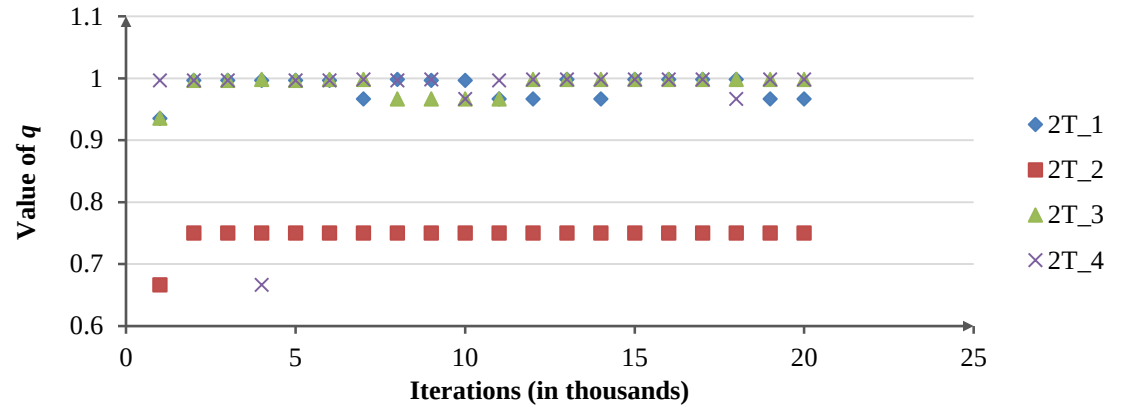


(b) 4 threads

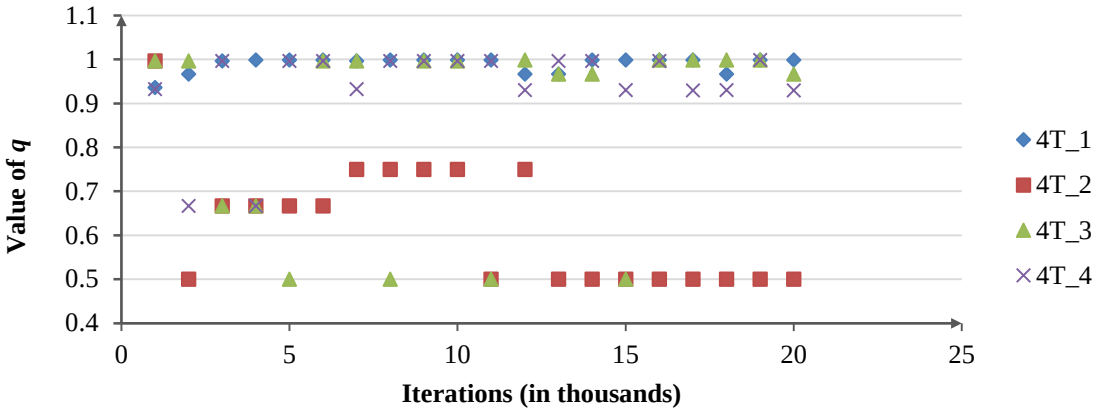


(c) 8 threads

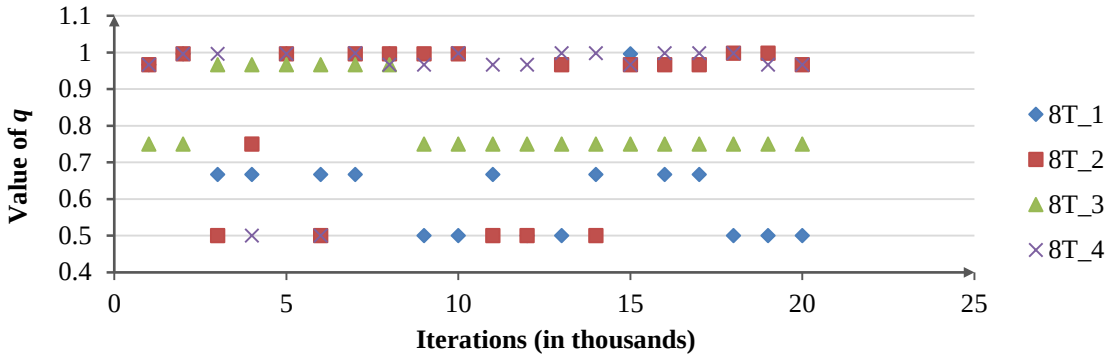
Figure 4.13: Thread transitional probability characteristics *p* with respect to iterations for different MSs for smallest application size



(a) 2 threads



(b) 4 threads



(c) 8 threads

Figure 4.14: Thread transitional probability characteristics *q* with respect to iterations for different mapping strategies for smallest application size

4.4.4 Training and Validation

In general, there is no straightforward method to precisely identify the smallest possible AS that can be sufficiently used as a training set. However, the pointers described in the following paragraphs act as sufficient guidelines for determining a suitable result. These pointers are divided into the two main parts of SML model design process: selection of the training data characteristics and validation methodology.

The training set is representative of all the thread stalling activities that take place in the actual AS. Hence, it should capture all these periodic patterns as accurately as possible. The characteristics of the training data set are listed below:

a) Each iteration finishes complete execution in exactly the same manner as the larger AS. This includes the concurrency, tasks per iteration, order of computation, number of synchronization barriers, pattern of memory accesses (including shared boundary terms) and number of MSs to be explored.
b) Training data should run for sufficiently long duration, so that it itself reaches the steady state performance and demonstrates all execution phases.
c) The training set AS should be larger than the cache size to incur sufficiently high number of cold misses. This should be held true, unless the actual AS is smaller than the cache size.
d) The other processes running are exactly intact as the real application.

The next aspect is the validation process, which involves accessing the model predictions for an independent data set. This prevents the model from representing a localized behavior or the noise in the data. The method used for this step in the current framework is called *cross-validation*.

In this process, at first the training data set is partitioned into multiple sub-sets. For the first part of the verification step, the first sub-set is used to confirm the predictions from the model generated with the second sub-set. The next step involves reversing the roles of the two sub-sets. This means that the first sub-set is used to build the model, while the second sub-set is verified for the predictions. The final result is the average of the two predictions. The confidence in the model can be improved by increasing the number of partitions in the training set and repeating the procedure equivalent number of times.

Cross-validation is performed on two levels in the present extrapolator. The first one is while creating the model on Weka. This step involves 10-fold cross-validation, where the

training set is randomly partitioned into 10 equal parts and verified according to the method described earlier. Mean squared error is used as a measure of fit for the model at this step. This part is in-built within the Weka tool. The second cross-validation is performed on the final framework results (predicted CPI). This means verifying the model for an AS when the empirical results (CPI) are already available. Standard deviation is used to summarize the errors in this step.

After this discussion, it important to examine case(s) with verification fails. The training data is regarded as unsuitable, when the model errors are outside the permissible limits (5% in the final results). Under such circumstances, the performance analyst needs to collect more training data. However, generating additional training data is not a very straightforward process. The performance analyst must pay attention to the guidelines on characteristics of training set, mentioned earlier in this section. In addition, it is important to localize the data points where the validation is actually incorrect.

For example, it may happen that the results are not matching for a particular MS, application iterations or the entire AS. In cases where the errors are restricted to a particular MS or iterations, more focus should be drawn to that particular subset of the training data. However, if the validation does not work for the complete AS, then the entire data is regarded as unsuitable. Under such circumstances, the small AS is increased to resemble actual AS. In addition, it is advisable to make the validation process more rigorous by increasing the number of sets for the cross-validation process.

4.5 Model Framework

This section combines all the individual components of the model framework described in Section 4.1. This consists of three primary parts as listed below:

a) Translating machine and application information into MCM states and the thread transition probabilities.

b) Extrapolating above thread transition probabilities for actual AS using MT model.

c) Calculating predicted CPI for actual AS from the extrapolated probabilities using MCM.

4.5.1 Translating Measurements into Probability

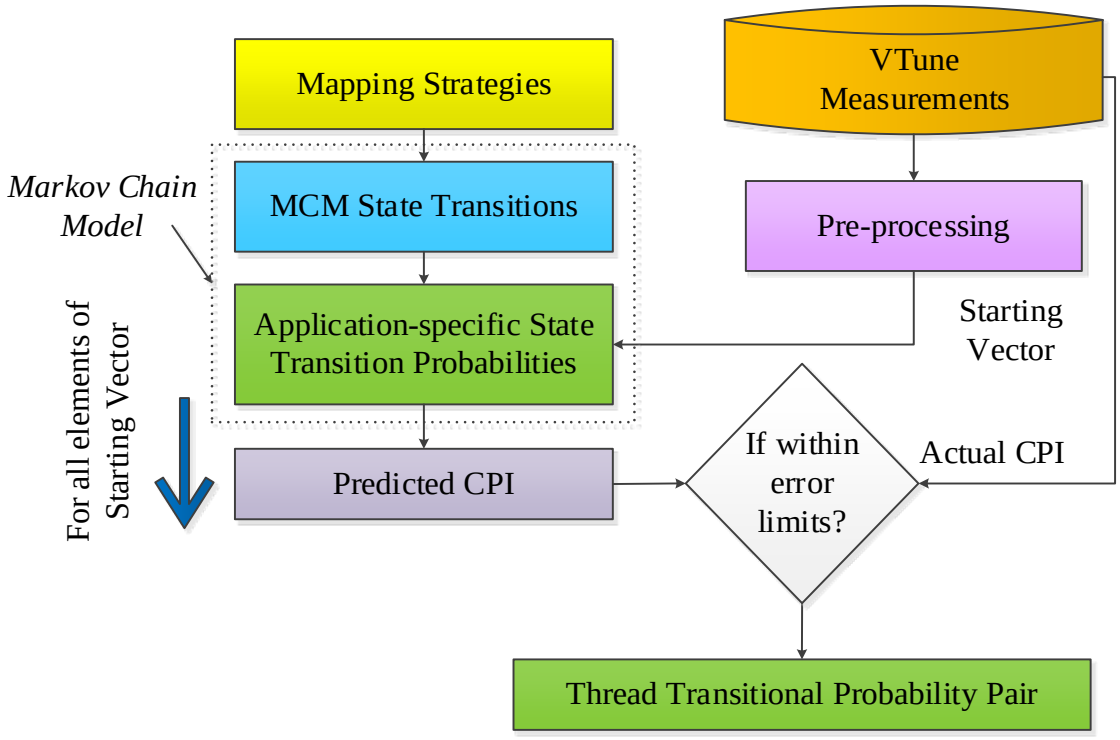


Figure 4.15: Workflow for generating training data (thread transitional probability pair)

This part (Figure 4.15) of model framework is the detailed description of Step 1 from Figure 4.1. It translates the small AS cycle-accurate VTune measurements into thread transitional probabilities. The MCM used here is constructed in Section 4.2.5. The states and probability expressions are dependent on the input MS information. The pre-processing methodology is illustrated in Section 4.3.2. The application probability with the minimum prediction error with respect to actual CPI from the VTune measurements

is selected. There is a correct solution pair for each of the MS for all the input iterations. This set of thread transitional probability pair will serve as the training data for the subsequent step in the model framework.

4.5.2 Model Tree Extrapolator

This section is about Step 2 in the model framework in Figure 4.1. This stage transforms small AS information to the actual data size performance data. All the input parameters from the MCM (AS, iteration, number of L2-cache, number of threads per L2-cache, *p*, *q* and the number of MC-states) are part of the training data. The model output at this stage is the extrapolated thread transitional probability pair. A sample output for this step is shown in Appendix D.

4.5.3 Actual Application Size Performance Prediction

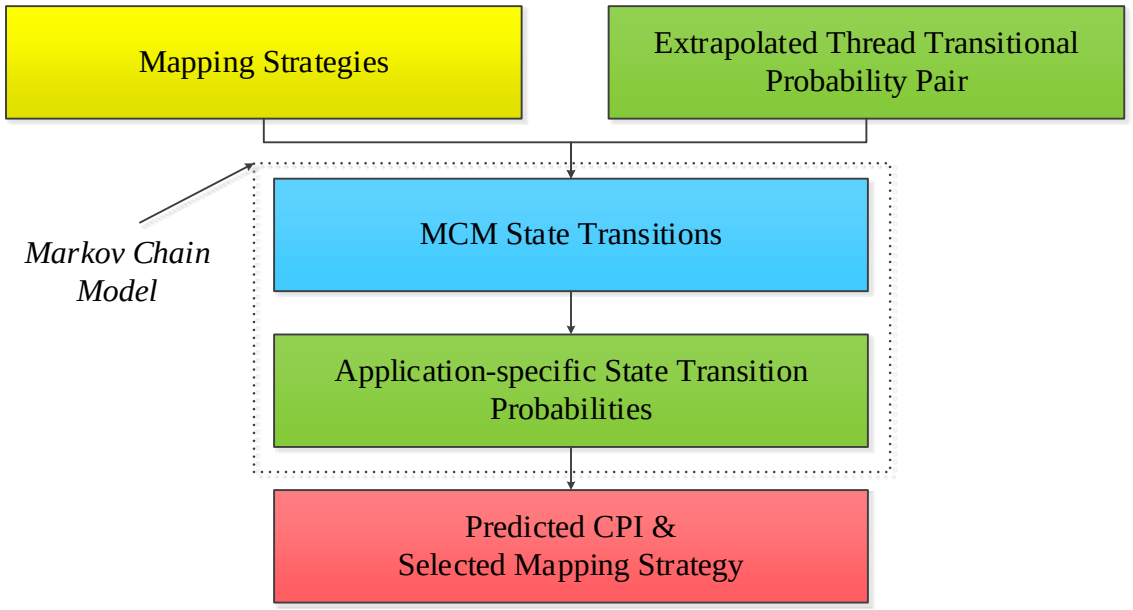


Figure 4.16: Workflow for actual application size performance prediction

This part of model framework is the description of Step 3 from Figure 4.1. This step transforms the extrapolated thread transitional probability pair (*p* and *q*) from the previous stage to actual performance prediction, as shown in the Figure 4.16. This MCM

is the same model as before, with exactly the same states and the probability expressions. However, this time it receives different values to denote the change in the system conditions.

4.6 Summary

Chapter 4 covers the details on the proposed framework methodology. It begins with a top-level overview of the main components of the framework. It is followed with a step-by-step development of each of the individual parts - pre-processing, MCM, MT-based extrapolator. The system-level architecture details, such as, the number of cores and the memory hierarchy information, are represented by the MCM states. Coarse-grained multithreading is shown as the allowable transitions among various states. The transition probabilities' values are determined from the program characteristics. They are representative of the common long latency operations such as cache misses, division, among others. MT-based extrapolator is used to estimate the behavior of the actual AS from the small AS performance information. This chapter also includes the basic parts of the learning algorithm, characteristics of the training set and the validation procedure. This chapter concludes with combining the individual pieces of the framework.

CHAPTER 5: RESULTS

The model framework results are presented in this chapter. There are three main sections. It starts with the application-specific parallel algorithm design and timing model results. The next section is on the thread-to-core prediction results, timing measurements from actual application execution and intermediate thread transitional probability results. Finally, it elaborates on the MCM prediction analysis based on the number of states and the probability values.

5.1 Timing-Based Model for Data Partitioning Strategies

The scientific computing application under study is an electromagnetics based two-dimensional Magneto-Static Wave (MSW) simulation. The MSW phenomena can be exploited to develop next generation devices, such as miniaturized circulators and spintronics applications. The current devices which can greatly benefit from this technology include magnetic amplifiers, frequency selective power limiters and phase shifters.

The computation order for the serial algorithm is first determined. The main constraints while designing the parallelizing strategy are reduced boundary sharing, decreasing stride penalties, reduced synchronization requirement and increased data sharing. These constraints are used to modify the order of computation. The next step is to develop the timing-based performance models. The main components of this model are computation, memory access and synchronization times. Finally, the actual speed-up results of the two parallelizing strategies are compared for identifying the better scheme.

5.1.1 MagnetoStatic Wave Simulation Description

This section describes the electromagnetic aspects of the Magnetostatic Wave simulation. The basic scheme for MSW algorithm builds on the standard Finite Domain Time Difference (FDTD) scheme [60]. In the traditional FDTD schemes, electric field intensity (*E*) and magnetic field intensity (*H*) values are computed for each cell. For a particular cell, the *E* calculations depend on the difference of the neighboring *H* values. Similarly, the *H* calculations are computed based on the differences of the neighboring *E* values. In the MSW calculations, two additional parameters are calculated, namely magnetic flux density (*B*) and magnetization. The magnetization parameter (*M*) is restricted to only certain cells. It is represented by the shaded area in Figure 5.1. The magnetic and electric sources introduce Gaussian pulses along the source "plane" row. The voltage and current calculations performed on the terminal "plane" row.

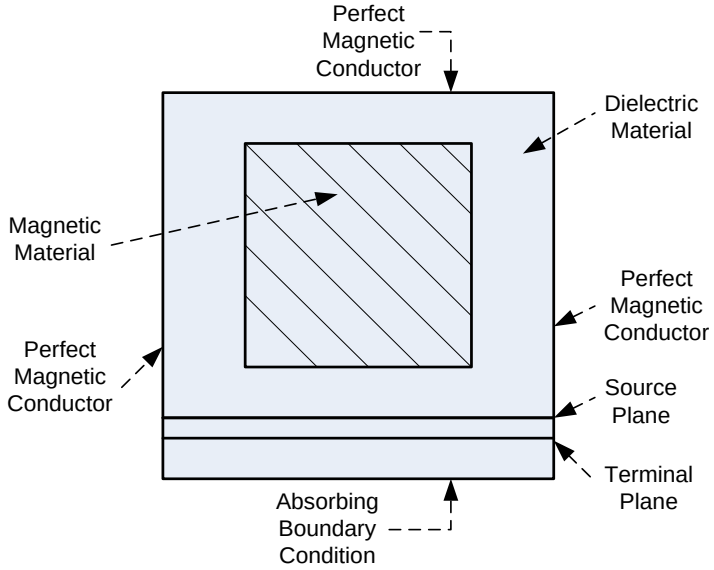


Figure 5.1: Magnetic material and dielectric distribution

For the simulation, a specimen of dimensions 25 cm by 25 cm is used. It is excited by a single port, with both soft electric and magnetic current density using transverse TEM

data, separated by half time-step. The magnetic material calculations are restricted to 17.5 mm by 8.75 mm (Figure 5.1), and are the most compute-intensive part of the algorithm. The *Liao* ABC (Absorbing Boundary Condition) scheme is used for the source boundary. The other three boundaries are covered by PMC (Permanent Magnetic Conductor). The overall number of cells is varied from 100 to 8000 cells in each of *x*-direction, *y*-direction and 1 cell in *z*-direction. The electric and magnetic properties of the material are mentioned in Table 5.1.

Table 5.1: Technical specifications of the material

Dielectric : Trans-Tech TTVG-1000		Ferrite: D-4, Cordierite		
	Length (mm)	Breadth (mm)	Height (mm)	Dielectric constant
Dielectric	25	25	1	14
Ferrite	17.5	8.75	1	4.5
Magnetic saturation : $4\pi M_s$ = 1000 G		Linewidth: ΔH = 795.77 Oe		
H_{ex} = 1x10^{19}		H_o = 700		

5.1.2 Serial Algorithm for MagnetoStatic Wave Simulation

For each iteration, the computation involves seven different parameters - electric field intensity (E_z), magnetic flux density (B_x and B_y), magnetization (M_x and M_y), and magnetic field intensity (H_x and H_y). The calculations progresses in the "leap-frog" fashion, both with respect to space and time based on *Yee's* scheme from [8]. The electric calculations (*E*) occur in the first half time step and the magnetic calculations (*B*, *M*, *H*) calculations happen in the second half time step. For a particular iteration, the serial algorithm calculates seven parameters in the given order:

$$E_z^{n+1/2} \rightarrow E_z^{n-1/2}, H_x^n, H_y^n \tag{10}$$

$$B_x^{n+1} \rightarrow B_x^n, E_z^{n+1/2} \tag{11}$$

$$B_y^{n+1} \rightarrow B_y^n, E_z^{n+1/2} \tag{12}$$

$$M_x^{n+1} \rightarrow M_x^{n-1}, M_x^n, M_y^n, H_x^n, H_y^n \quad (13)$$

$$M_y^{n+1} \rightarrow M_y^{n-1}, M_x^n, M_y^n, H_x^n, H_y^n \quad (14)$$

$$H_x^{n+1} \rightarrow B_x^n, M_x^n \quad (15)$$

$$H_y^{n+1} \rightarrow B_y^n, M_y^n \quad (16)$$

The notation B_x^n represents *x*-component of parameter *B* for time-step *n*. The left hand side to the arrow-head represents the parameter being calculated. The right hand side terms represent the parameters required for the calculation. The serial algorithm is shown in Figure 5.2. The term *NEND* represents the total number of iterations.

```
for niter = 1 to NEND
    calc_Ez
   calc_electric_source
   calc_voltage_current
   calc_Bx
   calc_By
   calc_magnetic_source
   calc_Mx
   calc_My
   calc_Hx
   calc_Hy
end  for
```

Figure 5.2: Serial MSW algorithm

As shown above, for $N \times N$ Yee cells in the specimen, the net calculation for a single iteration is of the order of $7N^2$ and the memory requirement is $11N^2$. The magnetization parameter computation is depends on two previous iterations. Hence, the memory requirement is very high. In general, it is the most compute and memory intensive among the four parameters.

There are multiple factors in this simulation which contribute to unbalanced computation load. First, the magnetization term (*M*) is restricted to only a certain portion of the entire matrix size (shaded portion in Figure 5.1). Secondly, the presence of extra computation at the source and terminal point results in an unbalanced load at different cells. Finally, the *Liao* boundary conditions are dependent on data from upper three rows of source boundary data.

5.1.3 Parallel Algorithm for MagnetoStatic Wave Simulation

The primary components of the execution time are computation, memory access, inter-processor communication and synchronization time. The main goal in this section is to build a composite performance analysis model to capture the effect of these terms. Apart from the efficient use of the available compute power, performance improvement greatly depends on efficient memory accesses. The objective is to partition the problem (to exploit task and data parallelism) to achieve the following goals:

a) Reusing data for multiple parameter computation.

b) Efficient use of multiple processors simultaneously.

c) Minimal interaction between neighboring processors.

d) Simplified synchronization between phases of computation.

Hence, to illustrate the spatial and temporal dependency for each parameter, dependency graphs are obtained. This information is used to estimate the memory requirements in terms of reads, additional boundary buffering, and need for data replication. Using these graphs, potential computation reordering schemes is generated. Additionally, this eliminates some of the barriers for the parallel algorithm, which were enforced by the serial version. Subsequently, the information from these graphs is used to

obtain the computational and communication bounds for the potential parallelizing schemes.

5.1.4 Dependency Graphs

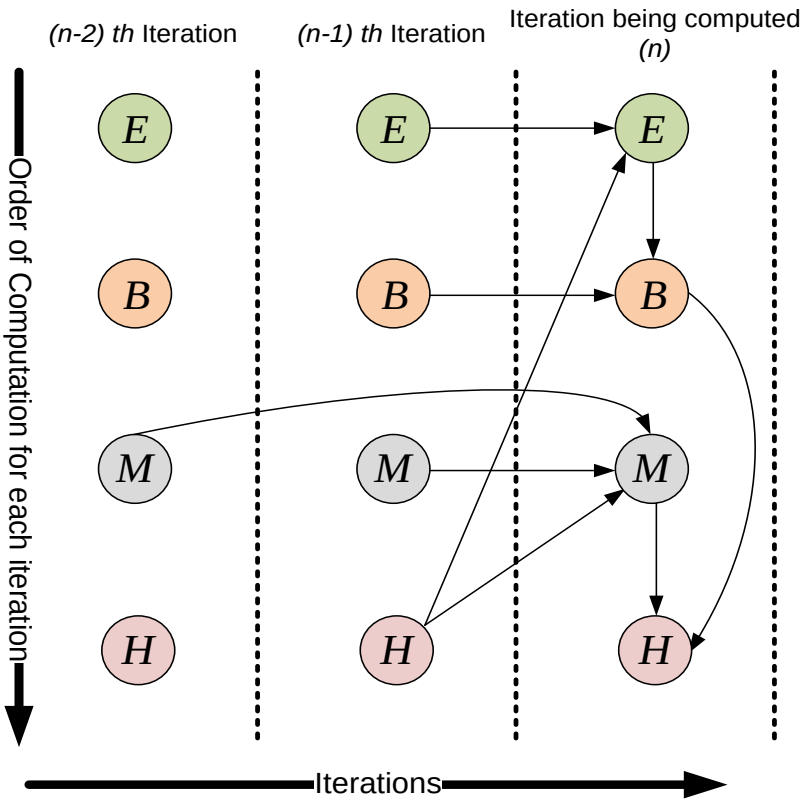


Figure 5.3: Task graph representing iteration variations

In order to examine the complex interrelationship between the various parameters, dependency graphs are derived. They represent the spatial and temporal dependency for each *Yee* cell. Figure 5.3 represents the task graph showing temporal relationship on a top level. Figure 5.4 represents the temporal dependency with respect to *x*, *y* and *z* components. Each node represents calculation of a particular parameter for that given time step. The edges represent number of terms of parent node used to calculate the child node.

In Table 5.2, the terms needed by the north, east, west and south neighbors for calculating a given term, are listed. This data is useful for estimating the intra-processor communication.

Table 5.2: Spatial dependency

Parameter Calculated	Boundary Terms needed			
	North	East	West	South
E_z	H_x	H_y	-	-
M_x	H_x, M_{yp}	H_y, M_{yp}	M_{yp}	M_{yp}
M_y	H_x, M_{xp}	H_y, M_{xp}	M_{xp}	M_{xp}
B_x	-	-	-	E_z
B_y	-	E_z	-	-
H_x	-	-	-	M_x
H_y	-	-	M_x	-

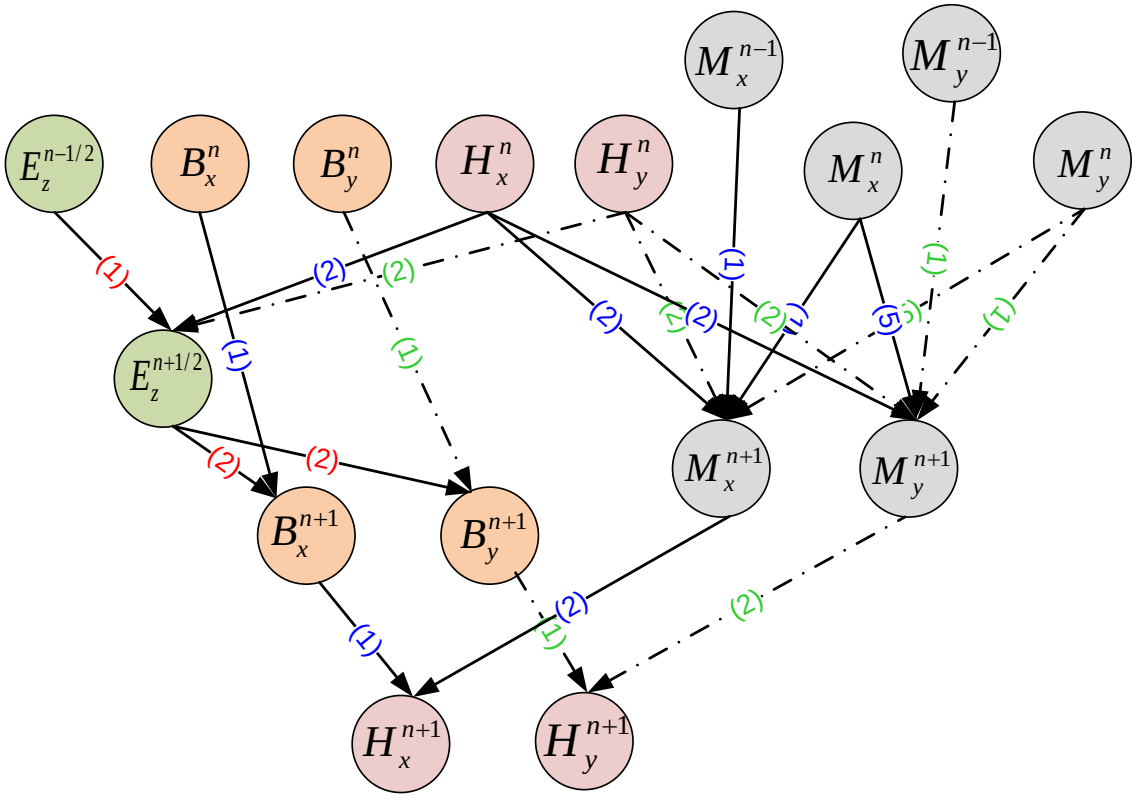


Figure 5.4: Temporal dependency graph

5.1.5 Preferred Order of Computation

In this section, a methodology is designed to reorder parameter calculations to extract optimal performance, while maintaining the correct order of computation. The parallel

performance usually suffers because of barrier wait time and cache misses. Hence, a part of the parallelizing strategy is reusing data fetched in the local memory for multiple parameter computation.

In the following paragraphs, all possible modifications in the computation order are developed:

1a) The MSW computation always begins with the E_z terms. In the serial computation all the *B* terms are generated after the *E* terms. In terms of memory, this translates to reuse of a single set of *E* terms, which is of the order of $O(N^2)$. Additionally, due to the presence of direct temporal dependency (Figure 5.4), there should be a barrier to ensure that all the *E* calculations are completed before calculating the *B* terms. Hence, a barrier is inserted.

Scheme 1a: $E_z \rightarrow$ Barrier 1 $\rightarrow B_x \rightarrow B_y$

1b) Since the *H* terms cannot be calculated without the *M* terms, a logical extension for Scheme 1a (above) is to compute the *M* terms. There are is no dependency between the *B* and the *M* term computations and, hence, so no barriers are placed.

Scheme 1b: $E_z \rightarrow$ Barrier 1$\rightarrow B_x \rightarrow B_y \rightarrow M_x \rightarrow M_y$

1c) The dependency graph in Figure 5.3 and Figure 5.4 shows that the *M* terms are dependent upon *H* terms from the previous iteration, and *H* terms depend on *M* terms from the *same* iteration. So, to extend this further, the *H* terms are added in the computation with barriers. This scheme allows reuse of $2N^2$ *M* terms for *H* computation and total reuse of $4N^2$ terms.

Scheme 1c: $E_z \rightarrow$ Barrier 1 $\rightarrow B_x \rightarrow B_y \rightarrow M_x \rightarrow M_y \rightarrow$ Barrier 2 $\rightarrow H_x \rightarrow H_y$

2a) An alternative to Scheme 1a (above) is to compute the *M* terms after the *E* computation, since it will involve reusing two set of *H* terms, which are of the order

$2N^2$. This may (or may not) translate to another barrier after the end of *M* calculation, depending on the next parameter. Thus, the following scheme is obtained:

Scheme 2a: $E_z \rightarrow M_x \rightarrow M_y$

2b) The next item is the *B* terms added to the preferred order of computation. Since all *E* computations should be completed before moving to the *B* calculations, a barrier is added as follows:

Scheme 2b: $E_z \rightarrow M_x \rightarrow M_y \rightarrow$ Barrier 1 $\rightarrow B_x \rightarrow B_y$

2c) Referring to Figure 5.3, there is no data dependency between the *B* and *M* parameter. Hence, there is no data reuse in any of the two schemes (1b and 2b). In order to incorporate such a feature, another alternative for Scheme 2b can be, calculating the *H* terms immediately after the *B* terms. However, this will introduce additional barriers to synchronize the entire computation.

Scheme 2c: $E_z \rightarrow M_x \rightarrow M_y \rightarrow$ Barrier 1 $\rightarrow B_x \rightarrow$ Barrier 2 $\rightarrow H_x \rightarrow B_y \rightarrow$ Barrier 3 $\rightarrow H_y$

It is important to note here that for the magnetic calculation it is necessary to calculate B_x terms before B_y. This is because of the presence of the magnetic source in the original algorithm.

3a) An alternative for Scheme 2b is to add the *H* terms with barriers. This scheme also allows reuse of $2N^2$ *B* terms for *H* computation.

Scheme 3a: $E_z \rightarrow M_x \rightarrow M_y \rightarrow$ Barrier 1 $\rightarrow B_x \rightarrow B_y \rightarrow$ Barrier 2 $\rightarrow H_x \rightarrow H_y$

The performance of the parallel implementation of the algorithm depends on two major factors - reduction in the barrier wait time and data reuse. The discussions are summarized in Table 5.3. Based on it, it is evident that the scheme 2d is the preferred order of computation. This is illustrated in Figure 5.5. The *E* and *M* calculations occur in

a single phase of computation. Then the values generated are used for the second phase of calculations (*B* and *H* computation).

Table 5.3: Summary of modified computation order performance

Schemes	Num of barriers	Order of terms reused
Scheme 1c	2	3 N^2
Scheme 2c	3	4 N^2
Scheme 3a	2	4 N^2

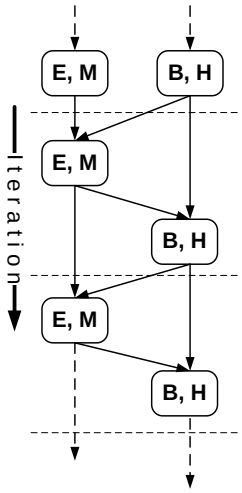


Figure 5.5: Selected parallel implementation

5.1.6 Design of Application-Specific Timing-Based Performance Model

In this section, the timing-based performance model for the parallel MSW algorithm for shared memory multicore is presented. Two data partitioning strategies are evaluated. The comparison parameters are computation, communication (main memory), synchronization and inter-processor communication times.

Broadly, there are two possible schemes of computation for a finite local memory, with each processor being assigned N^2/p cells (for every parameter):

a) This is a straightforward approach where the entire data size for each parameter gets calculated at a time. Then the computation moves to the next parameter calculations. For example, every processor calculates all the *E* terms for the given

domain size. The data size is constrained by the data it can fetch from the main memory. Once these values are written back to the main memory, it begins calculating the *E* terms for another region. This is repeated until all the *E* terms are computed for the entire matrix. The computation, then, shifts to calculating *B* terms for the first region. A similar pattern is repeated till all the parameters are completed. The order of computation is enforced by the barriers. Once all the parameters for a single iteration are completed, the same computation order is repeated for the next iteration.

b) All the (parameter) computations for a single iteration are completed in a small domain. Then it is directed towards immediate next domain and a similar pattern is repeated. For example, the *E* terms for a given domain size $n \times m$, (such that *n* and *m* are lesser than the application matrix size) is calculated. Subsequently, the *H* terms fetched to calculate *E*, is "re-used" to calculate some other parameter, before permanently flushing them out of the local memory.

The trade-off in the second scheme is increased data re-usage versus increased stride misses, C being a row major language. Secondly, there is a significant difference of MSW from the basic FDTD algorithm. The later cyclically uses one set of parameters (*E*) to calculate the other set and vice-versa. However, the requirement of current application is such that the multiple parameters are required for calculating a single one. It is also evident that the computation time for a single cell for *E*-*M* is lesser as compared to *B*-*H*, i.e. $t_{comp1} < t_{comp2}$. Also, there are data dependencies from two previous iterations. Hence, the data re-usage is not uniform over all the parameters for a single iteration. In summary, the data partitioning schemes should be designed to strike a balance between reducing

stride misses and maximizing data re-usage, by exploring the column boundary of the 2-D computation.

This design process begins with basic assumptions and description of the notations. The total memory required for each serial iteration is $11N^2$ (memory is required to store E_z, B_x, B_y, H_x, H_y, M_x, M_y and two previous iterations of M_x and M_y). Additionally, the entire data is maintained in the main memory and hence $11N^2$ is much lesser than the main memory size. But, the data size is sufficiently large that the L1 cache is inadequate for the entire data set and thus, $m << 11N^2$.

Notations:

N = Matrix row/column length for dielectric material.

N_{mag} = Matrix row/column length for magnetic material.

r = Ratio of N_{mag} / N, $r < 1$.

t_{comp1} = Computation time for unit cell for the parameters E, B and H.

t_{comp2} = Computation time for unit cell for the parameter M.

t_{barr} = Barrier time.

m = L1-cache size (local memory)

p = Number of processors available.

M = Main memory available.

q = Number of rows assigned to each processor per cycle of computation.

In each of the parallelizing strategies identified so forth, the net memory requirement is first computed, assuming an infinite memory. This also accounts for the data which is already present in the local memory. After finishing computation for one phase, the entire data is flushed out to read in new data for the next phase of computation. Then, on the basis of the memory requirement of each phase (E, M or B, H computation), the value of

the maximum number of rows that can be brought in at a time (blocking) can be evaluated. Each “cycle” of computation refers to a block of data being computed for a single phase (say, *E* and *M*). Finally, the total time taken for the four major steps: computation, main memory communication, inter-processor communication and synchronization for all the cells are estimated.

5.1.7 One-Dimensional Data Partitioning

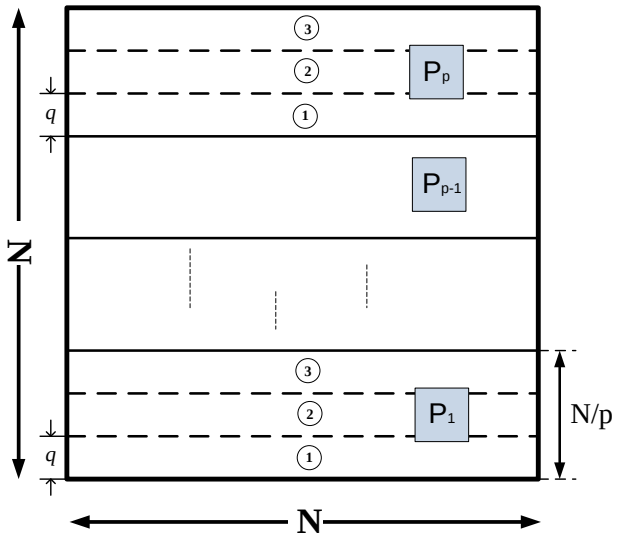


Figure 5.6: One-dimensional partitioning for $N/pq = 3$

```
for niter = 1 to END
  calc_Ez
  if source port thread
    calc_electric_source
  calc_Mx
  calc_My
  barrier 1
  calc_Bx
  calc_By
  if source port thread
    calc_magnetic_source
    calc_voltage_current
  calc_Hx
  calc_Hy
  barrier 2
End for
```

Figure 5.7: Parallel algorithm for one-dimensional data partitioning

Figure 5.6 represents the one-dimensional data partitioning scheme. In this case, the trade-off of stride miss versus data re-usage is in favor of the former. Also, there are barriers to impose global synchronization as shown in Figure 5.7. The subsequent paragraphs present mathematical representation of the three major components of the timing performance - computation, communication and synchronization.

The total memory requirement per phase per processor is the maximum of the requirement for each phase i.e. *E*-*M* or *B*-*H* as mentioned in Table 5.4. The number of

rows assigned to each processor per cycle of computation (q) can be computed using the constant L1-cache size constraint. This value can be used to estimate the three important timing parameters - computation, communication and synchronization.

Table 5.4: Memory requirement for each parameter for one-dimensional partitioning

Parameter	Boundary buffering	Parameters already in local memory	Net memory requirement
E_z	H_x	-	$\frac{3N^2}{p} + N$
M_x	H_x, M_{yp} - 2 terms	-	$\frac{6N_{mag}^2}{p} + 3N_{mag}$
M_y	M_{xp} - 2 terms	E_z, H_x, H_y, M_x, M_{xp}, M_{xpp}, M_{yp}	$\frac{8N_{mag}^2}{p} + 5N_{mag}$
B_x	E_z	-	$\frac{2N^2}{p} + N$
B_y	-	E_z, B_x	$\frac{3N^2}{p} + N$
H_x	M_x	E_z, B_x, B_y	$\frac{5N^2}{p} + 2N$
H_y	-	E_z, B_x, B_y, H_x, M_x	$\frac{7N^2}{p} + 2N$

The total number of cycles of computation required to complete per iteration of computation is $\frac{N^2/p}{Nq} = \frac{N}{pq}$. The values estimated for the two phases can be summed to be calculate the “per cycle” and “per iteration” timing parameters. These are summarized in Table 5.5 and 5.6.

Table 5.5: Timing performance model (per phase) for one-dimensional partitioning

Timing Parameters	Phase 1	Phase 2
Computation	$(Nqt_{comp1}) + (2N_{mag}qt_{comp2})$	$4Nqt_{comp1}$

Main memory accesses	$3Nq + 8N_{mag}q$	$7Nq$
Inter-processor communication	$N + 5N_{mag}$	$2N$
Synchronization	t_{barr}	t_{barr}

Table 5.6: Timing performance model for one-dimensional partitioning

Timing Parameters	Per cycle (for both phases)	Per Iteration
Computation	$(5Nqt_{comp1}) + (2N_{mag}qt_{comp2})$	$\left(\frac{5N^2}{p}t_{comp1} + \frac{2N_{mag}^2}{p}t_{comp2}\right) = \left(\frac{N^2}{p}\right)(5t_{comp1} + 2r^2t_{comp2})$
Main memory accesses	$10Nq + 8N_{mag}q$	$(10Nq + 8N_{mag}q) * \left(\frac{N}{pq}\right) = \frac{N^2}{p}(10 + 8r)$
Inter-processor communication	$3N + 5N_{mag}$	$(3N + 5N_{mag}) * \left(\frac{N}{pq}\right) = \frac{N^2}{pq}(3 + 5r)$
Synchronization	$2t_{barr}$	$2t_{barr} * \left(\frac{N}{pq}\right)$

5.1.8 Two-Dimensional Data Partitioning

Two-Dimensional data partitioning: In Figure 5.8, the two-dimensional data partitioning scheme is shown. This scheme differs from the former case in the sense that it reuses the data much more efficiently. The timing performance parameters are derived in a similar method as shown in Section 5.1.7. Hence, the end results are presented in Table 5.7.

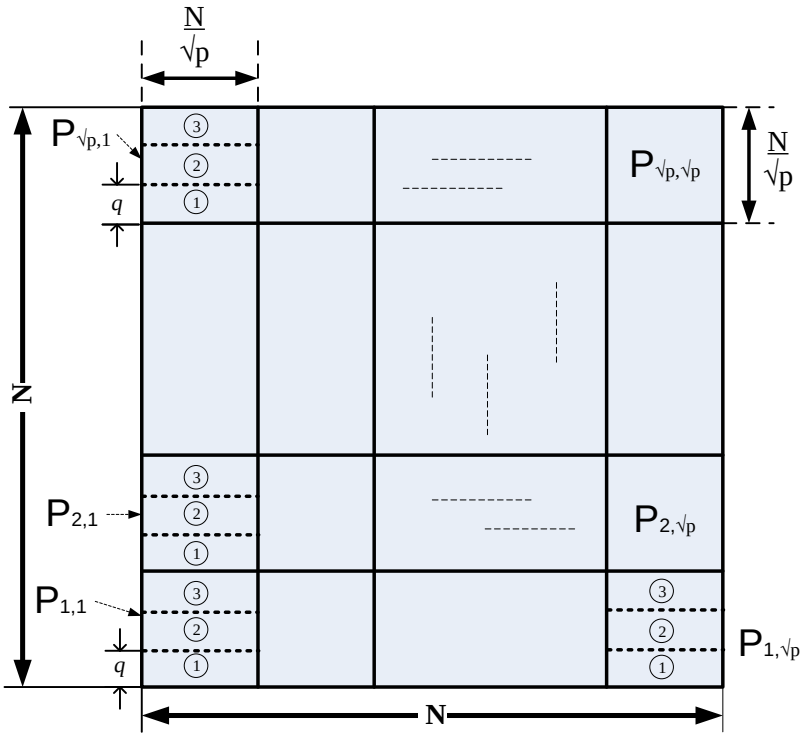


Figure 5.8: Two – dimensional Partitioning for $\frac{N}{q\sqrt{p}} = 3$

Table 5.7: Timing performance model for two-dimensional partitioning

Timing Parameters	Per cycle (for both phases)	Per Iteration
Computation	$\frac{5Nq}{\sqrt{p}} t_{comp1} + \frac{2N_{mag}q}{\sqrt{p}} t_{comp2}$	$\left(\frac{5N^2}{p} t_{comp1} + \frac{2N_{mag}^2}{p} t_{comp2}\right) = \left(\frac{N^2}{p}\right)(5t_{comp1} + 2r^2 t_{comp2})$
Main memory accesses	$\frac{10Nq}{\sqrt{p}} + \frac{8N_{mag}q}{\sqrt{p}}$	$\left(\frac{10Nq}{\sqrt{p}} + \frac{8N_{mag}q}{\sqrt{p}}\right)\left(\frac{N}{q\sqrt{p}}\right) = \left(\frac{N^2}{p}\right)(10 + 8r)$
Inter-processor communication	$\frac{5N}{\sqrt{p}} + \frac{10N_{mag}}{\sqrt{p}}$	$\left(\frac{5N}{\sqrt{p}} + \frac{10N_{mag}}{\sqrt{p}}\right)\left(\frac{N}{q\sqrt{p}}\right) = \left(\frac{N^2}{pq}\right)(5 + 10r)$
Synchronization	$3t_{barr}$	$(3t_{barr})\left(\frac{N}{q\sqrt{p}}\right)$

5.1.9 Results

The parallel code performance for “O3” optimized is measured against un-optimized serial code. Performance is observed for up to eight cores. All data used is single precision values and it is simulated for up to 5000 iterations. CPU utilization for each data partitioning specimen is compared with respect to that of the serial code.

Maximum speed-up (Figure 5.9) and CPU utilization (figure 5.10) are observed for maximum number of cores irrespective of data partitioning scheme, since more computing resources are available for each thread. Secondly, as compared to two-dimensional, one-dimensional data partitioning usually outperforms in the parallel MSW algorithm. This can be explained by comparing Table 5.6 and 5.7, where it is evident that inter-processor and barrier times are lesser in the better implementation.

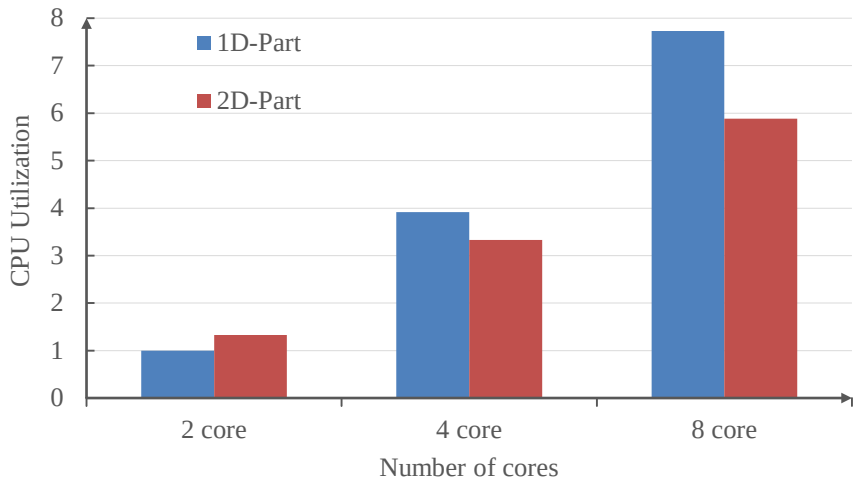


Figure 5.9: CPU Utilization versus number of cores for data size 8001x8001

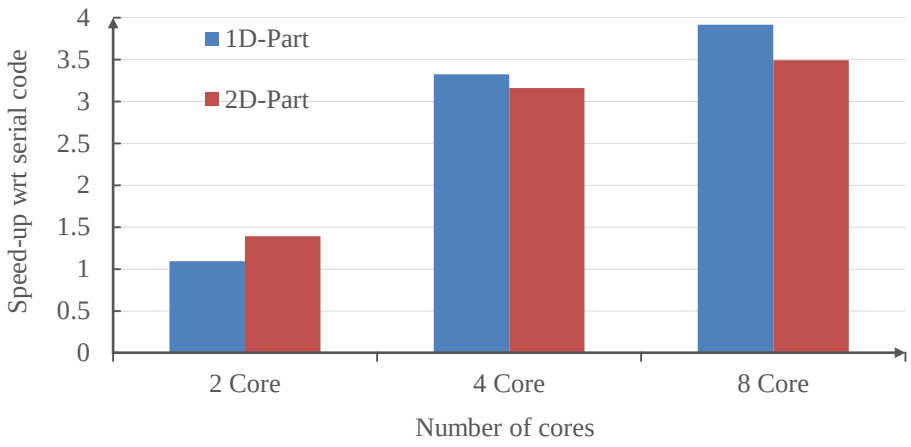


Figure 5.10: Speed-up versus number of cores for data size 8001x8001

5.2 Thread-to-core Model Results

There are nine main parts in this section. It begins with the experimental set-up and a brief on the MSs under exploration. The next three parts are on the overview of the electromagnetics application experiments for the model validation. The fifth part briefly describes the next application, Fast Fourier Transform (FFT). The sixth and seventh section is about the CPI prediction from the thread-to-core model framework. The eighth section contains information on the elapsed time measurements for the applications. Finally, predicted result discussions are presented in the last part.

5.2.1 Experimental Set-Up

VTune analyzer is installed on Linux (Red Hat release 5.4 (Tikanga)) with g++ (GCC) 4.1.2. Processor architecture details are provided in Table 3.1 and Section 3. The code is run with the compiler flag -O3. The parallelization is implemented using POSIX threads. Performance of four different configurations of three different thread counts (two, four and eight) is explored for the applications. The MSs in the experiments are listed in Table 5.8. The different MSs are to study the thread performance bottlenecks, within the same amount of designated parallel work.

Table 5.8: Mapping strategy symbols

Num. of threads	Mapping Strategy	Symbols
2 threads	Different L2-cache with different socket	2T_1
	Different L2-cache with same socket	2T_2
	2 threads per core	2T_3
	Same L2-cache with same socket	2T_4
4 threads	4 threads per core	4T_1
	1 thread per L2-cache	4T_2
	2 threads per core ALL on one socket	4T_3
	2 threads per L2-cache two on each socket	4T_4
8 threads	1 thread per core	8T_1
	2 threads per core	8T_2
	4 threads per core	8T_3
	8 threads per core	8T_4

5.2.2 MagnetoStatic Wave Benchmark Overview

MSW serial and parallel algorithm is described in Section 5.1. MSW1 stands for data size 60 MB approximately with around 600 data points for each row and column of matrix. MSW2 has double the number of row (and column) elements with respect to MSW1. The application size can be worked out with similar logic for MSW3 and MSW4. For model validation, MSW4 is chosen as the test data. MSW3 results are shown for verification.

5.2.3 N-Body Solver Benchmark Overview

An n-body problem is used to find the positions and velocities of a collection of interacting particles over a period of time. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation.

The output is typically the position and velocity of each particle at a sequence of user-specified times. Here, the N-Body solver is used to simulate the motions of planets or stars. The algorithm [61] uses Newton’s Second Law of Motion and his law of universal gravitation to determine the positions and velocities. The derivate terms are calculated

with Euler's method. The compute complexity for this algorithm, for *N* problem size, is *5N* and the memory requirement is *9N*.

5.2.4 Benchmark Mix for MagnetoStatic Wave Simulation

This section describes the application mix details used for the model validation. The driving idea in the validation process is to estimate the performance of a single application under different conditions. This is to gauge its change in performance when it is running by itself, in comparison to, sharing the resources (cache or network) with another application. However, only a reduced set of MSs for N-Body Solver is chosen for the second set of application mix. The scope of this dissertation is to display the validity of the thread-to-core model, rather than optimizing the number of experiments with multiple applications. Hence, examination of all possible MS combinations for multiple applications is left as a future work. The two set of experiments are listed below:

a) Application mix 1: all cores reserved for only MSW
b) Application mix 2: MSW with N-Body Solver in the background

In the current Clovertown machine, the odd numbered cores are mapped on the first socket and the even numbered cores are mapped on another socket. And the core arrangement is made in such a way that the first four cores (0 to 3) are connected to an individual L2-cache. In regard to this core numbering information, the binding details for the different MSs are listed in Table 5.9.

Table 5.9: Thread-to-core binding for application mix 1 (only MSW)
(a) 2 Threads

Mapping Strategy	Thread 0	Thread 1
2T_1	0	1
2T_2	0	2
2T_3	0	0
2T_4	0	4

(b) 4 Threads

Mapping Strategy	Thread 0	Thread 1	Thread 2	Thread 4
4T_1	0	0	0	0
4T_2	0	2	1	3
4T_3	0	0	4	4
4T_4	0	4	1	5

(c) 8 Threads

Mapping Strategy	Thread 0	Thread 1	Thread 2	Thread 4	Thread 5	Thread 6	Thread 7	Thread 8
8T_1	0	4	2	6	1	5	3	7
8T_2	0	0	4	4	2	2	6	6
8T_3	0	0	0	0	4	4	4	4
8T_4	0	0	0	0	0	0	0	0

In the second benchmark mix, the two threads MSs of MSW, i.e. 2T_1, 2T_2, 2T_3 and 2T_4 are paired with two threads per core for the N-Body Solver. The four threads are paired with 4 threads per core and so forth.

5.2.5 Fast Fourier Transform Overview

FFT decomposes a periodic function in time into its component frequencies. For given N points, the naïve Fourier operation involves $O(N^2)$ arithmetical operations, whereas FFT completes it in $O(N \log_2 N)$ operations [62]. The memory requirement for the current implementation is $2N^2 + 4N$.

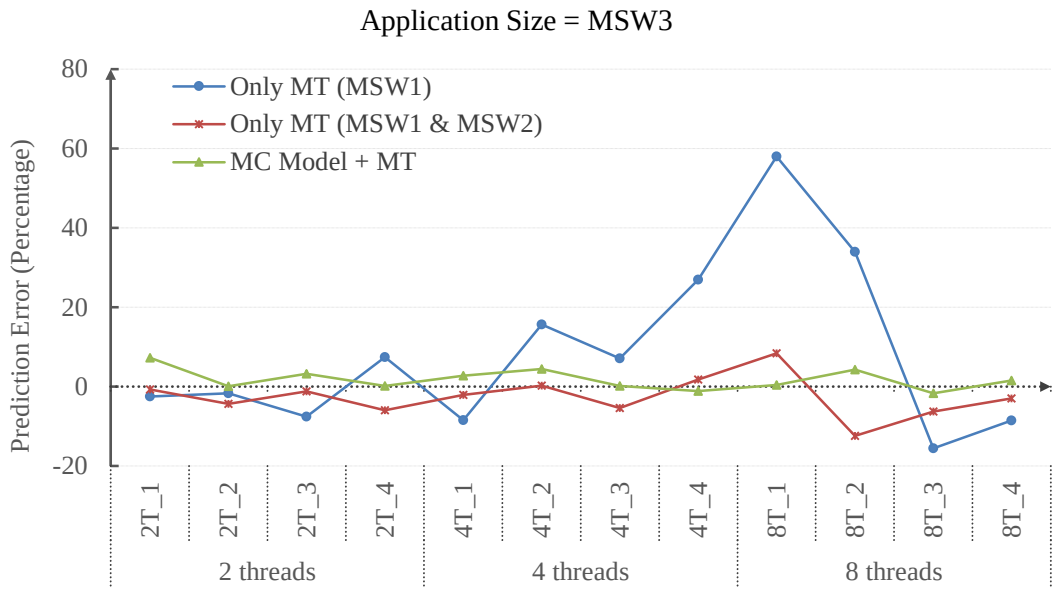
In this experiment, all the 12 MSs as shown in Table 5.8 are explored. The thread-to-core bindings are based on core id information in Table 5.9. The smallest data set has a size of 1,024, and is represented as FFT_1. This translates to a memory size of 8 MB approximately. This increases by a factor of 2 for the subsequent sets. So, the complete training data includes sizes: 2048 (FFT_2), 4,096 (FFT_3) and 8,192 (FFT_4). The validation is performed for N = 16,384 (FFT_5) and the actual AS is 32,678 (FFT_6). The memory footprint for the actual AS is about 8 GB. The different training data is

chosen to compensate for the lack of iteration information in this application. In the experiment, FFT is executed as a singular application, without any other user applications running in background.

5.2.6 Results for MagnetoStatic Wave Simulation

The error in prediction results with increasing core and application sizes are presented in Figure 5.11 and Figure 5.12 for application mix 1 and 2, respectively. The reported values are the percentage error with respect to actual measurements from VTune for steady state CPI (around 20,000 iterations or more). Results are shown from three different models (a) MT with only smallest AS for training (MSW1), (b) MT with the two small AS for training, (c) MCM and MT.

Intermediate results in the form of extrapolated probability pair for actual application size from MT is shown in Table 5.11. The same information for application mix 2 is presented in Table 5.13.



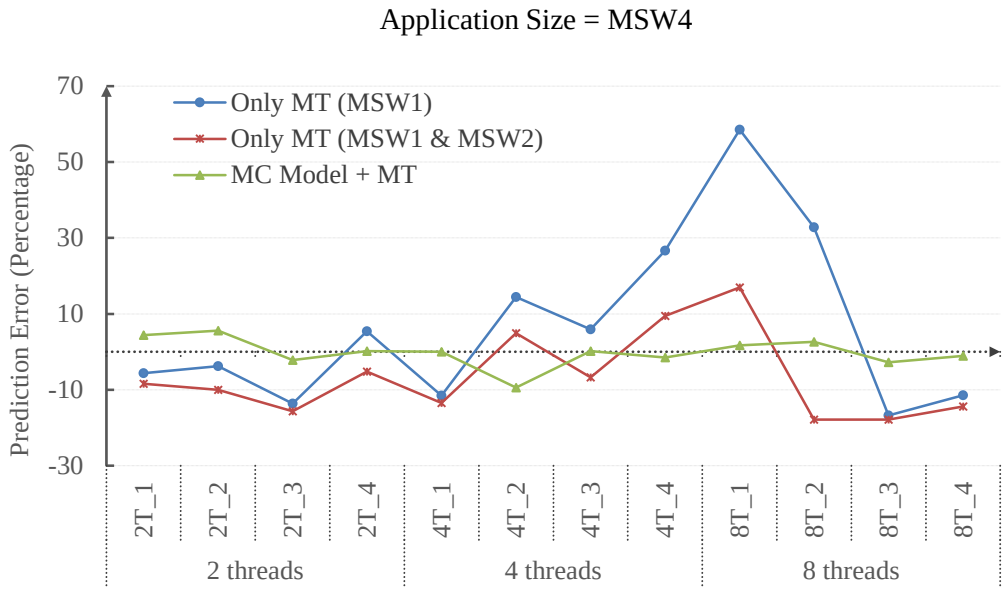


Figure 5.11: Prediction error for steady state CPI versus mapping strategies with core and application scaling for application mix 1 (only MSW)

Table 5.10: Summary of prediction error values for application mix 1

	Only MT (with MSW1)	Only MT (with MSW1 and MSW2)	MCM + ML
Average error	7.92589246	-5.4679317	-0.1681387
Std Deviation	22.2158795	11.4576263	3.866066

Table 5.11: Transitional probability variation for actual application size data (MSW4) for application mix 1

Mapping Strategies		Training Data		Actual AS Data	
		p	*q*	*p*	*q*
2 threads	2T_1	0.001	0.966667	0.001	0.969753
	2T_2	0.25	0.75	0.275272	0.763373
	2T_3	0.001	0.998333	0.001	0.982736
	2T_4	0.001	0.998333	0.001	0.972849
4 threads	4T_1	0.001	0.998333	0.001	0.984722
	4T_2	0.999	0.5	0.9	0.673064
	4T_3	4.93E-06	0.966667	0.002003	0.931579
	4T_4	0.999	0.666667	0.973859	0.666667
8 threads	8T_1	0.5	0.966667	0.532037	0.966667
	8T_2	0.25	0.966667	0.269423	0.966667
	8T_3	0.999	0.75	0.9999	0.729884
	8T_4	0.001	0.966667	0.010698	0.988694

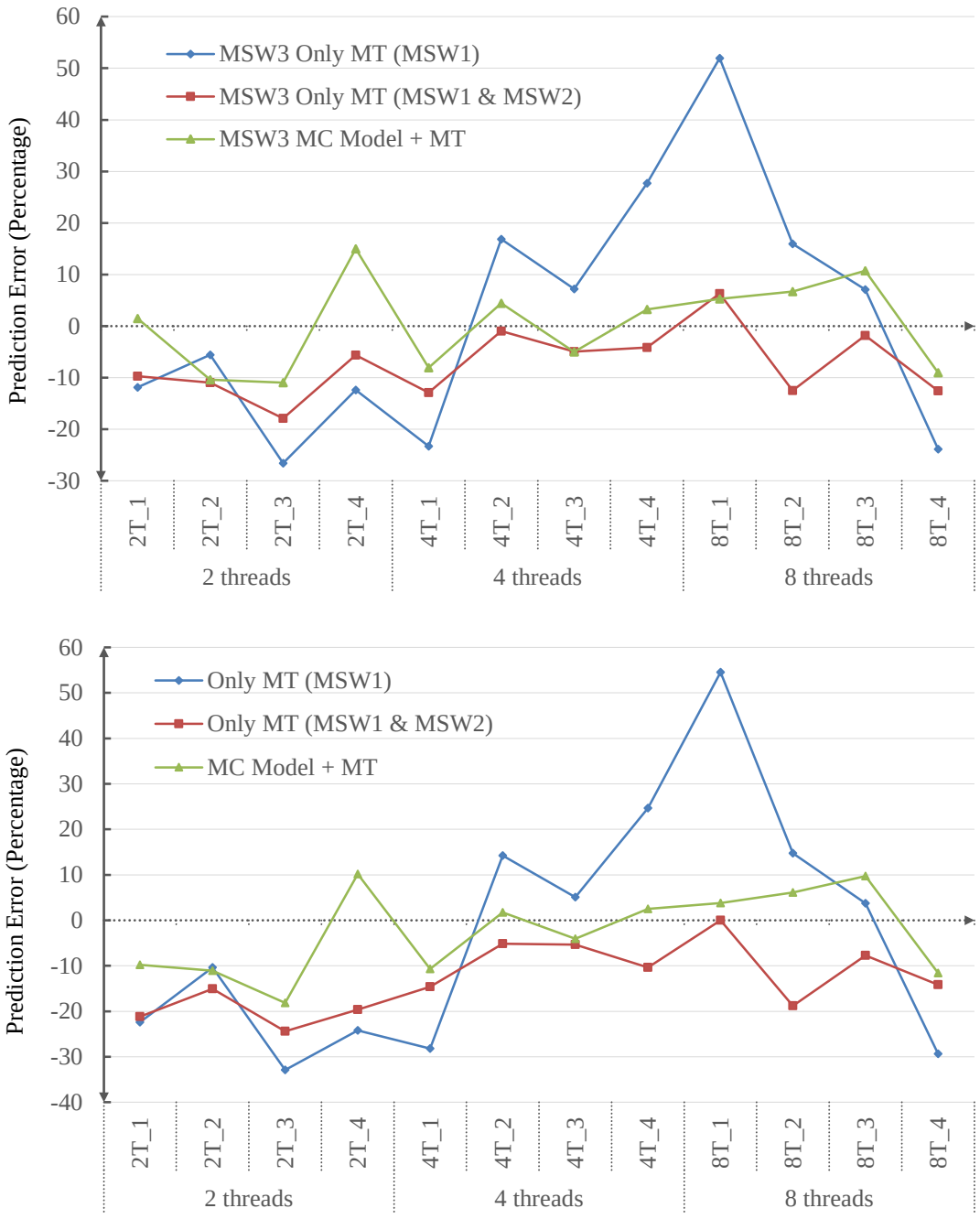


Figure 5.12: Prediction error for steady state CPI versus mapping strategies with core and application scaling for application mix 2 (MSW with N-Body Solver in the background)

Table 5.12: Summary of prediction error values for application mix 2

	Only MT (with MSW1)	Only MT (with MSW1 and MSW2)	MCM + ML
Average error	-2.5501748	-13.058848	-2.618146
Std Deviation	26.7915402	7.42696016	9.487566

Table 5.13: Transitional probability variation for
actual application size data (MSW4) for application mix 2

Mapping Strategies		Training Data		Actual AS Data	
		p	*q*	*p*	*q*
2 threads	2T_1	0.001	0.998333	0.001	0.956443
	2T_2	0.75	0.2	0.737062	0.241648
	2T_3	0.001	0.966667	0.001	0.999
	2T_4	0.0249855	0.966667	0.009133	0.95283
4 threads	4T_1	0.0000739	0.998333	0.001	0.982944
	4T_2	0.999	0.7	0.929972	0.694483
	4T_3	0.75	0.75	0.873477	0.730283
	4T_4	0.999	0.8	0.970366	0.78102
8 threads	8T_1	0.5	0.966667	0.688248	0.952814
	8T_2	0.25	0.966667	0.220528	0.973659
	8T_3	0.99	0.8	0.9	0.784496
	8T_4	5.99E-06	0.966667	0.019233	0.953487

5.2.7 Results for Fast Fourier Transform

This section describes the CPI p rediction results for FFT with validation and actual AS. It is organized in a manner similar to the previous section. The dependent parameter in the *y*-axis represents the difference of predicted CPI with respect to the actual CPI measurements from VTune. These are compared with the following models generated from the direct use of MT:

(a) FFT_1

(b) FFT_1 and FFT_2

(c) FFT_1 to FFT_3

(d) FFT_1 to FFT_4

Intermediary results in form of the transitional probability terms are presented in Table 5.15.

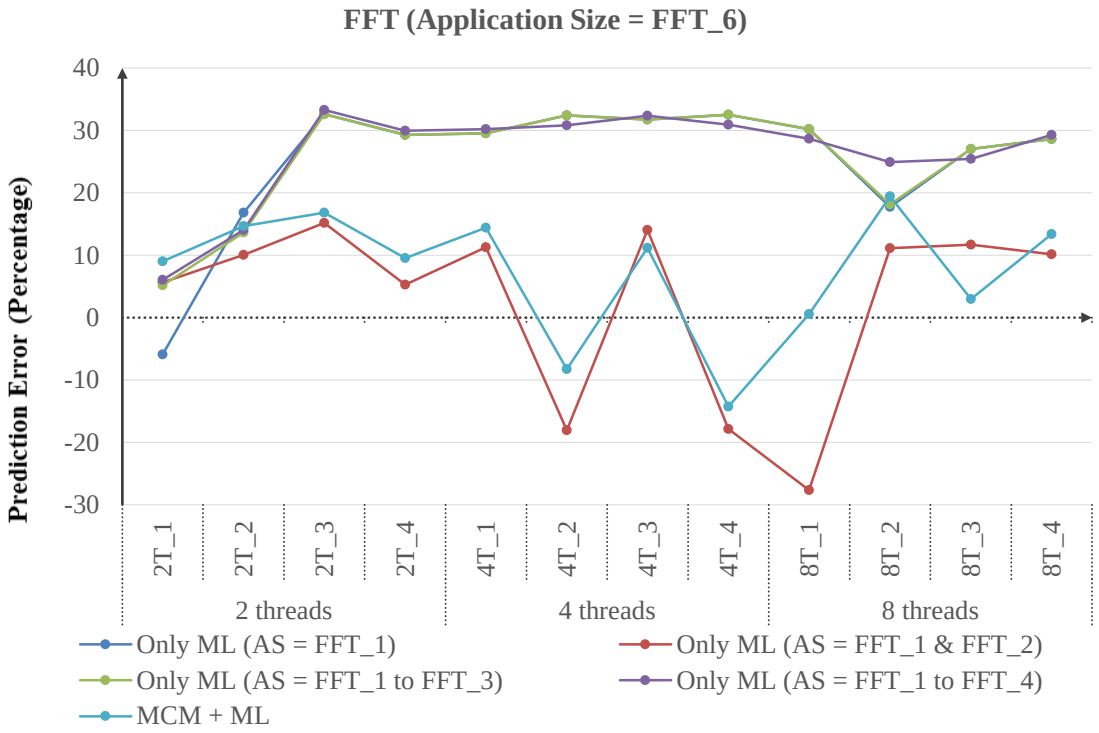


Figure 5.13: Prediction error for CPI versus mapping strategies with core scaling for Fast Fourier Transform

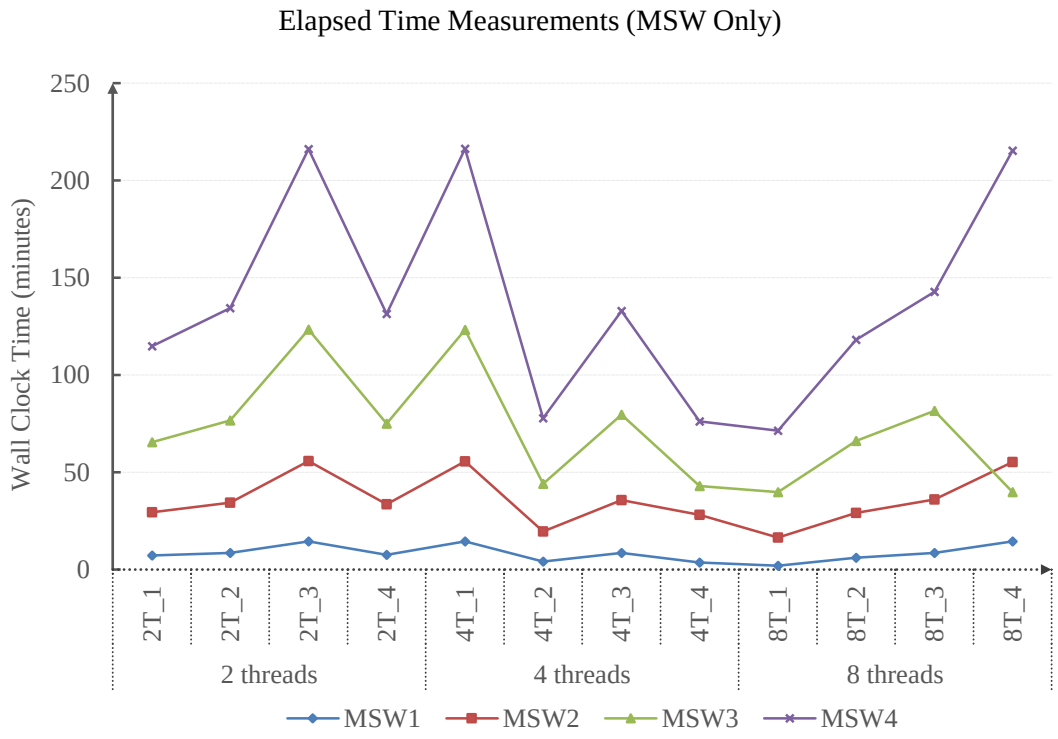
Table 5.14: Summary of prediction error values for Fast Fourier Transform (FFT_6)

	Only ML (AS = FFT_1)	Only ML (AS = FFT_1 & FFT_2)	Only ML (AS = FFT_1 to FFT_3)	Only ML (AS = FFT_1 to FFT_4)	MCM + ML
Average error	25.21921	2.589647	25.91624	26.33398	7.46667
Std. Deviation	11.16867	14.79908	8.819354	8.161058	10.31759

Table 5.15: Transitional probability variation for actual application size data (FFT_6) for Fast Fourier Transform

Mapping Strategies		Training Data		Actual AS Data	
		p	*q*	*p*	*q*
2 threads	2T_1	0.001	0.996667	0.013628	0.957735
	2T_2	0.001	0.996667	0.001	0.994361
	2T_3	0.001	0.996667	0.001	0.998512
	2T_4	0.001	0.996667	0.001	0.98092
4 threads	4T_1	0.001	0.996667	0.001	0.994361
	4T_2	0.99	0.666667	0.863151	0.632334
	4T_3	0.8	0.75	0.675436	0.729553
	4T_4	0.6	0.8	0.67248	0.76517
8 threads	8T_1	0.3	0.966667	0.275	0.944596
	8T_2	0.001	0.996667	0.001	0.998512
	8T_3	0.001	0.996667	0.007193	0.999
	8T_4	0.001	0.996667	0.001	0.994361

5.2.8 Elapsed Time Measurements



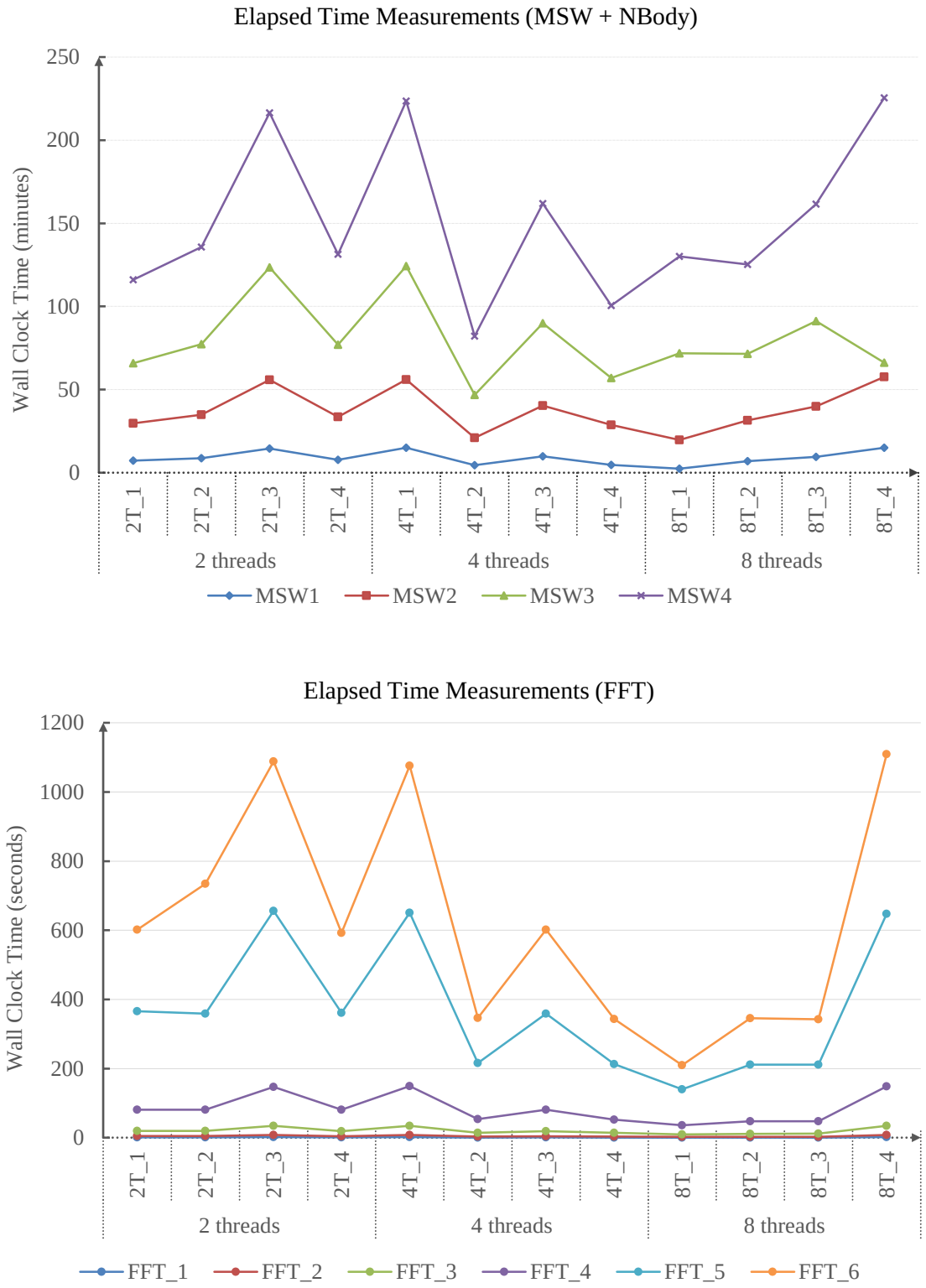


Figure 5.14: Wall clock time versus mapping strategies with core and application scaling for the different experiments

The actual wall clock time with the different application sizes and the MS are plotted in Figure 5.14. These results are from the Linux command time measurements. The first two graphs are for 20,000 iterations of MSW, which translate to around 9.67E11 instructions.

5.2.9 Discussion on Selection of Mapping Strategy

This section covers the primary criteria to select the best possible thread-to-core binding strategy for the given parallel application (MSW). The primary component of this selection is the lowest response time. Another crucial factor is the resources (core and cache count) used for achieving the speed-up. For example, if the two MSs have comparable response times, then it is advisable to select the one with the lowest thread count. This releases the additional cores for use by another application. However, if the number of threads remain the same, the selection process is driven by the least number of caches.

The relationship on CPI and application response time is explained earlier in Section 1.1.1. In the current set of applications, the number of threads is assumed to be constant. This discussion begins with the first application mix with only MSW. Following cases are examined to identify the best and worst results, and their practical implications:

a) Lowest response time - For the constant thread count, MS with highest steady state CPI has the lowest (application) wall clock time. This is to factor in the variation in frequency through the application run. Hence, the MS for each thread count with the highest CPI are as follows:
    i) 2 threads: 2T_1 (different L2-cache with different socket)
    ii) 4 threads:

- 4T_2 (1 thread per L2-cache)
- 4T_4 (2 threads per L2-cache two on each socket)

iii) 8 threads: 8T_1 (1 thread per core)

b) Overall selected - The criteria for MS selection is mentioned is based on the lowest number of cores and caches used to achieve the lowest application response time. In accordance to it, 4T_4 (2 threads per L2-cache two on each socket) is the overall selected MS. The selected MS 4T_4 has similar response time as 4T_2 and 8T_1. However, 4T_2 and 8T_1 are essentially monopolizing all the 4 L2-caches in the machine and hence discarded.

However, as recommended by the VTune profiler, high CPI (over 2) means that 8T_1 is most likely to show improvement with optimizations. In comparison, the other two (4T_2 and 4T_4) with CPI around 1, will potentially plateau performance-wise in spite of the additional modifications. However, exploring these optimizations is beyond the scope of this work and left as a future work.

c) Overall rejected - Best (lowest) predicted CPI is for 2, 4 and 8 threads per core (Range of 1.00001 to 1.0031). However, these MSs have the worst response time. These MSs demonstrate the performance penalty for running multiple threads per core on a coarse-grained multithreaded machine.

The next set of discussion is focused on the second benchmark mix evaluated using the framework. This mix consists of MSW with NBody-Solver running in the background. The key points are highlighted below:

a) Lowest response time - As observed in the earlier results, the MS with highest steady state CPI has the lowest (application) wall clock time. The MS with the lowest response time are summarized in the table below:

Table 5.16: Comparison of response time for best MS with thread count variation

Thread Count	Mapping Strategy	MSW only (in minutes)	MSW + NBody Solver (in minutes)
2 threads	2T_1	114.9	116.1
4 threads	4T_2	77.85	82.083
	4T_4	76.15	100.62
8 threads	8T_1	71.43	130.033

- As observed in the table above, the best 8 thread MS suffers a significant performance loss. Comparing third and fourth columns, the current response time is double that of the previous case. This is because one of the cores is shared between the two applications. This causes the thread with the shared core to stall the others leading to significant performance loss.
- The MS for 4 threads shows a 5% performance loss due to the shared cache. However, the loss is so low, because both the applications are running on independent cores.
- The 2 thread MS do not show any effects because all the resources (cores, cache and network) are independent for the two interfering applications.

b) The worst ranked MS remain exactly same as before showing the performance loss due to higher number of threads throttling the pipeline.

The next item under discussion is about MS selection for the FFT workload. The comments on CPI and response time relationship hold true as mentioned earlier in the section and hence, they are not repeated here for brevity. The key comments related to FFT performance are listed below:

- Best response time is with 8 threads with 1 thread per core (8T_1). However, this MS completely monopolizes the machine and hence, it is advisable to explore some other thread-to-core choices.
- The second best elapsed time of 340 seconds is observed with these four MSs: 4T_2, 4T_4, 8T_2 and 8T_3. This value is approximately 1.63 times higher than the overall lowest value (of 8T_1). The resources used are calculated using information in Table 5.8 and Table 5.9. This is summarized below in Table 5.17. The results clearly show that by using just 2 L2-cache and 1 core, the best MS in terms of resources (both core and cache count) is 8T_3 (8threads with 4threads per core). Thus, the best MS in terms of ooverall resources is 8T_3 (8threads with 4threads per core).

Table 5.17: Resource reserved for mapping strategies with the second best elapsed time

Mapping Strategy	Number of Cores	Number of L2-cache
4T_2	4	4
4T_4	4	2
8T_2	4	2
8T_3	2	1

- Increasing the number of threads improves performance. This is demonstrated by the best performing MS with 8T_1. Secondly, all the 4 thread MSs perform much better than the 2 thread ones. Similarly, other than the exception of 8T_4, rest of the 8 threads MSs perform almost as well as 4 threads MSs. This is true in spite of the coarse-grained threading restriction.
- Access to more memory per thread does not necessarily improve performance. This behavior can be shown by the similar elapsed times in 2 thread MSs on separate (2T_1) and same cache (2T_4). This behavior is also demonstrated in comparing 8T_2 and 8T_3. This behavior points to the compute-intensive nature of the application.

5.3 Characterization of Markov Chain Model

This section elaborates on the MCM prediction analysis based on the change in basic input values i.e. the number of states and the probability values. Prediction trends with the change in thread suspension activities and cores/threads are presented in section 5.3.1. The second section is explores the interrelationship of the two transition probabilities with respect to the application behavior. The analysis in the first section is independent of the hardware event counters. However, its effect on the performance is investigated in the second part of this section.

Although the primary focus for the current research is on how an application behaves with the platform, rather than architecture exploration. However, an agreement between the predicted performance characteristics and actual trends brings forth confidence in the designed model. In addition, the strategy of multiple starting points in the pre-processing block (Section 4.3.1) is developed from studying these plot characteristics.

5.3.1 General Probability and Configuration Characteristics

In the current section, the predicted CPI from the MCM solver is plotted with the varying inputs to correlate real and observed performance. The inputs to the model, as explained in the previous sections, are:

a) Configuration variations (number of L2-cache and threads per L2-cache), and
b) Thread transitional probability pair (probability of active-to-suspend - p, and suspend-remaining-suspended - q) - varying from 0 to 1.

The MCM (without empirical measurements) considers infinite cache and bandwidth and, hence, the characterization study does not include threads per L2-cache. The effect of multiple threads accessing a common cache is incorporated in the model design as the

thread transitional probability pair values. This means that different applications running on the same machine should (typically) have very different probability values, for the same number of MCM states and probability expressions. Thus, in this section 5.3, the predicted CPI is independent of the performance gain due to cache sharing. In Figures 5.15 and 5.16, four different thread counts - 2, 4, 8 and 16, are taken under consideration.

The different architecture configurations under investigated in this section are listed below:

a) 2 L2-cache with 1 thread per core

b) 2 L2-cache with 2 threads per core

c) 2 L2-cache with 4 threads per core

d) 4 L2-cache with 1 thread per core

e) 4 L2-cache with 2 threads per core

f) 4 L2-cache with 4 threads per core

g) 8 L2-cache with 1 thread per core

h) 8 L2-cache with 2 threads per core

The configuration 8 L2-cache with 4 threads per core is intentionally omitted here, because it leads to very high number of states (around 400,000). And, state optimization is beyond the scope of the current work. This work is focused on laying the framework and validation of the MCM based performance model.

For each configuration, predicted CPI is plotted with a pair of p and q value. The input probability values are in the following range:

a) 0.001 to 0.009, with intervals of 0.002 (e.g., 0.001, 0.003 and so forth)

b) 0.01 to 0.09, with intervals of 0.02

c) 0.1 to 0.9, with intervals of 0.2

d) 0.91 to 0.99, with intervals of 0.02

e) 0.991 to 0.999, with intervals of 0.002

The two boundary values are chosen as 0.001 and 0.999. They are synonymous with the lowest and highest possible probability values. For each value of *p* or *q*, there are total 25 individual points as shown above. Pairing each *p* value with a *q* value, there are 625 (= 25×25) data points for any given configuration.

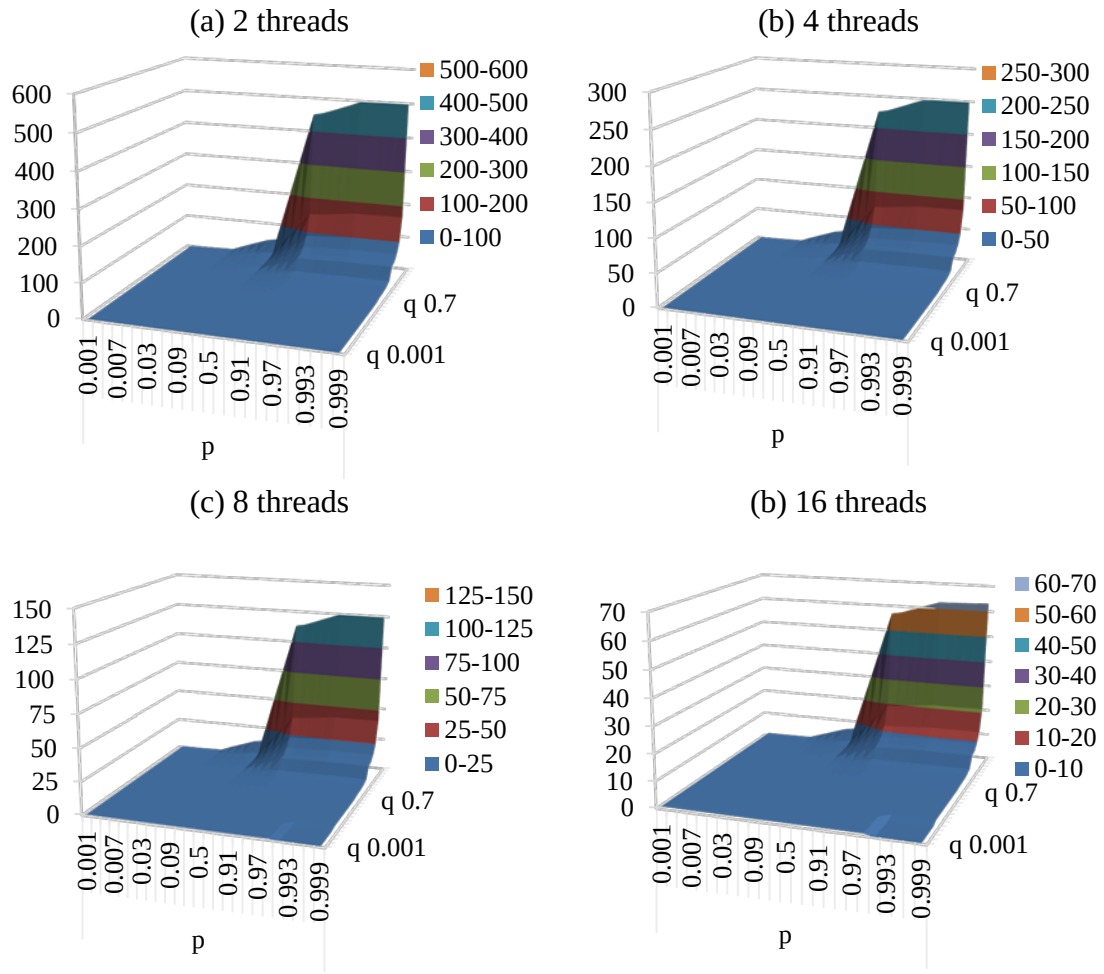


Figure 5.15: Characterization of Markov Chain Model with varying thread count and thread transitional probability pair

Figure 5.15 is a three-dimensional plot showing predicted CPI (output) variations from the MCM. This is plotted with respect to the 625 (input) data points for a given thread

count. These graphs demonstrate the global minima, global maxima and general trends on variations. A more detailed behavior is shown by the plots in Figure 5.16 with a constant value of *q*. The vertical axis represents the predicted CPI in both set of figures (5.15 and 5.16). Also, the primary horizontal axis represents the *p* variations in both these figures. For the characterization curves in Figure 5.15, the depth axis represents the change in the *q* terms.

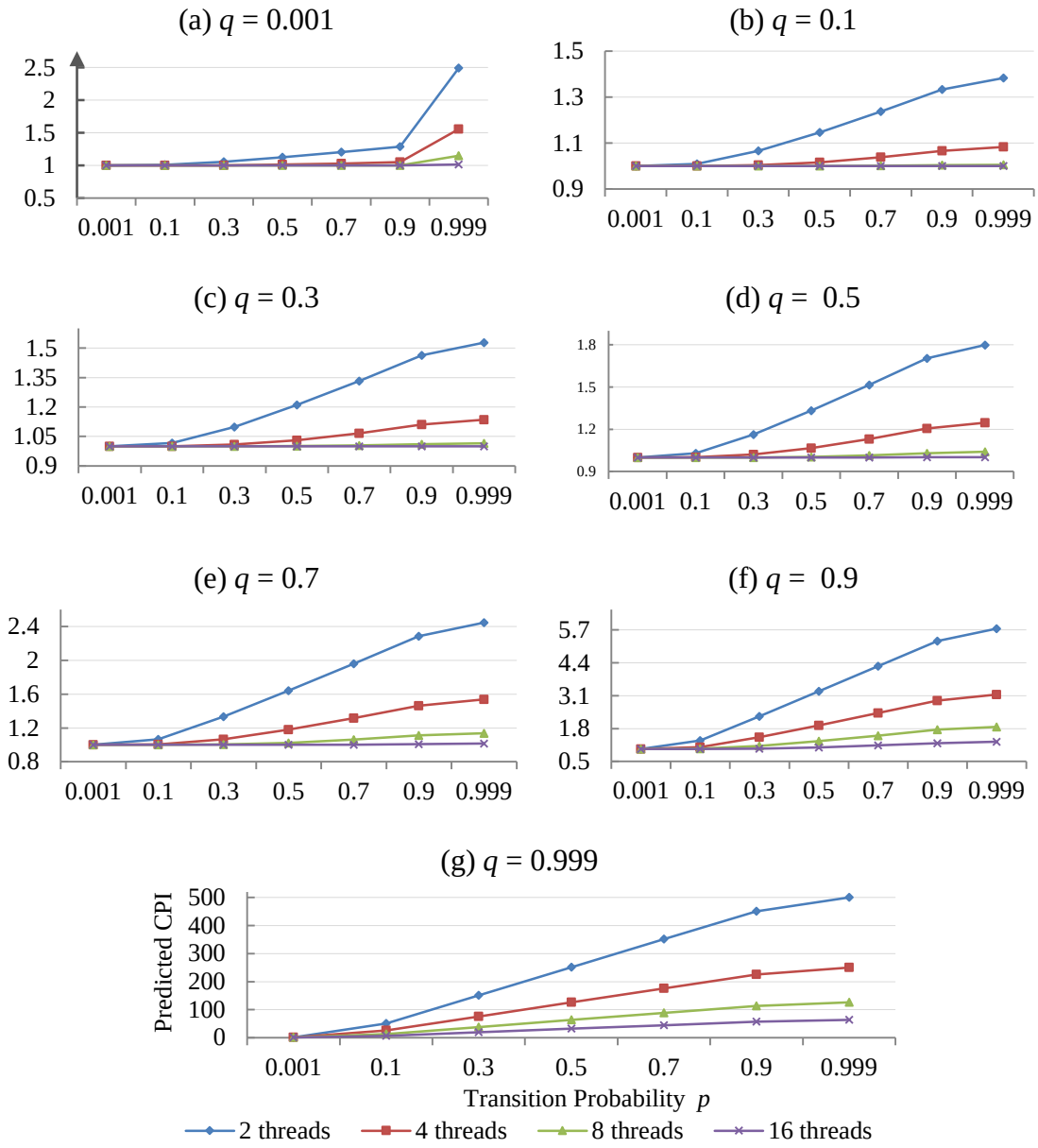


Figure 5.16: Predicted CPI with varying thread suspension probability

Table 5.18: Maximum & minimum predicted CPI variations

Total number of threads	Number of L2 & threads per core	Number of states in MCM	Maximum Predicted CPI	Minimum Predicted CPI
2	2 L2 with 1 thread(s)/core	4	500.2	1.0101±1%
4	2 L2 with 2 thread(s)/core	9	250.526	1.0101±1%
	4 L2 with 1 thread(s)/core	16	250.526	1.0101±1%
8	2 L2 with 4 thread(s)/core	25	125.719	1.0101±1%
	4 L2 with 2 thread(s)/core	81	125.716	1.0101±1%
	8 L2 with 1 thread(s)/core	256	125.425	1.0101±1%
16	4 L2 with 4 thread(s)/core	625	63.0732	1.0101±1%
	8 L2 with 2 thread(s)/core	6531	63.08	1.0101±1%

A summary of results from Figure 5.14 and 5.15 is presented in Table 5.18. The columns represent the maximum and minimum predicted CPI results with the various configurations. The characterization curves show that for all the given thread counts, maximum predicted CPI is always for p = 0.990 to 0.999, and q = 0.999. However, the minimum values are over a bigger range of data points. For example, the transition probability range for the minimum predicted for 2 L2 with 1 thread per core is as follows:

a) p = 0.001 and q = 0.001 to 0.995

b) p = 0.003 and q = 0.001 to 0.970

c) p = 0.005 and q = 0.001 to 0.970

d) p = 0.007 and q = 0.001 to 0.950

e) p = 0.009 and q = 0.001 to 0.930

For all the other configurations, a similar range of probability values exists. However, they are not reproduced here for brevity.

On a fundamental level, high CPI means that the core takes more number of clocks for retirement of a single instruction. This implies poor application performance. The comments to correlate the predicted performance from MCM with the actual scenario are henceforth discussed. The main points on model predictions (without empirical measurements) are listed below:

a) Worst-case performance for higher thread suspension probabilities: As shown in Figure 5.15g, the MCM predicts that the highest CPI is always with (p = 0.990 to 0.999 & q = 0.999). The vice-versa is also true as shown in Figure 5.15a. For very low values of thread transition probability pair (p and q), the predicted CPI is consistently the lowest possible value in all the configurations.

   In the real system, higher thread transition probability pair refers to a large application running on a relatively smaller capacity processor. In such situations, there is an increase in the number of stall-causing instructions, and higher average stall cycles for the long latency operations. This, in turn, brings up the average cycles for an instruction retire, and consequently, causes loss of performance. The reverse situation points to a machine with very low workload. In such a case, threads get suspended less frequently, and the ones which are suspended revert back to active state in the immediate next cycle. In such situations, pipeline stalls are very low and performance is the highest.

b) Maximum performance when either suspension probability is minimum: This case is a modification of the first comment. As shown in Figure 5.14, CPI remains the lowest, when either p or q term is minimum.

Considering the first of the two possible cases, when p is lowest and q is highest. In the real system, this means that the threads never get suspended and hence, there is no performance loss. In the converse case with highest p and lowest q, threads tend to get suspended very frequently. However, they revert back to active state in the very next cycle.

On comparing these two (hypothetical) cases, performance loss is slightly more in the second case. Since in the latter situation, once the thread gets stalled, the machine spends a single cycle to recover back to the active state. This increases the average number of cycles required for an instruction retirement. In terms of the MCM, it is observed that the model correctly predicts this scenario. Performance is better in lowest p (and highest q), rather than vice-versa.

c) Performance improvement with increase in the parallel resources: As shown in Table 5.18 in column 4 and Figure 5.15, CPI increases with the decrease in the number of threads. Secondly, CPI consistently remains lower even with high values of the transition probabilities with the rise in thread count, as shown in Figure 5.15g. For example, the lowest of the “highest” predicted CPI is observed with 16 threads.

d) Threads with higher suspension probability will benefit more with higher thread-level parallelism: Comparing CPI for 16 and 2 threads for two probability cases (a) q = 0.3 and (b) q = 0.9, as shown in figures 5.15(c) and 5.15(f). For constant p = 0.9, it is observed that the ratio of CPI for 2 threads with respect to 16 threads is 4 times for case (b), with the higher value of q = 0.9. On the other hand, this ratio of CPI is almost constant in case (a), with the lower value q = 0.3. Similar trends can

be observed with variation of p, with constant q. This example refers enumerates the decrease in CPI (or performance gain) with higher thread counts, in cases with higher thread suspension probability.

In the real system, applications suffering from thread stalls need to resolve the resource contention problem to gain back performance. In such cases, increasing the thread count provides access to more number of cores, cache and bandwidth. However, applications with lesser stalls may have reached a performance plateau and hence may not benefit from increasing the thread count.

Although some of the above comments may appear obvious. However, the designed model becomes more plausible as the predicted performance trends point towards realistic situations.

A limitation of this model is that the lowest predicted CPI always remains unity (Table 5.18 column 5). The predicted CPI is computed as $1/(1 - p_s)$ at steady state (Section 4.2.1). The minimum value of this probability term (p_s) is zero, since the negative probability terms are omitted here for all practical purposes. Thus the minimum value of the overall throughput expression comes out as unity. Since, the current work is for high performance computing application optimization; at this point this limitation is not of much concern.

5.3.2 Thread Transition Probability Pair Characteristics

In the previous section, predicted performance trends are explored with varying states and transition probabilities. However, it is important to explore the factors that influence the change of these probability values. Hence, the study of the thread transition probability pair relationship is presented in this section.

The *p* value is dependent upon instructions causing a thread to get suspended. The average penalty cycles associated with each suspension occurrence defines the *q* value. Their characteristics studies are indicative of the application-architecture interaction, which is crucial for application optimization. A fixed probability range to specific algorithm or architecture type seems to be an over-simplification at this stage. However, an overview of conditions for its increase or decrease is covered in this section.

First, comments about probability of active-to-suspend are presented. Then, probability of suspend-remaining-suspended analysis is shown.

The *p* value is calculated from the single and multi-threaded implementations, as shown in equations (5) and (6). There are three primary comments about this:

a) With the increase in the thread disruption activities such as cache misses, computation, among others, threads tend to get suspended more frequently. Hence, there is an increase in the *p* value. For example, in the same application running on the same architecture, typically, the *p* value should increase with increasing application data size. Another example can be a compute-intensive application tending to benefit from a more powerful ALU, then a string matching application. In such a machine, the former case will have a lower *p* value as compared to the later.

b) Multiple applications with different computation pattern running on the same architectures can have equal *p* value. This is true, provided that the rate of thread suspension activities is identical even though the cause of suspension may be very different. In such situations, the change in predicted performance comes from the variation in *q* values. For example, (1) influence of cache misses in a fluid dynamics simulation with a large AS and, (2) bus transactions in a correlation computation with a large number of cross computations with smaller data set, can be comparable. Consider the following applications, with instruction retired for both cases is *n*:

TABLE 5.19: Example - 1 for variation of *p*

Events	Application # 1 (same architecture)			Application # 2 (same architecture)		
	Single-threaded	Multi-threaded	PEC	Single-threaded	Multi-threaded	PEC
Branch misses	x_1	x_2	$x_1 - x_2$	y_1	y_2	$y_1 - y_2$
Division	y_1	y_2	$y_1 - y_2$	0	0	0
Cache Misses	0	0	0	x_1	x_2	$x_1 - x_2$

In both the cases, the *p* value is calculated as $\frac{(x_1 - x_2) + (y_1 - y_2)}{n}$.

c) Similarly, applications with different computation patterns running on very different machines can have equal *p* value. This is true with the same condition mentioned above that the ratio terms are identical in both the cases. An example can be a small embedded application running on an 8-bit processor and a scientific computing application on a 64-bit processor. This is illustrated with the following example. In both the cases, the *p* value is 0.4.

Table 5.20: Example - 2 for variation of *p*

Events	Application # 1, Architecture # 1			Application # 2, Architecture # 2		
	Single-threaded	Multi-threaded	PEC	Single-threaded	Multi-threaded	PEC
Branch misses	20	2	18	210	40	170
Division	55	33	22	0	0	0
Cache Misses	0	0	0	55	80	50
Instructions retired		100	Total = 40		500	Total = 200

The next set of discussions is on the probability of suspended thread remaining suspended. The *q* value is dependent upon two factors - application behavior, and event latency cost, as shown in equation (8). However, the more dominant influence comes from the latency or the per-event cost, as seen in the subsequent examples. The main observations are listed below:

d) For a single event, as the average stall cycle increases, *q* value increases as shown in Figure 5.17. It follows a linear asymptotic behavior reaching towards unity for higher values, as obtained by plotting the equation (7). Realistically, any pipeline with smaller buffer size will cost more in terms of the higher number of stall cycle. Performance-wise this means that the suspended threads with *higher* latency costs should have a *higher* chance of remaining suspended, and vice-versa.

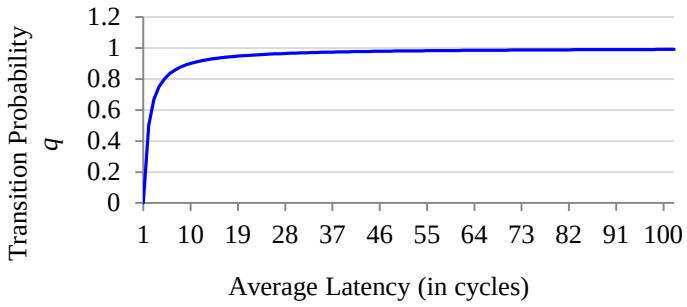


Figure 5.17: Probability of suspend-remaining-suspended variation with average latency (single event)

e) Due to linear asymptotic behavior shown above in Figure 5.16, the *q* value stabilizes significantly with higher values of average latency cycles. An important implication is that for the same application, running on the same architecture, typically the *q* value should maintain similar range with increasing AS.

f) This comment is similar to the previous observation *(d)*, but in a more expanded context. Consider the same parallel application (with exactly same thread count) running on multiple architectures. This means that the cost of same stall causing event will be different in each case. Even with the same number of stalls, the idea is to determine which of the machines will have a higher likelihood of suspended threads remaining suspended.

Mathematically, this is demonstrated by the four cases (columns 3 to 6) in the following example in Table 5.21. Here, the number of occurrences of each event (column 2) is exactly equal for all the four cases. The value of *q* is the highest for the last case (0.999), where the cost of cycle is the highest. However, as shown in the comment (e) above, in spite of the increase in average latency cycles, the value of *q* stabilizes at an early knee point.

Table 5.21: Example - 1 for variation of *q*

Events	Number of occurrences (f_i)	Cycles per occurrence (m_i) – same application			
		Architecture # 1	Architecture # 2	Architecture # 3	Architecture # 4
A	3	2	20	2	20
B	2	4	4	40	40
M		8.8	246.4	642.4	880
q		0.886	0.996	0.998	0.999

For example, for the same number of L1 and L2 cache-misses for a given application, threads tend to get stalled with much higher probability in machines with lowest capacity of both these caches.

g) The next case explores the opposite of the above comment. This refers to applications with different execution patterns running on very different machines. In spite of their different computation and memory access patterns, they can have equal *q* value. This can happen only when the average overall latency cost becomes exactly identical. The following example demonstrates this point:

Table 5.22: Example - 2 for variation of *q*

Events	Architecture # 1, Application # 1		Architecture # 2, Application # 2	
	Cycles per occurrence	Total cycles	Cycles per occurrence	Total cycles
Branch misses	*u*	*x*	*v*	*y*
Division	*v*	*y*	*w*	"0"
Cache Misses	*w*	"0"	*u*	*x*

A realistic example of such a case can be a compute-intensive application running on a platform with a fast ALU should have similar *q* value as a memory-intensive application on architecture with large memory and bandwidth.

Summarizing the ideas from the above discussion on MCM characterization, some basic predicted and actual performance trend agreements have emerged. First, the model successfully predicts the worst-case performance when both thread suspension probabilities are the maximum. There is slight gain in performance, if just one of them is the maximum. Secondly, the model demonstrates performance improvement with the increase in the number of threads. This is in accordance with the Amdahl's Law, since the part of application which can be parallelized remains same all throughout. Third, the

model points to cases with different applications with similar performance in spite of their dissimilar behavior. This idea is extended further in the Multiple Starting Points in Section 4.3.1. In this block, the MCM maps the performance to a certain band, irrespective of the cause.

5.4 Summary

Chapter 5 focusses on the benchmark, experimental set-up and framework validation results. It begins with the application-specific timing model for MSW. This is for representing the communication and computation times for data partitioning strategies. In the next section, the hybrid model is validated with the results for two different applications - MSW & FFT. The first set has two types of experiments. The first one being a singular application (MSW) running on the machine. The second type of experiments again predicts the performance of the same parallel program. However, in this experiment, another application (N-Body Solver) is running in the background. The second set is with FFT prediction. The third (last) section in this chapter covers the characterization of MCM prediction with respect to the change in the number of states and probability values.

CHAPTER 6: CONCLUSION AND FUTURE WORK

This chapter summarizes the main ideas described in this dissertation. Top-level comments on the problem statement, model metric, proposed solution and model design are presented in the first section. The second section proposes ways of extending the designed framework to improve the scope of the solution.

6.1 Conclusion

The objective of this dissertation is to design a framework for improving the productivity of the parallel programmers. The fast, accurate and portable solution should enable the application development community to explore large number of MS within reasonable machine hours, without detailed information about the program or platform.

The current performance analysis framework is a fast and accurate tool for identifying steady state CPI characteristics of a given parallel application for various thread-to-core mapping choices. The architecture under study is a hierarchical, shared memory, multiple issue multicore processor, with coarse-grained multithreading.

The four primary contributions of the dissertation are summarized in the following paragraphs. First is identifying an approach for the MS exploration, which is fast, accurate and portable. This also includes laying down the model framework in the form of hybrid combination of MCM and MT for steady state CPI prediction for the parallel application. It consists of designing the MCM states, translating the probability expressions into measurements, and extrapolating transitional probability by MT model.

Second contribution is developing an application-specific timing model to select the data partitioning strategy. The results demonstrate that the 1-D partitioning works better than 2-D due to increased data re-usage.

The third contribution is thread-to-core model validation on Intel Xeon Clovertown (X5365) using electromagnetics application (core scaling) against VTune measurements. As demonstrated in the results section, the model effectively predicts CPI with an average error of 0.168% with standard deviation of 3.866%. In comparison, a purely SML based model has an average error of 7.926% with standard deviation of 22.216%. The total run time for the model is of the order of minutes, whereas the actual application execution time is in terms of days. This is including the overhead of running the application with the performance analyzer.

The last contribution is use of the framework to explore the MS for the given electromagnetics benchmark. The most interesting result from this step demonstrates that maximizing core and cache count is not required to obtain the lowest response time. The best selected MS on an 8-core machine is with 4 threads (2 threads per L2-cache with 2 on each socket). It is also observed that the result remains true even with MSW has another application (N-Body Solver) running in the background.

In conclusion, this dissertation develops a framework for fast, accurate and portable CPI prediction for data-intensive iterative parallel applications running on multicores. The steady state value translates to information about the actual response time for the various MS. This CPI data, in turn, can be used for effectively exploring the design space. Thus, the performance analyst can make informed selections about the various

thread-to-core schemes, without developing an ad-hoc application-specific model or a detailed understanding about how the program interacts with the platform.

6.2 Future Work

This section is divided into two main parts. The first part focusses on how to use this framework for the I/O (input/output) bound problems. The second part discusses the other possible improvements.

6.2.1 I/O Bound Problems

In the current scenario, it is assumed that all the data resides in the main memory. Hence, it is a CPU- or memory-bound, rather than an I/O bound problem. The subsequent paragraphs provide ideas for modifying the MCM to extend the scope of this dissertation for I/O bound problems.

These I/O bound problems with extremely large data sets spend several thousands of clock cycles for reading data from the external devices to the main memory. On the core side, this is overlapped by small bursts of activity in terms of CPU execution, followed by very long wait periods. These extended periods represent the processes waiting for completion of the I/O reads and writes. Thus, it is very different from thread suspension, due to unavailability of pipeline resources or cache misses. Hence, with the enormous difference in the (clock) cycle cost for I/O accesses, it is a much accurate representation to treat the I/O behavior separately from the processor and memory.

In the current context of the MCM, the I/O could be treated as two different states - idle (inactive) and busy (reading disk or writing to main memory). The thread behavior can remain similar to the states shown in Figure 4.3. Thus, the overall system can be represented by the following four states:

- I/O idle, thread active
- I/O idle, thread suspended
- I/O busy, thread active
- I/O busy, thread suspended

In view of the discussion above, it is very clear that most of the times the system will remain in the last two states with busy I/O. However, the first two are included here for sake of completeness.

The next important aspect of MCM design is the transitional probability expressions for the I/O accesses. The probability of system transitioning from idle to busy is determined by two main factors: (1) ratio of size of main memory to external disk, and (2) total size of data set. For example, a system with a smaller main memory will move to the busy state with a much higher probability, in comparison with another one with a larger memory for the same application. The second probability term of system remaining in the busy state can be decided based on the cost cycles and frequency of the disk accesses. This concludes a brief overview of extending the framework for I/O bound problems.

6.2.2 Other Improvements

As part of the future work, the following other enhancements can improve the scope of the framework:

a) Currently the framework usage gets restricted by the availability of VTune measurements. A significant improvement will be removing this dependence by developing a chain of memory and execution models to assign the thread transitional probability pair values.

b) Another logical extension to the framework is to dynamically vary the MS during execution. Although past work is done for dynamically varying the thread count, but it would be interesting to examine the behavior even with same number of threads, but switching the thread binding based on requirement.

c) It is important to identify performance improvement using a particular memory or computation optimization. Hence, another key usage enhancement will be covering the effect of a particular group of events (e.g. misses or divisions) exclusively.

REFERENCES

[1] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” In Proceedings of AFIPS, pp. 483 - 485, 1967.

[2] D. Eager, J. Zahorjan, and E. Lazowska, “Speedup versus Efficiency in Parallel Systems,” IEEE Transactions on Computers, 38(3):408–423, 1989.

[3] K. Sevcik, “Characterizations of Parallelism in Applications and Their Use in Scheduling,” In Proceedings of 10th ACM Conference on Measurement and Modeling of Computer Systems, pp. 171–180, Berkeley, USA, 1989.

[4] A. B. Downey, “A Parallel Workload Model and its Implications for Processor Allocation,” In Proceedings of 6th IEEE International Symposium on High Performance Distributed Computing, pages 112-123, Portland, USA, 1997.

[5] M. D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” Computer, vol. 41, no. 7, pp. 33 - 38, 2008.

[6] X. H. Sun, Y. Chen, “Reevaluating Amdahl’s law in the multicore era”, Journal of Parallel and Distributed Computing, 70(2): 183–188, 2010.

[7] S. Eyerman and L. Eeckhout, “Modeling Critical Sections in Amdahl’s Law and its Implications for Multicore Design,” In Proceedings of ACM/IEEE International Symposium on Computer Architecture - 37, 2010.

[8] A. Cassidy, K. Yu, H. Zhou, A. Andreou, “A High-Level Analytical Model for Application Specific CMP Design Exploration,” In Proceedings of IEEE Conference on Design, Automation & Test in Europe (DATE), page 1-6, 2011.

[9] N. Gunther, S. Subramanyam, S. Parvu, “A Methodology for Optimizing Multithreaded System Scalability on Multi-Cores,” http://arxiv.org/abs/1105.4301 2011

[10] L. Yavits, A. Morad, R. Ginosar, “The Effect of Communication and Synchronization on Amdahl’s Law in Multicore Systems,” Elsevier Journal of Parallel Computing, 40(1):1-16, 2014.

[11] B. M. Tudor and Y. M. Teo, “A Practical Approach for Performance Analysis of Shared-Memory Programs,” In Proceedings of IEEE International Parallel & Distributed Processing Symposium - 27 - 27, 2011.

[12] D. B. Noonburg and J. P. Shen, “Theoretical Modeling of Superscalar Processor Performance,” In Proceedings of International Symposium on Microarchitecture - 27, pp. 52 - 62, 1994.

[13] S. S. Nemawarkar, R. Govindarajan, G. R. Gao, V. K. Agarwal, “Analysis Of Multithreaded Multiprocessors with Distributed Shared Memory,” In Proceedings of IEEE Symposium on Parallel and Distributed Processing, pp. 114-121, 1993.

[14] S. S. Nemawarkar and G. R. Gao, “Measurement and Modeling Of EARTH-MANNA Multithreaded Architecture,” In Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 109–114, San Jose, 1996.

[15] A. Jacquet, V. Janot, C. Leung, G. R. Gao, and R. Govindarajan, T. L. Sterling, “An Executable Analytical Performance Evaluation Approach for Early Performance Prediction,” in Proceedings of IEEE International Parallel and Distributed Processing Symposium, 2003.

[16] R. H. Saavedra-Barrera, and D. E. Culler, “An Analytical Solution for a Markov Chain Modeling Multithreaded Execution,” TR UCB/CSD-91-623, EECS Dept., University of California, Berkeley, 1991.

[17] V. Bhaskar, “A Closed Queuing Network Model with Multiple Servers for Multi-Threaded Architecture,” In Computers and Electrical Engineering, 31(14):3078–3089, 2008.

[18] A. Navarro, R. Asenjo, S. Tabik and C. Cascaval, “Analytical Modeling of Pipeline Parallelism,” in Proceedings of International Conference on Parallel Architectures and Compilation Techniques – 18, pp. 281 - 290, 2009.

[19] R. Zilan, J. Verdú, J. García, M. Nemirovsky, R. A. Milito and M. Valero, “An Abstraction Methodology for the Evaluation of Multi-core Multi-threaded Architectures,” in Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 478 – 481, 2011.

[20] H. Jung, M. Ju and H. Che, “A Theoretical Framework for Design Space Exploration of Manycore Processors,” in Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems - 19, pp. 117 - 125, 2011.

[21] M. Ju, H. Jung and H. Che, “A Performance Analysis Methodology for Multi-core, Multithreaded Processors,” IEEE Transactions on Computers, 2011.

[22] X. E. Chen and T. M. Aamodt, “A First-Order Fine-Grained Multithreaded Throughput Model,” in Proc. of Intl. Symp. on High-Performance Computer Architecture – 15, pp. 329 - 340, 2009.

[23] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, A.J. Wasserman, and M. Gittings, “Predictive Performance And Scalability Modeling Of A Large-Scale Application,” In Proceedings of IEEE/ACM Supercomputing ’01, November 2001.

[24] L. Carrington, A. Snavely, X.Gao, and N. Wolter, “A Performance Prediction Framework for Scientific Applications,” In Proceedings of International Conference on Computational Science Workshop on Performance Modeling and Analysis (PMA03), pp. 926–935, June 2003.

[25] L. Carrington, N. Wolter, A. Snavely and C.B. Lee, “Applying an Automatic Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications,” In Department of Defense Users Group Conference, June 2004.

[26] G. Marin and J. Mellor-Crummey, “Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models,” In Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics), pages 2–13, June 2004.

[27] I. Sharapov, R. Kroeger, G. Delamarter, R. Cheveresan and M. Ramsay, “A Case Study in Top-Down Performance Estimation for a Large-Scale Parallel Application,” In Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 81-89, 2006.

[28] L. T. Yang, X. Ma and F. Mueller, “Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution,” In Proceedings of IEEE/ACM Supercomputing , page 40, 2005.

[29] V. Adve and R. Sakellariou, “Application Representations for Multiparadigm Performance Modeling of Large-Scale Parallel Scientific Codes,” International Journal of High Performance Computing Applications, 14(4): 304-316, 2000.

[30] A. van Gemund, “Symbolic performance modeling of parallel systems,” IEEE Transactions on Parallel and Distributed Systems, 14(2): 154-165, 2003.

[31] E. Ipek, B. R. de Supinski, M. Schulz and S. A. McKee, “An Approach to Performance Prediction for Parallel Applications,” In Proceedings of Euro-Par Parallel Processing, pp. 196 - 205, 2005.

[32] B.C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh and S. A. McKee, “Methods of Inference and Learning for Performance Modeling of Parallel Applications,” In Proceedings of ACM Symposium on the Principles and Practice of Parallel Programming - 12, pp. 249 - 258, 2007.

[33] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, “A Regression-based Approach to Scalability Prediction,” In Proceedings of ACM International Conference on Supercomputing - 22, pp. 368–377, 2008.

[34] Z. Wang, and M. F. P. O'Boyle, “Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach,” ACM Sigplan Notices, 44(4): 75 - 84, 2009.

[35] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos, “Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes,” IEEE Transactions on Parallel Distributed Systems, 19(10):1396–1410, 2008.

[36] M. Curtis-Maury, “Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems,” Ph.D. Thesis, http://vtechworks.lib.vt.edu/handle/10919/26697

[37] K. Singh, M. Curtis-Maury, S. McKee, F. Blagojevic, D. Nikolopoulos, B. de Supinski and M. Schulz, “Comparing Scalability Prediction Strategies on an SMP of CMPs,” in Proc. of Intl. European Conf. on Parallel and Distributed Computing, pages 143 - 155, 2010.

[38] A. Ganapathi, K. Datta, A. Fox and D. Patterson, “A Case for Machine Learning To Optimize Multicore Performance,” In HotPar, 2009.

[39] P. Bose and T. Conte, “Performance Analysis and Its Impact On Design,” Computer, 31(5):41–49, May 1998.

[40] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, J. E. Smith, “The future of simulation: A field of dreams,” Computer, 39(11): 22 - 29, 2006.

[41] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” Technical Report ucb/eecs-2006-183. EECS, University of California, Berkeley, Dec. 2006.

[42] Intel 64 and IA - 32 Architecture Optimization Manual.

[43] O. Wechsler. Inside Intel Core Microarchitecture. White paper.

[44] Intel VTune Amplifier Documentation http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/lin/lin_ug/index.htm

[45] D. Levinthal, “Cycle Accounting Analysis on Intel Core 2 Processors,” White

paper, http://assets.devx.com/goparallel/18027.pdf 2008.

[46] B. Sprunt, “The Basics of Performance-Monitoring Hardware,” IEEE Micro, 22(4):64 - 71, 2002.

[47] J. L. Hennessy and D. A. Patterson, “Computer Architecture: A Quantitative Approach,” Morgan Kauffman, 4th ed., 2007.

[48] A. Grama, A. Gupta, G. Karypis, V. Kumar, “Introduction to Parallel Computing,” Addison Wesley, 2nd ed., 2003.

[49] http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/Chapter11.pdf (MCM reference)

[50] R. J. Quinlan, “Learning with Continuous Classes,” In Proceedings of 5th Australian Joint Conference on Artificial Intelligence, Singapore, pp. 343-348, 1992.

[51] I. H. Witten, E. Frank, L. E. Trigg, M. A. Hall, G. Holmes, S. J. Cunningham, “Weka: Practical machine learning tools and techniques with Java implementations,” Dept. of Computer Science, University of Waikato, http://www.cs.waikato.ac.nz/ml/weka/

[52] Y. Wang, and I. H. Witten, “Inducing Model Trees for Continuous Classes,” In Proceedings of 9th European Conference on Machine Learning, pp. 128-137, 1997.

[53] C. Perlich, and F. Provost, “Tree Induction vs Logistic Regression,” NIPS Workshop, Beyond Classification and Regression, 2002.

[54] N. Landwehr, “Logistic Model Trees,” Ph.D. Dissertation, University of Freiburg, http://www.cs.uni-potsdam.de/ml/landwehr/diploma_thesis.pdf 2003.

[55] Scott Fortmann-Roe, “Understanding the Bias-Variance Tradeoff,” http://scott.fortmann-roe.com/docs/BiasVariance.html

[56] R. Mitra, B. S. Joshi, R. Adams, A. Ravindran, A. Mukherjee, “Modeling and Performance Analysis of Parallel Magneto-static Wave Calculations on Multicore Architecture”, 22nd Annual Supercomputing Conference, November 2009.

[57] R. Mitra, B. S. Joshi, A. Ravindran, R. Adams, A. Mukherjee, J. Byun and K. Datta, “Performance Modeling of Parallel Magneto-static Wave Calculations on Shared Memory Multicore”, IEEE Southeast, March 2010.

[58] R. Mitra, B.S. Joshi, A. Ravindran, A. Mukherjee and R. Adams, “Performance

Modeling of Shared Memory Multiple Issue Multicore Machines," In Proceedings of International Conference on Parallel Processing Workshops - 41, pp. 464 - 473, 2012.

[59] R. Mitra, B. S. Joshi and R. Adams, "Thread Mapping using System-Level Throughput Prediction Model for Shared Memory Multicores", In Proceedings of IEEE International Performance Computing and Communications Conference -2014.

[60] K. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," IEEE Transactions on Antennas and Propagation, vol. 14, no. 3, pp. 302 - 307, 1966.

[61] http://www.cs.usfca.edu/~peter/ipp/index.html

[62] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms", 2nd ed. (MIT Press, Cambridge, MA, 2001)

[63] D. P. Bovet, and M. Cesati, "Understanding the Linux kernel", O'Reilly Media, Inc., 2005

[64] T. Sherwood, E. Perelman and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," In Proceedings of IEEE Parallel Architectures and Compilation Techniques, pp. 3-14, 2001.

APPENDIX A: MARKOV CHAIN SOLVER

Algorithm for the MCM solver is listed below:

a) Transition probability at the beginning of run, p_{ij} at $t = 0$, is given by:

$$p_{ij}(t=0) = \begin{bmatrix} (1-p)^8 & \dots & p^8 \\ \vdots & \ddots & \vdots \\ (1-q)^8 & \dots & q^8 \end{bmatrix} \quad (17)$$

The above equation represents all eight working threads (shown in Figure 4.8.1).

b) Compute $(p_{ij})^t$ when $t \to \infty$. For practical purposes, it is assumed that $t = 1000$.

c) Assuming that all threads are active in the beginning, the probability vector for starting distribution, u, is $[1 \quad 0 \quad \dots \quad 0]$. The state of machine at $t \to \infty$ i

$$P = u \times (p_{ij})^t \quad (18)$$

d) In the probability vector, P, the first element represents the probability of machine at state 0 (all threads active), the next one is for state 1 (any one thread suspended), and so on.

e) The throughput of the machine (Cycles per Instruction, or CPI) is equal to:

$$(1 - P_N) \quad (19)$$

where, N represents state where all threads are suspended. In the current case N = 81.

APPENDIX B: VTUNE - INTEL'S CYCLE-ACCURATE PERFORMANCE ANALYZER

It is important to understand the basics of performance monitoring hardware [44] and, in particular, Intel's detailed cycle-accurate commercial performance analyzer − VTune. It is a well-recognized application and can be used from embedded systems through supercomputers. The main features in VTune analyzer include graph generation, time-based and event-based, thread profiling, performance tuning, among others. The five main categories of performance events are

a) program characterization

b) memory accesses

c) pipeline stalls

d) branch prediction

e) resource utilization.

The two main categories of profiling are time-based sampling and event-based sampling based on the timing of when the application is interrupted by the performance monitoring hardware. The current project uses event-based sampling or EBS. Time-based sampling is unavailable in Linux version of VTune analyzer. This means that instead of interrupting the application at regular time intervals (as in time-based sampling), the performance monitoring hardware interrupts the application after a specific number of performance events has occurred. The samples taken during application execution are instruction address (module, source line and assembly line), OS process and OS thread.

The maximum number of events which can be recorded at a time is four. This is restricted by the number of Performance Monitoring Unit registers.

Sample After value (SAV) is the frequency or the number of events after which the VTune analyzer interrupts the processor to collect a sample during EBS. By default, the calibration option is enabled and the VTune analyzer adjusts the Sample After value automatically. During the first run of the application (called “Activity”), the VTune analyzer records every occurrence of an event. It, then, calculates the total number of events and calibrates the Sample After value. The second time the “Activity” runs, the VTune analyzer collects samples based on the calibrated Sample After value. Typically, the analyzer calculates the CPU's speed and sets the Sample After value so that there will be 1000 samples per second.

Average latency or penalty cycle associated with each event (m_i in equation (8)) is calculated using this Sample After values with the following formula:

$$\frac{SAV\ for\ CPU\_CLK\_UNHALTED.CORE}{SAV\ for\ event-in-interest} \tag{20}$$

The VTune event - CPU_CLK_UNHALTED.CORE counts the number of core cycles while the core is not in a halt state.

APPENDIX C: VTUNE PERFORMANCE EVENTS

This list is leveraged from the main stall events information mentioned in the profiler documentation.

a) Explanation about *p* events is presented below:

i) BR_INST_RETIRED.MISPRED: Retired mispredicted branch instructions.

ii) BUS_BNR_DRV: Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents.

iii) DIV: divide operations executed. This includes integer divides, floating point divides and square-root operations executed.

iv) FP_ASSIST: floating point operations executed that required micro-code assist intervention for denormalized input or underflow output.

v) ITLB_MISS_RETIRED: Retired instructions that missed the ITLB.

vi) L1I_MISSES: all instruction fetches that miss the Instruction Fetch Unit (IFU) or produce memory requests.

vii) LOAD_BLOCK.L1D: Loads blocked by the L1 data cache.

viii) LOAD_BLOCK.OVERLAP_STORE: Loads that partially overlap an earlier store, or 4K aliased with a previous store.

ix) LOAD_BLOCK.UNTIL_RETIRE: Loads blocked until retirement.

x) MEM_LOAD_RETIRED.DTLB_MISS: retired loads that missed the DTLB.

xi) MEM_LOAD_RETIRED.L1D_LINE_MISS: load operations that miss the L1 data cache and send a request to the L2 cache to fetch the missing cache line.

xii) MEM_LOAD_RETIRED.L2_LINE_MISS: load operations that miss the L2 cache and result in bus request to fetch the missing cache line.

b) Following events are under consideration for *q* calculation:

i) BUS_LOCK_CLOCKS: Bus cycles when a LOCK signal is asserted.

ii) CYCLES_DIV_BUSY: number of cycles the divider is busy executing divide or square root operations.

iii) CYCLES_L1I_MEM_STALLED: Cycles during which instruction fetches are stalled.

iv) DELAYED_BYPASS.FP: number of times floating point operations use data immediately after the data was generated by a non-floating point execution unit.

v) DELAYED_BYPASS.LOAD: number of delayed bypass penalty cycles that a load operation incurred.

vi) IDLE_DURING_DIV: number of cycles the divider is busy (with a divide or a square root operation) and no other execution unit or load operation is in progress.

vii) ILD_STALL: Instruction Length Decoder stall cycles due to a length changing prefix

viii) INST_QUEUE.FULL: Cycles during which the instruction queue is full.

ix) L1D_CACHE_LOCK_DURATION: Duration of L1 data cacheable locked operation.

x) L1D_REPL: number of lines brought into the L1 data cache.

xi) L2_LINES_IN.SELF.ANY: number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. SELF represents events initiated by this core only.

xii) MACHINE_NUKES.MEM_ORDER: Execution pipeline restart due to memory ordering conflict or memory disambiguation misprediction.

xiii) MEMORY_DISAMBIGUATION.RESET: cycles during which memory disambiguation misprediction occurs.

xiv) PAGE_WALKS.CYCLES: Duration of page-walks in core cycles.

xv) RAT_STALLS.ANY: number of stall cycles due to:

- Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe.
- Cycles when partial register stalls occurred
- Cycles when flag stalls occurred
- Cycles floating-point unit (FPU) status word stalls occurred

xvi) RESOURCE_STALLS.ANY: number of cycles while resource-related stalls occur:

- Number of instructions in the pipeline waiting for execution or retirement reached the limit the processor can handle.

- Number of load or store instructions in the pipeline waiting for retirement reached the limit the processor can handle.
- There is an instruction in the pipe that can be executed only when all previous stores complete and their data is committed in the caches or memory. For example, SFENCE and MFENCE instructions require this behavior.
- The pipeline recovers from a mispredicted branched that was executed.
- The floating-point unit (FPU) control word is written.

xvii) SB_DRAIN_CYCLES: Cycles while stores are blocked due to store buffer drain.

APPENDIX D: WEKA OUTPUTS

A sample Weka output is shown in this section. The input CSV file has the following column labels:

a) AS (Application_size)

b) Iteration

c) Number of L2-cache (Num_L2)

d) Number of threads per L2-cache (Num_threads_per_L2)

e) MS (Mapping_Strategy)

f) Number of MCM States (Num_MC_States)

g) Value of *p* (p_value)

h) Value of *q* (q_value)

The summary of the linear models are shown in the Table A.4.1.

Table A.1: Sample Model Tree created using Weka

Value of *q*	**Number of MCM States**	**Linear Model**
≤ 0.84	≤ 20.5	LM1
≤ 0.84	> 20.5	LM2
> 0.84 ≤ 0.982		LM3
> 0.982		LM4

The different linear models from the third column in the above table are listed below:

LM1:

p_value =

0 * Iteration

+ 0.1009 * Num_L2

+ 0.0698 * Num_threads_per_L2

- 0.0072 * Mapping_Strategy= 2threads_different_L2_same_socket , 8threads_2threads_per_core , 8threads_1threads_per_core , 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.1019 * Mapping_Strategy= 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.002 * Num_MC_States

- 1.6038 * q_value

+ 1.1725

LM2:

p_value =

0 * Iteration

+ 0.0196 * Num_L2

+ 0.0413 * Num_threads_per_L2

- 0.0072 * Mapping_Strategy= 2threads_different_L2_same_socket , 8threads_2threads_per_core , 8threads_1threads_per_core , 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.1019 * Mapping_Strategy= 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.002 * Num_MC_States

- 0.5726 * q_value

+ 1.0028

LM3:

p_value =

-0.0412 * Num_L2

- 0.0009 * Num_threads_per_L2

- 0.0075 * Mapping_Strategy= 2threads_different_L2_same_socket , 8threads_2threads_per_core , 8threads_1threads_per_core , 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.3059 * Mapping_Strategy= 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.0027 * Num_MC_States

- 0.5804 * q_value

+ 0.6007

LM4:

p_value =

-0.016 * Num_L2

- 0.0011 * Num_threads_per_L2

- 0.009 * Mapping_Strategy= 2threads_different_L2_same_socket , 8threads_2threads_per_core , 8threads_1threads_per_core , 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.0547 * Mapping_Strategy= 8threads_4threads_per_core , 4threads_with_1threads_per_L2

+ 0.001 * Num_MC_States

- 0.231 * q_value

+ 0.2471