SCHEDULING OF MULTIPLE MULTI-THREADED APPLICATIONS ON CMPs

by

Sanem Arslan

B.S, Computer Engineering, Marmara University, 2009

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2011

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis supervisors Prof. Oğuz Tosun and Prof. Haluk Topçuoğlu, for their support, encouragement, and guidance throughout my graduate study and completion of this thesis. I also want to thank Prof. Mahmut Kandemir (from Pennsylvania State University) for his help and suggestions. I would also thank to Prof. Fikret Gürgen, Assoc. Prof. Can Ozturan, and Assist. Prof. Esin Onbaşıoğlu for their participation in my thesis jury, their useful comments, and feedback.

I gratefully acknowledge the financial support of The Scientific and Technological Research Council of Turkey (TUBITAK) through a research grant (Project Number 108E035).

I would like to thank my dear friends Betül Demiröz and Işıl Öz for their generous help and encouragement throughout this thesis. I would also thank to my colleagues Fatma Ergin, Berna Kiraz, and Emel Küpcü for their friendship and motivation.

Finally, I would like to thank my father, Hüseyin, my lovely sister, Azime, and my dear fiancé, Tanju, for their encouragement, patience, and support. The biggest credit goes to my mother, Mukaddes, who was my most significant teacher. I am eternally grateful to her encouragement, patience, and support throughout my education. Thanks, Mom. I can feel your presence.

ABSTRACT

SCHEDULING OF MULTIPLE MULTI-THREADED APPLICATIONS ON CMPs

Due to the limitations in the conventional processor designs, chip multiprocessors (CMPs), which have multiple cores on a single chip, are a promising alternative to single-core architectures for performance improvements. The potential performance gains that can be achieved by the using CMPs decline when there is contention for the shared cache structure for multiple multi-threaded applications. Our main focus is to present mapping strategies of multiple multi-threaded applications on multicore architectures. We propose and develop a novel prediction-based mapping strategy. Our approach analyzes thread behavior of different applications on the shared cache by considering all possible thread combinations of different applications. It finds the best thread combinations of different applications that result in minimum cache disturbance. Our prediction-based framework has two components: a static component and a dynamic component. The collection of the training data which is given to the curve fitting model as an input is done off-line at the static component. After receiving the predicted values, the threads of each application that shares the same core are arranged. Communication with curve fitting model, receiving predicted results, and finally mapping according to these values are done on-line at the dynamic component. The communication between the application code and the curve fitting model is provided by a runtime module which collects the training data from the application code and sends them to the curve fitting model and receives predicted data from the curve fitting model and sends them to the application code. Any interference with the program is avoided at every step of the execution.

ÖZET

YONGADA ÇOKLU-İŞLEMCİLİ MİMARİLER İÇİN ÇOKLU ÇOKİZLEKLİ UYGULAMA PLANLAMA

Klasik işlemci tasarımınıdaki sınırlamalardan dolayı, tek bir yongada birden fazla çekirdeğe sahip olan Yongada Çoklu İslemciler (CMP) performans gelişimi için tek çekirdekli mimarilere ümit verici bir alternatiftir. CMP kullanımı ile elde edilebilir performans artışı, çoklu çokizlekli (multi-threaded) uygulamalarda paylaşımlı önbellek yapısındaki çekişmeden dolayı azalabilir. Bizim esas odak noktamız, çoklu çokizlekli uygulamalar için haritalama stratejileri sunmaktır. Biz bu tezde, yeni bir tahmin-tabanlı haritalama stratejisi sunuyor ve geliştiriyoruz. Bu yöntem, farklı uygulamaların izleklerinin paylaşımlı önbellek üzerindeki davranışlarını analiz eder, farklı uygulamaların tüm izlek kombinasyonlarını tahmin eder, ve farklı uygulamaların en az önbellek karışıklığına sebep olacak en iyi izlek kombinasyonunu bulmaya çalışır. Tahmin tabanlı çerçevemizin iki bileşeni vardır: statik bileşen ve dinamik bileşen. Tahmin sürecinin eğitim aşaması statik bileşende çevrimdışı olarak yapılır. Tahmin edilen değerler alındıktan sonra, her uygulamadan kaç adet izleğin aynı çekirdeği paylaşabileceği ayarlanır. İkinci bileşende ise, eğri uydurma modeli ile iletişim kurulması, tahmin sonuçlarının alınması ve bu tahmin sonuçlarına göre en son haritalama izlemi belirlenmesi çalışma zamanında yapılır. Uygulama kodu ile eğri uydurma modeli arasındaki iletişim runtime modulü tarafından gerçekleştirilir. Bu modül, eğitim aşaması için gerekli olan bilgiyi uygulama kodundan alır, eğri uydurma modeline iletir ve tahmin edilen bilgileri eğri uydurma modelinden alır, uygulama koduna gönderir. Hiçbir adımında programa karışılmaz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

3D	Three Dimensional
4D	Four Dimensional
$CAPS$	Cache - Contention Proactive Scheduling
CC	Correlation Coefficient
CMP	Chip Multi-Processing / Chip Multi-Processor
CPI	Cycle Per Instruction
CPU	Central Processing Unit
DML	Device Modeling Language
FFI	First Fit Increasing
$GEMS$	General Execution-Driven Multiprocessor Simulator
GoF	Goodness of Fit
IPC	Instruction Per Cycle
$L1\ Cache$	Level 1 Cache
$L2\ Cache$	Level 2 Cache
$K-NN$	K-Nearest Neighbor
MAE	Mean Absolute Error
MSE	Mean Square Error
PDF	Parallel Depth First
PMU	Performance Monitoring Unit
$RTID$	Related Thread ID
RAE	Relative Absolute Error
$RMSE$	Root Mean Square Error
SSE	Sum of Square Error
SSR	Sum of Square Regression
$SpMV$	Sparse Matrix-Vector multiply
SoC	System on Chip
WS	Work Stealing

1. INTRODUCTION

Processor frequencies may not be increased at arbitrary rates, so processor designers aim to place more processor cores in a chip to increase parallelism and throughput. This tendency resulted in the development of the single Chip Multi-Processors (CMPs) with multiple levels of cache structure. A CMP contains multiple Central Processing Units (CPUs), an interconnection fabric, and some memory components, all packaged into a single chip. CMPs have several advantages over single processor solutions [2–4]. Combining smaller, less complex cores onto a single chip makes it silicon area efficient. Dynamically switching between cores and powering down unused cores make the chip energy efficient. It increases throughput performance by computing multiple jobs simultaneously.

Although cache sharing for CMPs is important for reducing inter-thread communication latency and empowering flexible use of the cache, it has disadvantages like resource contention among co-running processes. This may cause the considerable performance degradation [5]. As the number of cores integrated on a CMP continues to grow [6], an effective way of utilizing CMP architectures is to execute multiple applications at the same time. In this scenario, one critical issue is the management and partitioning of the shared CMP resources such as processor cores and shared on-chip caches.

Multi-threaded programs produce higher throughput than single threaded programs on chip multiprocessors, but the performance gains from increasing threads reduce when there is contention for shared resources [7–9]. Therefore, thread scheduling is an important issue for effective using of shared resources to extract performance from multicore systems. Scheduling and mapping terms are interchangeably used in this study. We target to map threads onto set of cores dynamically using prediction-based mechanisms, whereas scheduling threads of different applications on a given core is handled by OS-scheduler.

In this work, we try to find ways of mapping and dynamically adapting the execution of the multiple multi-threaded applications on CMPs. Our main focus will be on monitoring the effects of threads on each other from different applications and our mapping strategy will be determined based on the obtained results. Different threads of different applications can use the same shared cache in the execution of the program. According to the shared cache disturbance, the threads of each application that share the same core can be arranged. Here, the similarity of applications does not result in cache disturbance. Since we use different applications, the cache usage behavior of different applications results in cache disturbance.

Figure 1.1 gives a case of X different applications each having T threads which are mapped on a N-core CMP. In such a case, it is not possible and practical to try all possible combinations of threads that can execute well in minimum cache disturbance. With small number of combination runs (10%-15%), all remaining combinations can be predicted. To achieve this, curve fitting techniques are used. As a result, we target to determine the best degree of multi-threading for each application and the best thread combination to share the same cache structure.

To achieve that, our framework has two components. The first one is a static component in which the training phase of prediction is done off-line, and the second one is a dynamic component in which the training phase of prediction is done on-line. In the static component, for a given number of multi-threaded applications, one application is fixed as the base application and it is observed how threads of other applications affect the performance of the base application. The base application and different combinations of other applications' threads share the same cache structure. Threads of other applications disturb the base application's data on L2 cache. This disturbance affects the execution time of the program.

In the dynamic component, for a given number of multi-threaded applications, in this case, there is no fixed base application. The degree of multi-threading is fixed for all applications. In the execution of the program, each core is monitored in a proper

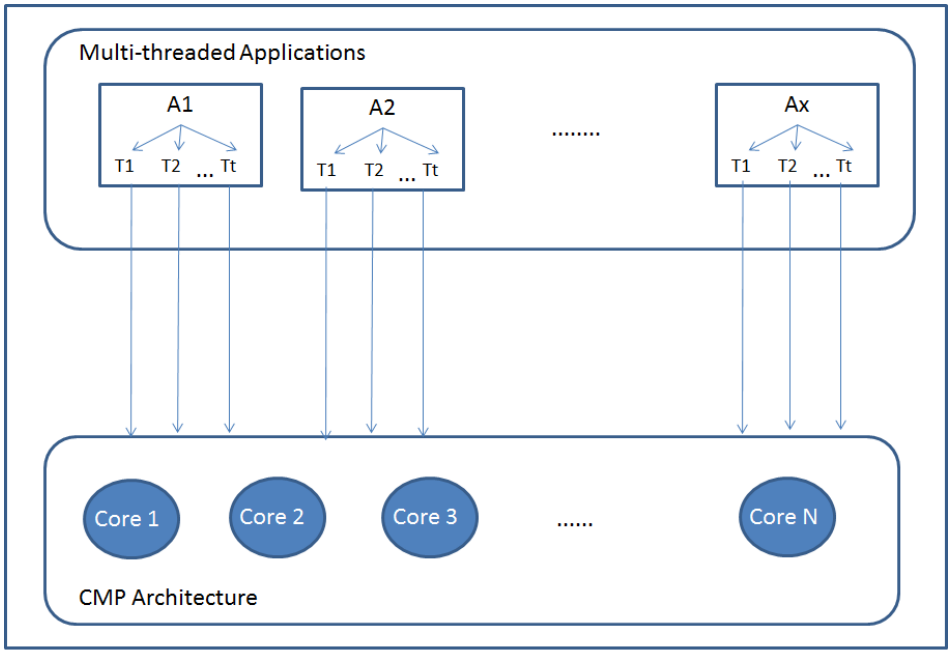

Figure 1.1. Running Multiple Multi-Threaded Applications on an N-core CMP.

period of time and training data is collected for curve fitting model. After sending training data and receiving the predicted data from the curve fitting model, the final thread mapping strategy is determined. The same tasks for the static component are performed in an automatic way in the dynamic component. Any interference with the program is avoided at every step of the execution. Therefore, the dynamic component is more complicated and difficult to implement but more practical to use.

The remainder of thesis is organized as follows. In Chapter 2, we provide a brief background and literature review of application scheduling and mapping for multicore architectures and performance prediction. In Chapter 3, we present our prediction based thread scheduling framework for multicore systems. This describes the static and dynamic components that we have used. In Chapter 4, the evaluation of our framework and experimental study is examined. Finally, Chapter 5 summarizes the contributions of the application mapping approaches proposed in this thesis.

2. RELATED WORK

As increased processor clock rates have come to a limit, CPU designers aim to place more processor cores in a chip to increase parallelism and throughput. This tendency has led the development of single chip multiprocessor (CMPs) with multiple levels of cache structure. Multi-threaded programs produce higher throughput than single threaded programs on CMPs, but the performance gains from increasing the number of threads reduces when there is contention for shared resources [7–9]. Therefore, thread mapping is an important issue for effective use of shared resources on multicore systems. In this section, we summarize related work on mapping of multi-threaded applications. Then we present performance prediction and estimation techniques given in the literature.

2.1. Thread Mapping and Scheduling

Efficient task mapping and scheduling is critical for achieving high performance in heterogeneous parallel and distributed systems. There are many scheduling algorithms for parallel and distributed systems in the literature, and these algorithms are divided into two groups according to when the scheduling action is taken. In static scheduling, mapping of the tasks to processors is decided upon compile time, whereas in dynamic scheduling, this decision is taken at runtime. In this study, we consider both static and dynamic assignment techniques.

A recent study [10] presents multi-threaded applications into two phases: serial and parallel. In the serial phase, there is only one active thread in the system that makes data preparation for the parallel phases. In the parallel phase, there are many active threads in the system and heavy independent calculations are performed in this phase. Current scheduling techniques do not differentiate between these phases, resulting in sub-optimal utilization of multiprocessor resources. This lack of awareness results is lower throughput and jitter in the application's runtime and unfairness between the

applications. Serial phases compete for CPU time with many concurrently executing parallel threads. If the serial phases were executed quickly, this bottleneck could be freed. Therefore, the authors presents a new thread scheduling technique that takes into account the different requirements of each phase [10]. First, the scheduler monitors the number of active threads in each application, and then grants higher priority to the serial threads. As a result, when a serial thread is executed concurrently with a parallel thread, the serial thread is always granted a core for itself. Based on their experimental study, the jitter in the execution time is decreased by as much as 88%. The throughput in some cases increases by more than 16%, and the fairness metric is improved by up to 26%.

As we mentioned, multi-threaded programs produce higher throughput than single threaded programs on chip multiprocessors, but the performance gains from increasing threads decline when there is contention for shared resources. A resource aware application co-scheduling is proposed in [7] in order to achieve better performance and energy results. They determine resource requirements by examining cache miss rates and data bus contention. Then this information is used to schedule threads to cores. Scheduling programs that are compatible in shared resources minimizes contention. With this scheduling technique, the overall performance is improved by 19%, and energy consumption is reduced by 26%. The aim of their paper is same as our approach, but they examine application characteristics rather than thread characteristics. Their framework is performed in a static way and does not suffer from thread migration, which is a serious cost for us.

In another work, program locality analysis is combined with job co-scheduling [11]. They claim that locality analysis offers a large scope of application's data access patterns and cache requirements. Their lightweight locality model predicts the cache requirements and the co-run performance of programs. The locality analysis based on reuse distance between the programs is examined. They propose a cache-contention aware proactive scheduling (CAPS) technique, which embeds this locality model into runtime scheduling. This scheduling technique assigns processes to processors according

to the predicted cache-contention sensitivities. It reduces the performance degradation by 15.7% and unfairness by 47% than the default scheduler. Among the 12 programs from SPEC CPU2000 benchmarks, 7% speed up is reached. Their aim is also similar with our approach, but our framework for cache requirements is not based on reuse distance.

A new operating system scheduling algorithm is proposed for multicore systems in [12]. Their algorithm is a cache-fair algorithm that decreases the effects of unequal CPU cache sharing. It redistributes CPU time to threads according to their unequal cache sharing. If the performance of a thread decreases on account of unequal cache sharing, it gets more time. How a thread's performance is influenced by unequal cache sharing is determined by limited information from hardware. The runtime statistics and analytical models are used. With this scheduling technique, the co-runner dependent performance that is the result of unequal cache sharing is reduced significantly.

In another study [13], the authors compare the performance of two schedulers that are Parallel Depth First (PDF) which is designed for constructive cache sharing, and Work Stealing (WS), which is a more popular and traditional design. By more efficient use of cache resources, the PDF scheduler outperforms the WS. They claim that task granularity is important for cache performance. An automatic approach for electing effective grain sizes for PDF technique is also presented in the scope of this paper. In [14] they compare scheduling algorithms for heterogeneous many core architectures and propose Hierarchical Hungarian Scheduling Algorithm, which is based on linear programming approach. This algorithm decreases the scheduling overhead without a loss of accuracy.

In another study [15], the authors propose a new scheduling technique based on sharing patterns of threads with the help of performance monitoring units (PMUs). PMUs are fine-grained and available in today's processing units. They have relatively low overhead. With this scheduling technique performance improvement is provided up to 7%. In [16] the authors present an analytical model to predict the cost of running an

affinity-based thread schedule on multicore systems. This model has three sub-models to calculate the cost of executing a thread schedule. These are an affinity-graph sub-model, a memory hierarchy sub-model, and a cost sub-model. With this estimated cost, they determine which schedule will provide better performance. An approximation algorithm is designed to compute near-optimal solutions. Using optimized thread schedule can improve the program performance by 25%.

A new thread scheduling algorithm is based on core affinity and distance based migration is proposed in [17]. The authors present a scheduling framework for multicore systems that uses high-level information supplied by the user to guide thread scheduling. They group of threads under the concept of Related Thread ID (RTID), which is a collection of threads that share some data. It provides a high level of abstraction for the scheduler to improve runtime performance.

Core-partitioning techniques are examined in some of the related work [8, 18]. The authors propose an algorithm that dynamically divides available cores for multiple multi-threaded applications that are simultaneously running on the same CMP. This partitioning technique uses an input from a curve fitting result that predicts the best operating points for an application at runtime. Six multi-threaded applications are used, and it is assumed that two threads of different applications are not mapped onto same core. In our approach, we also use a runtime module that runs the curve fitting model and returns the best thread combinations, but different threads of different applications can share the same core.

2.2. Prediction

In our approach, there is a dynamic performance prediction part that predicts the behavior of all thread combinations. Performance prediction, estimation, and analysis are widely studied topics in the literature. In [9], the authors try to predict the L2 cache contention of a program with the help of five machine learning algorithms. The first one is Linear Regression, which assumes the existence of linear relationship between the

variables. The second one is Artificial Neural Networks, which is based on co-operative information processing as neurons in the brain. The third one is Locality Weighted Linear Regression, which is based on instance based learning algorithms. The fourth one is Model Trees, which are binary decision trees with linear regression functions at the leaf nodes. And the last one is Support Vector Machines, which find the instance called support vector in boundary classes. These algorithms generate the regression models used for prediction phase. PMUs are used as in [15] to measure performance events. The prediction metrics they used are Correlation Coefficient (C), Mean Absolute Error (MAE), Root Mean Square error (RMSE), Root Relative Squared Error (RRSE), and Relative Absolute Error (RAE). According to the results they have found, the best appropriate algorithm is Model Tree with the minimum error value. Their approach is similar to ours with finding L2 cache contention with the prediction methods, but their methods are machine learning methods.

In another study [19], the authors propose a new method for estimating execution times of applications in order to use it for dynamic scheduling in heterogeneous systems. The execution time is modeled as a random variable and k-Nearest Neighbor (K-NN) regression algorithm, which is a method for classifying objects based on closest training examples in the feature space [20] is used to make a prediction from a set of observations. According to the results, their method can make accurate execution time predictions. In [21], the authors claim that as heterogeneous multicore systems grow, future compilers need to select from very large implementation of a given function to optimize these systems. They proposed a heuristic for compilation tools that statistically selects important input values, measures the execution time, and then predicts the execution time of all combinations to select the best implementation of a given function. In this prediction mean error and root-mean square error metrics are used. Their heuristic achieves a 6.2% prediction error and a 7.4% root-mean-square-error. In [22], a model for prediction of popular weather forecasting application on multicore systems is proposed. A mathematical method that uses regression analysis for performance prediction is used. The error percentage is used for validating their approach. With this heuristic, a 10% average prediction error is achieved for multicore systems.

A number of papers are analyzed about performance prediction. Since the relation between the variables is not known, a specific model cannot be used as described above. Cubic interpolation type is used for static component and n-dimensional polynomial fitting type is used for dynamic component due to the lower error values.

3. PREDICTION-BASED THREAD SCHEDULING FRAMEWORK

In this work, we try to find ways of mapping and dynamically adapting the execution of the multiple multi-threaded applications on CMPs. Our main focus will be on monitoring the effects of threads on each other from different applications and mapping strategy will be determined based on the obtained results. Different threads of different applications can use the same shared cache in the execution of the program. According to the shared cache disturbance, it can be determined that how many threads of the applications can share the same cache. It is not possible and practical to try all possible combinations of threads that can execute well in minimum cache disturbance. With small number of combination runs (10%-15%), all remaining combinations can be predicted. To achieve this, curve fitting techniques are used. As a result, we target to determine the best degree of multi-threading for each application and best thread combination to share the same cache structure.

To achieve that, our framework has two components. The first one is a static component in which the training phase of prediction is done off-line, and the second one is a dynamic component in which the training phase of prediction is done on-line. In this chapter, details of both components are explained.

3.1. Static Component

The goal of our work is to find the best thread combinations of different applications that result in minimum cache disturbance. In the static component, communication with curve fitting model, receiving predicted results, and determining final mapping strategy according to these results are done statically. For a given number of multi-threaded applications, one application is fixed as the base application and the effect of other applications on the performance of base application is observed. The base application and different thread combinations of other applications share the same

cache structure. Threads of other applications disturb the base application's data on L2 cache. This disturbance can be monitored in execution time of the program.

At the first attempt, three applications are selected and one of them is selected as the base application. All of them share the same L2 cache structure and they are executed simultaneously. The base application runs on one thread and the degree of multi-threading is varied for other two applications. The program is executed for all possible combinations and after its completion, a small number of combinations are selected randomly and sent to the curve fitting model. After receiving the predicted results, the remaining combinations and predicted results are compared. This shows the accuracy of prediction part. As a result, instead of trying all possible combinations, the predicted results determine the final mapping strategy.

3.1.1. Applications

In order to evaluate our approach, three applications Jacobian Matrix Calculation, Barnes-Hut, and Sparse Matrix-Vector Multiplication (SVM) algorithms are used in the static component. All the applications are parallel (multi-threaded) and implemented using *Pthread API*. The important characteristics of applications are explained below.

- Jacobian Matrix Calculation performs addition operations on the elements of a given matrix A and stores the result in another matrix B. For the calculation of each data element of matrix B, row-wise and column-wise neighboring data elements are accessed in matrix A.
- Barnes-Hut Method is one of the most popular approximation algorithms, and reduces the computational complexity of the N-body problem to O(n log n). In this method, the data elements of application threads are dynamically mapped onto on-chip memory components. This method includes HW and SW optimizations to take full advantage of the resources offered by the architecture.
- Sparse Matrix-Vector Multiply (SpMV) is an important kernel in scientific computing, which has irregular data access patterns due to matrix sparsity. SpMV

works on a sparse matrix A and a vector x and calculates Ax=B.

In the static component, the Jacobian application is fixed as a base application and it runs on one thread. The other two applications run on multiple threads from 0 to 16. The effects of Barnes-Hut and SpMV threads on the Jacobian thread are monitored.

3.1.2. Target Architecture

The target multicore machine is a 2-core machine, where each processor has its own private 32 KB L1 (data and instruction) caches and each of the two cores share an on-chip 1 MB L2 cache, as shown in Figure 3.1. We map multiple multi-threaded applications on this architecture and the threads of different applications can share the same core. Since we monitor L2 cache disturbance, even if we create a 2-core machine, both of them are connected to the same shared L2 cache. The initial thread mapping is left to the operating system because no matter which threads are assigned to which core, all of them use the same shared L2 cache structure. The important characteristics of the simulated multicore and the simulated cache structure are given in Table 3.2

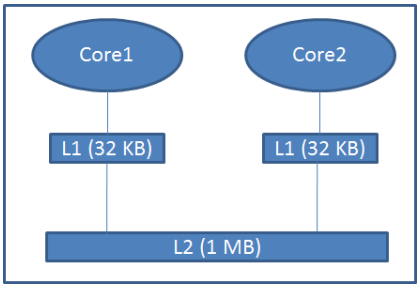

Figure 3.1. Target Architecture for Experimental Study.

3.1.3. Curve Fitting Model

Curve fitting is the process of plotting a curve that has the best fit to a series of data points [23]. In the static component, the Surface Fitting Tool, which is a product

Table 3.1. Simulated Architecture Parameters.

Processor type	X86Pentium4
Processor frequency	1 GHz
Hard disk size	20GB
Main memory size	256MB
L1 cache size	32K/core
L2 shared cache size	1MB
Operating system	Linux Fedora 5

of MATLAB, is used as a curve fitting model. With the help of this toolbox, 3D curve fitting techniques can be used. The Surface Fitting Tool comes with the MATLAB 2009 b release in the curve fitting toolbox, which makes it easy to plot and analyze fits. Interactive analysis can be done easily with the help of graphical user interface of the Surface Fitting Tool. After creating a fit, a variety of post-processing methods can be applied for plotting, interpolation, and extrapolation. It offers us four different types of fits [24]:

- Interpolation, which is a curve fitting model in which the curve must go exactly through the data points [25].
- Polynomial fit type finds the coefficients of a polynomial that fit a set of data in a least-squares sense. It is used for linear regression [26].
- Lowess (Locally Weighted Scatter Plot Smoothing) fit type is used for local smoothing regression.
- Custom Equation is a fit category for defining our own equation for nonlinear regression.

After the fitting of a curve with the methods given above, MATLAB returns Goodness of Fit (GoF) statistics that describe how well the data fits a set of observations. The curve supports these goodness-of-fit statistics for the following parametric

models:

- Sum of Squares Due to Error (SSE): This statistic measures the total deviation of the response values from the fit to the response values. It is also called the summed square of residuals and is usually labeled as SSE [27].
  For example,

$$y_i = a + b_1 x_{1i} + b_2 x_{2i} + \cdots + \varepsilon_i, \tag{3.1}$$

  where y_i is the i^{th} observation of the response variable, x_{ji} is the i^{th} observation of the j^{th} explanatory variable, a and b_i are coefficients, i indexes the observations from 1 to n, and ε_i is the i^{th} value of the error term.
  If $\hat{a}$ and $\hat{b}_i$ are the estimated coefficients, then

$$\hat{y}_i = \hat{a} + \hat{b}_1 x_{1i} + \hat{b}_2 x_{2i} + \ldots \tag{3.2}$$

  is the i^{th} predicted value of the response variable. The SSE value of i^{th} element is the square of the difference between the i^{th} observation and the i^{th} predicted value:

$$SSE(i) = w_i \left(y_i - \hat{y}_i\right)^2 \tag{3.3}$$

  The SSE is the sum of the squares of the differences of the each observations and the predicted values:

$$SSE = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i\right)^2 \tag{3.4}$$

- R-square: This statistic measures how successful the fit is in explaining the variations of the data. R-square is the square of the correlation between the response values and the predicted response values. It is also called the square of the multi-

ple correlation coefficient and the coefficient of multiple determination. R-square is defined as the ratio of the sum of squares of the regression (SSR) and the total sum of squares (SST). SSR is defined as the sum of the squares of the differences of the predicted values and the mean value of the response variable:

$$SSR(i) = w_i \left(\hat{y}_i - \bar{y}\right)^2 \tag{3.5}$$

$$SSR = \sum_{i=1}^{n} w_i \left(\hat{y}_i - \bar{y}\right)^2 \tag{3.6}$$

SST is also called the sum of squares about the mean, and is defined as the sum of squares of the difference of the dependent variable and its mean:

$$SST(i) = w_i \left(y_i - \bar{y}\right)^2 \tag{3.7}$$

$$SST = \sum_{i=1}^{n} w_i \left(y_i - \bar{y}\right)^2 \tag{3.8}$$

where SST = SSR + SSE. Given these definitions, R-square is expressed as

$$R - square = \frac{SSR}{SST} = 1 - \frac{SSE}{SST} \tag{3.9}$$

- Root Mean Squared Error (RMSE): RMSE is a frequently used measure of the difference between values predicted by a model and the values actually observed from the environment that is being modeled. These individual differences are also called residuals, and the RMSE serves to aggregate them into a single measure of predictive power. This statistic is also known as the fit standard error and the standard error of the regression. It is an estimate of the standard deviation of the

random component in the data, and is defined as

$$RMSE = s = \sqrt{MSE} \tag{3.10}$$

where MSE is the mean square error or the residual mean square

$$MSE = \frac{SSE}{n} \tag{3.11}$$

Evaluation of the static component is shown in Experimental Setup and Results section. In the static component, the program is executed with three applications. After the completion of the program, a small number of combinations are selected and sent to the curve fitting model in a static way. After receiving the predicted values, we make a decision about which combinations should be selected. Then the program is executed with the given selected combinations and initial thread mapping is done according to these combinations. All the procedure is processed in a static way. However, there is another alternative, which is to provide a dynamic prediction-based thread scheduling framework.

3.2. Dynamic Component

The goal of our work in this section is to find the ways of mapping and dynamically adapting the execution of the multiple multi-threaded applications on CMPs. The same tasks in the static component are performed in an automatic way in the dynamic component. Collecting the training data, communication with the curve fitting model in MATLAB, receiving predicted results, and determining the final mapping strategy according to these values are done at runtime. Any interference with the program is avoided at every step of the execution. Therefore, the dynamic component is more complicated and difficult to implement but, is more practical to use.

In dynamic component, for a given number of multi-threaded applications, in this case, there is no fixed base application. The degree of multi-threading is fixed for all applications. In the execution of the program, each core is monitored during periodic time intervals and training data is collected for curve fitting model. After sending training data and receiving the predicted data from curve fitting model, a final thread mapping strategy is determined.

At a first attempt, three applications, each have 32 threads running on an 8-core machine. Initial thread mapping is done randomly. No core is allowed to be idle. While the threads are running on the cores, each core is monitored during periodic time intervals. These periodic time intervals are called as *epochs*. The thread number of each application for each core is stored at the end of each epoch.

For example, the termination of the epochs is:

- From the three applications, 96 threads are mapped onto eight cores randomly.
- At the end of first epoch, eight thread combinations from eight cores are stored.
- Some threads are selected randomly and migrated to the different cores to increase diversity.
- After some time passes, second epoch combinations are stored from each core.
- At that time, there are 16 thread combinations from eight cores.
- This procedure continues like this until a sufficient amount of combinations is obtained.

Determining epoch value and the total number of epoch values are parameters that can be changed. The epoch value is analyzed in more detail at Epoch Value and Performance Metric section. After getting sufficient amount of training data, this data should be sent to the curve fitting model in MATLAB. However, our application code runs on a target machine and MATLAB runs on a host machine. To communicate the application code to MATLAB, a runtime module is used as an external module. This module behaves like a bridge between our application code and the curve fitting model.

The details of this module are explained in the Runtime Module section. Application code sends the training combinations to the runtime module and forwards this data to the curve fitting model in MATLAB. After the curve fitting model predicts the remaining combinations, it sends this data to the runtime module again. The runtime module selects the best eight feasible combinations from the predicted list and sends them to the application code again. Application code does the final thread mapping according to these combinations. The details of dynamic component is explained in the sub sections.

3.2.1. Target Architecture

The target multicore machine is 8-core machine, where each processor has its own private L1 (data and instruction) caches and each two cores share on-chip L2 cache, as in Figure 3.3. Threads of each applications are mapped on this architecture and the threads of different applications can share the same core. Since L2 cache disturbance is monitored, even if 8-core machine is created, each two cores are connected to the same shared L2 cache. The important characteristics of the simulated multicore and the simulated cache structure are given in Table 3.2

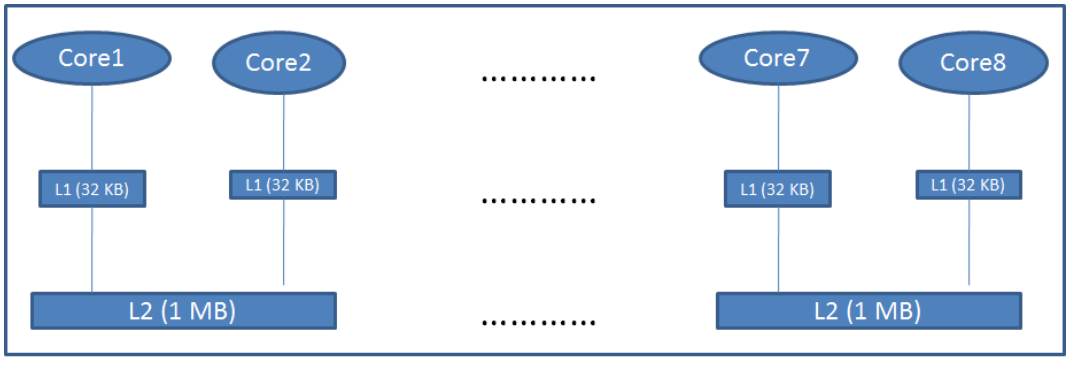

Figure 3.2. Target Architecture for Dynamic Component.

Table 3.2. Simulated Architecture Parameters.

Processor type	X86Pentium4
Processor frequency	1 GHz
Hard disk size	20GB
Main memory size	256MB
L1 cache size	32K/core
L2 shared cache size	1MB
Operating system	Linux Fedora 5

3.2.2. Applications

In order to evaluate dynamic component, three applications which are Jacobian Matrix Calculation, Black-Scholes Method and Sparse Matrix-Vector Multiplication (SpMV) algorithms are used. Each application has 32 threads and implemented using *Pthread API*. Since the details of Jacobian Matrix Calculation and SpMV applications are explained in the static component, only the details of Black-Scholes Method are explained here.

The Black-Scholes application is from the PARSEC Benchmark Suite [28]. It uses the Black-Scholes partial differential equation to calculate the prices for a portfolio of European options [29]. The program is parallelized by dividing the number of portfolios into a number of work units and assigning work units to application threads dynamically. Each thread iteratively calculates the price of the given portfolio for a predefined time interval.

3.2.3. Epoch Value and Performance Metric

As mentioned, each core is monitored during periodic time intervals. These periodic time intervals are called *epochs*. The thread number of each application for each

core is stored at the end of each epoch. Epoch value is determined according to the iteration number. For example, the first five iterations could be first epoch, and the other seven iterations could be the second epoch, etc. The size of second epoch is larger than the first one because of the thread migration. As we said, at the end of each epoch, some threads are migrated to the other cores. The actual happening of this situation takes some time in the physical environment. Therefor,e the size of other epochs is larger than the first one.

In this approach, all cores are not obliged to have the first epoch at the same time. Each core can arrive its first epoch at different times, but the epoch size should be same for all cores. The total number of epochs simply determines the size of the training set. If the epoch size is too large, the training set size also becomes too large. This makes the prediction phase more accurate, but decreases the effect of the post-prediction phase, which is more important for our framework. If the epoch size is too small, the training set size also becomes too small. This makes the prediction phase less accurate but increases the effect of the post-prediction phase. Therefore, determining the epoch size is very important to extract more efficiency in this framework. Determining the epoch value and the total number of epoch values are parameters that can be changed.

As a performance metric, execution time is used. At the end of each epoch, in addition to the thread number of each application, the execution time at that point is also recorded. The curve fitting model predicts the execution time values for all possible thread combinations, with given training thread combinations and execution times.

3.2.4. Runtime Module

As mentioned, there is a runtime module that takes input thread combinations from the application code and forwards them to the curve fitting module as an input, receives the predicted values from curve fitting model, and sends the most feasible eight thread combinations to the application code. This module behaves like a bridge between our application code and the curve fitting model. This runtime module is the

g-cache module of the simulator program (SIMICS) we have used. The g-cache and curve fitting modules are explained in more detail in the below.

3.2.4.1. G-Cache Module. In order to communicate between the application code and the curve fitting model, the *g-cache* module of the simulator program (SIMICS) is used. The g-cache module is the standard cache model of the SIMICS. It handles one transaction at a time in a flat way: all needed operations (copy-back, fetch, etc.) are performed in order and at once. The cache returns the sum of the stall times reported for each operation. It publishes cache statistics such as read misses, write misses, etc. It is a complex code written in "C" programming language and can be modified by user [30]. The g-cache module has been modified to communicate with the curve fitting model. This module runs whenever the program begins to execute and it runs on host machine on which MATLAB also runs.

It has several complicated source files and it is hard to make a decision about where our source codes should be injected. We have modified the gc.c file of g-cache module to communicate with the curve fitting model. G-cache module becomes a mid-layer between our application code and the curve fitting model in MATLAB.

3.2.4.2. Curve Fitting Model. In the static component, the Surface Fitting Tool, which is a product of MATLAB, is used as a curve fitting model. With the help of this toolbox, 3D curve fitting techniques can be applied. However, in the dynamic component there are four variables that are the thread number of the first, second, and third applications, and the performance value. Therefore 4-dimensional curve fitting techniques should be used. It is not as easy as using 3D curve fitting because the Surface Fitting Tool handles 3D curve fitting from a graphical user interface, but no tools are readily available in order to achieve 4-dimensional curve fitting. In order to achieve this, an *n-dimensional polynomial fitting* is used. The first reason for using this fitting type is that it satisfies the usage of 4-dimensional fitting. The second and the most important reason is that we do not know the relationship between the variables. If we knew the relationship between the variables, we could use the most suitable fitting type. This would permit

the usage of any degree of polynomial. If the first degree of polynomial is used, it gives us a linear curve; if the second degree of polynomial is used, it gives us quadratic curve; if the third degree of polynomial is used, it gives us cubic curve; etc. Since the minimum error value is obtained from fourth degree polynomial, quartic curve fitting is used.

3.2.5. Execution Steps

In order to evaluate our framework, an example is explained in this part. Suppose there is a 8-core machine and three different applications each having 32 threads. Initial thread mapping is done randomly because no core is allowed to be idle. After some time passes, at the end of first epoch, each core is monitored and the thread number of each application is stored for each core. At the end of first epoch, there are eight thread combinations from eight cores. After the first epoch some threads are selected randomly and migrated to the other cores. At the end of second epoch, there are additional eight combinations, and 16 combinations in total. At the end of third epoch, there are additional eight combinations and 24 combinations in total. This scenario can be seen in Figure 3.3. After initial mapping, for the first core, there are three threads from the first application, five threads from the second application and seven threads from the third application. This combination and the execution time value at that point is recorded. Then three threads of the first application are migrated to the other cores. Second application threads remain as the same, and three additional threads from the third application are migrated to the first core. This combination and the execution time value is also recorded at the end of second epoch. This procedure continues until there is a sufficient amount of data.

When these combinations and execution time values are recorded, this data should also be sent to the g-cache module. Communication between the host machine and the target machine is handled with the *ebx* register. A side that wants to communicate with the other side, writes the data to the ebx register, and the other side reads the data from this register. The ebx register is a 32-bit register, and we should send more than one value like the thread number of each application, the epoch value, and the

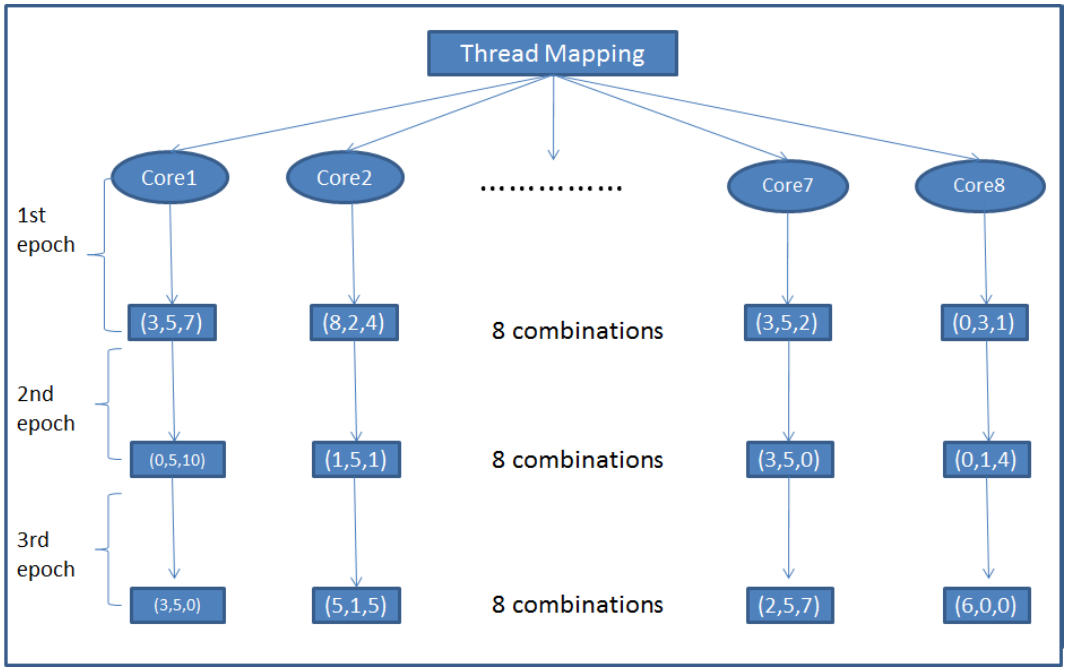

Figure 3.3. Basic Scenario.

core information. To send this data to the g-cache module, a codeword is formed like in Figure 3.4. The first digit can take the values of 1,2 or 3. The value of 1 indicates that the program is at the start phase and g-cache module publishes the initial cache statistics. The value of 2 indicates that combinations are coming in following digits. The value of 3 indicates that the program is at the end and g-cache module publishes the end cache statistics. In Figure 3.4, the value of first digit is 2, which says that combinations are coming in the following digits. Each two digits are reserved for information. The value of 04 indicates the epoch value of the core, the value of 05 indicates the thread number of first application, the value of 06 indicates the thread number of second application, and the value of 11 indicates the thread number of third application. As it is noted, the core information is not sent within this register. The g-cache module can read the core information of the thread that sends the ebx register.

After the g-cache collects the training set from the application code, it should send this information to the curve fitting model in MATLAB. In the source code of the

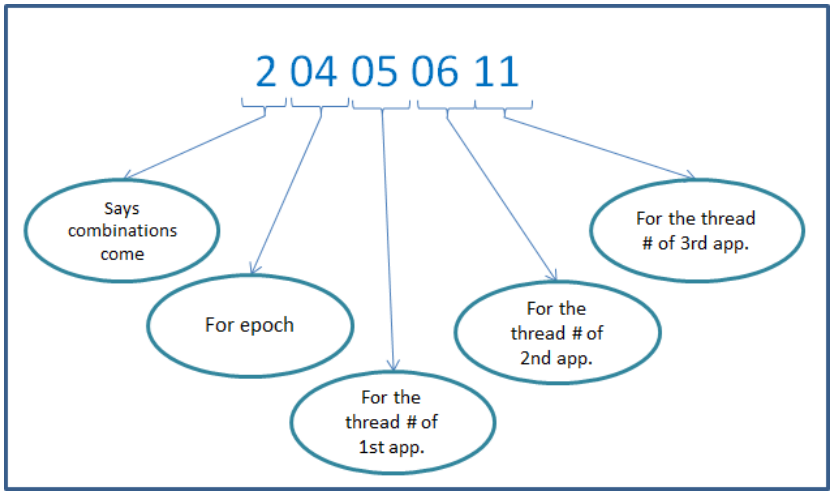

Figure 3.4. The ebx Register.

g-cache module, the MATLAB engine is called with external commands. The training data is sent to the MATLAB side with *memcpy* commands of the C library. This communication can be seen in Figure 3.5.

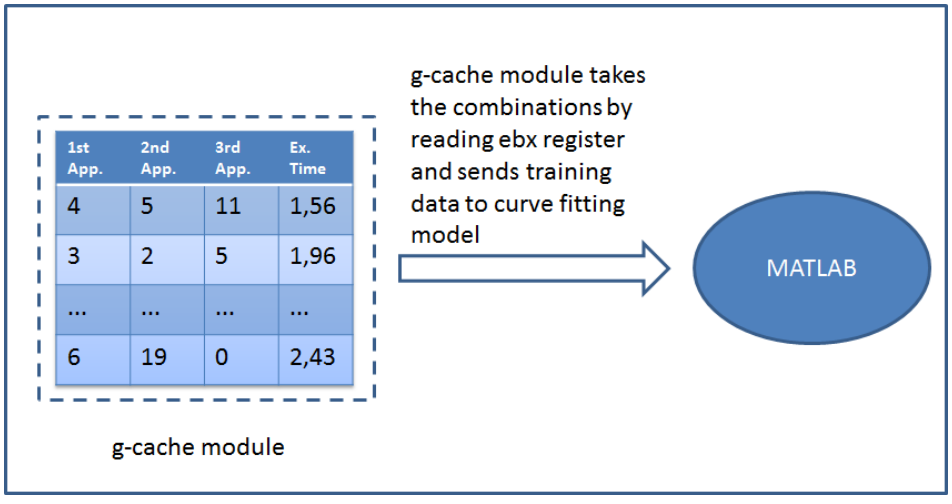

Figure 3.5. G-cache - Matlab Communication.

MATLAB executes a 4-dimensional polynomial fitting and sends back the predicted values to the g-cache module. As shown in Figure 3.6, MATLAB predicts the performance values of all possible combinations for which the size is 32x32x32. The

g-cache receives the predicted values again with the help of *memcpy* commands of the C library.

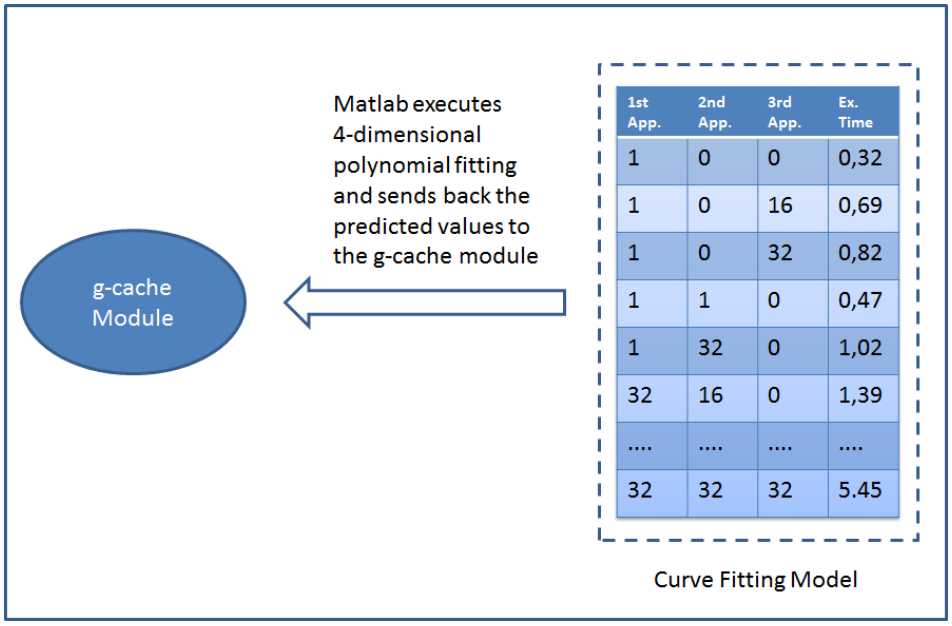

Figure 3.6. Matlab - G-cache Communication.

After getting the predicted values, the g-cache module should send them to the application code. However, sending 32x32x32 combinations to the application code is not practical. It should send the necessary amount of data which is eight combinations for eight cores in this case. The g-cache module should select eight combinations, but it is hard to determine which combinations to send. Sending the best eight combinations does not make sense because the total number of threads in the combinations can exceed the total number of threads in the application code. Therefore, sending the most feasible eight combinations is more logical. The total number of threads for each application in this eight combinations should not exceed 32 for each application. This problem is seen as the *Bin Packing Problem* in the literature. The Bin Packing Problem is a combinatorial NP-hard problem, in which objects of different volumes must be packed into a finite number of bins of capacity V in a way that minimizes the number of bins used [31]. In our case, the cores behave like bins and the combinations behave like objects. In order to solve this problem, the *First Fit Increasing* (FFI) algorithm

is used. The First-Fit algorithm places an item in the first bin in which it fits. The FFI algorithm handles items in non-decreasing order with respect to their sizes, placing them using First-Fit [32]. In our case, the combinations are sorted in non-decreasing order of execution time, then first fit algorithm is applied. The pseudo-code for the FFI algorithm can be seen in Figure 3.7. First of all, the combinations are sorted into increasing order. Then beginning from the combination that has the smallest execution time value,we are looking for a core in which this combination fits. If this combination exceeds the total number of threads, we discard it and look for a new combination in the increasing order list.

```
FirstFitIncreasing()
core[8] ⇐ 0 ;
sum ⇐ 0 ;
j ⇐ 0 ;
// Sort predicted_z values into increasing order, giving the sequence
// predicted_z[1]<=predicted_z[2]<= ... <=predicted_z[n].
for i = 1 to n do
   //Look for a core in which predicted_z[i] fits
   sum += predicted_z[i];
   if  sum < THREAD_NUM then
      core[j++] = predicted_z[i];
   else
      sum -= predicted_z[i];
   end if
end for
```

Figure 3.7. Pseudo-code for First-Fit Increasing Algorithm.

The overall scenario is summarized in Figure 3.8. Application threads begin to run on the target machine and the initial thread mapping is done randomly. At the end of each epoch time, a responsible thread on each core sends the core information to the g-cache module with the help of the ebx register. After the g-cache module

collects the training data, it runs the MATLAB engine with external commands and sends the training data with the help of memcpy commands. MATLAB executes the 4-dimensional polynomial fitting and predicts the execution time values of all possible thread combinations. The g-cache module receives the predicted values with the help of memcpy commands again. The g-cache module applies the FFI algorithm and finds the most feasible eight thread combinations for eight cores and sends them to the application code. The last thing is doing final mapping according to the selected thread combinations.

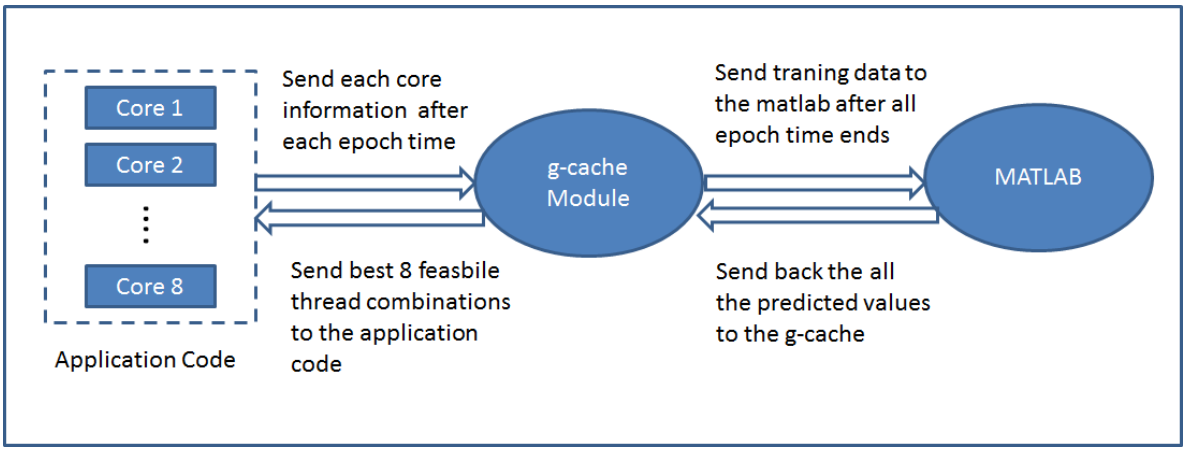

Figure 3.8. Overall Scenario.

Final mapping at the application code side is handled by main thread of the program. Instead of mapping the threads from beginning to end, the main thread does the *migration-aware mapping*. Before doing the final mapping, the main thread monitors the thread numbers on each core. For each core, if there are threads on it, the main thread adds or removes the necessary number of threads according to the prediction results. An example can be seen in Figure 3.9. In this example, according to the prediction results, in the first core, there should be six threads of the first application, five threads of the second application and three threads of the third application. At the current time, there are four threads of the first application, five threads of the second application and eleven threads of the third application in the first core. In such a case, the main thread adds two threads for the first application, remains the same for the

second application threads, and removes eight threads of the third application from the first core. Therefore, it makes migration-aware mapping. This approach protects us from a large amount of migration, which is serious overhead for program performance.

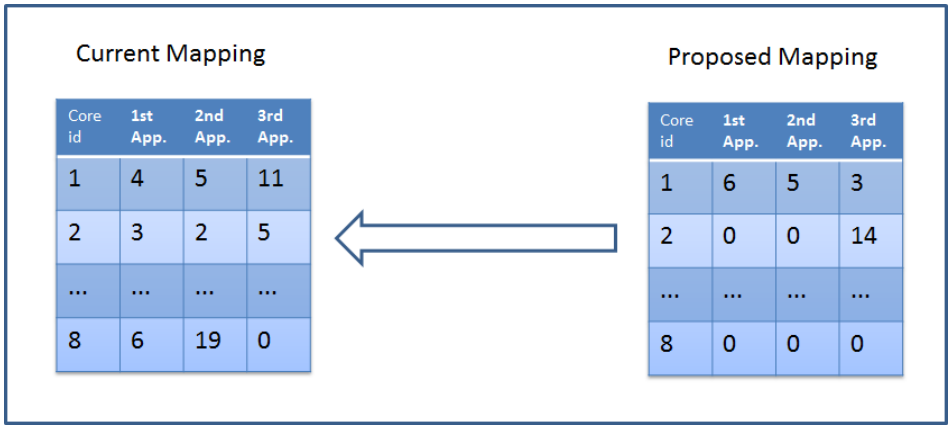

Figure 3.9. Migration Aware Mapping.

Cache statistics are also examined to show how efficient our dynamic framework is in order to decrease L2 cache contention. In order to access the cache statistics, the application should communicate with the Simics simulation environment. The application sets the ebx register on the Intel x86 architecture and calls a special NOP command, which is termed the magic instruction by the simulator. As soon as the magic instruction is called, the application stops and the simulator is invoked by a hap callback. The simulator calculates the cache hits and misses of all caches defined and calculates the overall miss ratio. The miss ratio is sent to the application by using ebx register, and the application continues its execution.

On Intel x86 architecture, we used the gcc inline assembly code as shown in Figure 3.10. Data passing to the Simics simulation environment by using the ebx register is shown on line 3. To stop the execution of the application and start cache statistics calculations, a special NOP command (xchg) given in line 4 is used. After Simics completes the cache statistics collection, it sends the miss ratio to the application through ebx register, and the miss ratio is accessed by the application using the val

variable as in line 5.

```
static inline unsigned int myMagic(unsigned int n)
val = n ;
asm volatile ("movl %0, %%ebx" :: "g" (val) : "ebx")
asm volatile ("xchg %bx,%bx")
asm volatile ("movl %%ebx, %0" : " = g" (val) :)
return (unsigned int) val
```

Figure 3.10. Gcc Inline Assembly Code for Linux Running on x86 Intel Architecture.

The pseudo-code for main function is shown in Figure 3.11. Here, some initializations are done and then application threads are created. The main thread only does some initializations and final mapping. The pseudo code for applications threads is shown in Figure 3.12. Each application creates its own application threads according to the value of THREAD_NUM which is initialized in the main function. Each application makes some calculations according to the characteristics of the specified application. A responsible thread for each core sends the core information running on at the end of each epoch time.

The pseudo-code for g-cache module is shown in Figure 3.13. The g-cache module takes each magic callback and increases counter value by 1. If this counter value shows the last epoch of last core, then it calls the matlab engine. It puts the variables to the Matlab Workspace with *engPutVariable* command, executes 4-dimensional polynomial fitting with *engEvalString* command, and receives the predicted results with *engGetVariable* command. After getting the predicted results, it applies the FFI algorithm in Figure 3.7 to select the most feasible eight combinations. Then it sends them to the main thread of the application code.

```
Main ()
CORENUM ⇐ 8 ;
THREAD_NUM ⇐ 32 ;
EPOCH ⇐ 4 ;
CORE_LIMIT ⇐ 10 ;
for i = 0 to CORENUM do
  core_epoch[i] ⇐ 0;
  core_counter[i] ⇐ 0;
end for
Create Jacobian application thread
Create Black − Scholes application thread
Create SpMV application thread
    Wait for prediction results from g-cache
    Arrange the combinations to do Migration − Aware Mapping
    Do Final Mapping
Wait for Jacobian application thread
Wait for Black − Scholes application thread
Wait for SpMV application thread
```

Figure 3.11. Pseudo-code for Main Function.

Main ()

Do initializations and determine start and end points of threads

for $i = 0$ to $THREAD\_NUM$ **do**

  Create a thread

  Bind the thread at a random core

end for

Thread_Code ()

for $i = 0$ to $NUMBER\_OF\_RUNS$ **do**

  Make calculations about application

  Lock the current core

  core_counter[current_core]++;

  if $core\_counter[current\_core] < CORE\_LIMIT$ **then**

    if $core\_epoch[current\_core] < EPOCH$ **then**

      core_epoch[current_core]++;

      Send the thread numbers of applications running on this core to the $g-cache$ module with $myMagic$ call

      Migrate random threads to random cores

    end if

  end if

  Unlock the current core

end for

Figure 3.12. Pseudo-code for Application Threads.

Main ()

counter $\Leftarrow 0$;

if *it catches a magic hap callback* **then**

Magic_Break();

end if

Magic_Break()

if $ebx == 1$ **then**

get_statistics_1;

end if

if $ebx == 3$ **then**

get_statistics_3;

end if

if $ebx == 2$ **then**

Parse the ebx register

Store the combinations coming from the application code

counter $++$;

if $counter == (CORENUM \times EPOCH)$ **then**

Call_Matlab()

$FirstFitIncreasing()$;

Send best 8 feasible combinations to the application code

end if

end if

Call_Matlab()

Run Matlab Engine and Send training data with *memcpy* commands

Place the variables in the Matlab Workspace with $engPutVariable$ commands

Run $4-dimensional\ polynomial\ fitting$ with $engEvalString$ commands

Receive predicted results from Matlab Workspace with $engGetVariable$ command

Figure 3.13. Pseudo-code for $G-cache$ Module.

4. EXPERIMENTAL SETUP AND RESULTS

The evaluation of our framework is done using the SIMICS system simulator. There are test results for both the static and the dynamic components. The test results and the SIMICS simulator are explained in more detail below.

4.1. Simulation Platform SIMICS

SIMICS [17] is a full system simulation platform from Virtutech [33]. It is a commercial product of Virtutech, but it is also freely available for academic use. To get it for academic use, you have to prove your academic identity and usage by simply registering on their website and providing some academic information about yourself. SIMICS supports the design, development, and testing of computer hardware and software. It achieves a balance between accuracy and performance; it is accurate enough to run commercial workloads and is fast enough to run real workloads, interactive desktop applications, and games. It can also model embedded systems, multiprocessor systems, clusters, and networks. The overview of SIMICS can be seen in Figure 4.1 [1].

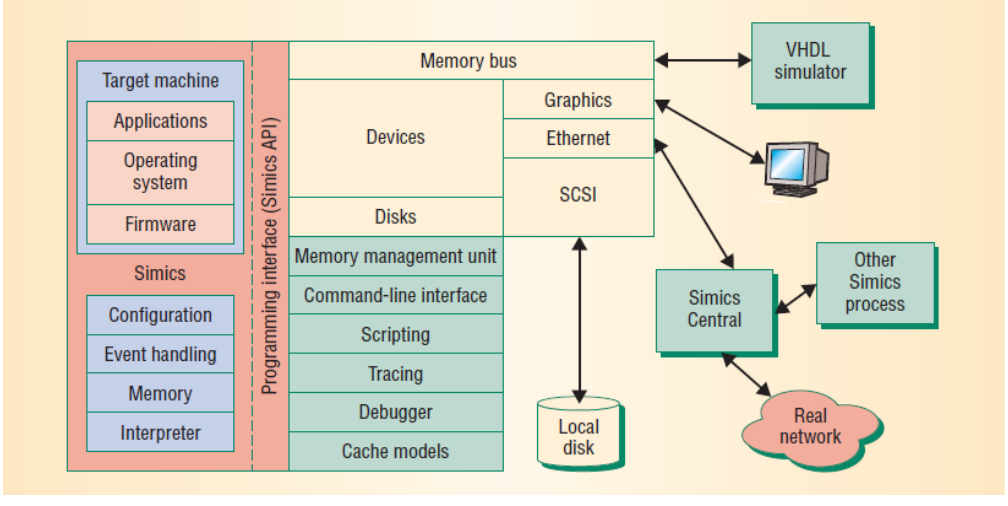

Figure 4.1. Overview of SIMICS Simulator [1].

SIMICS simulates processors at the instruction-set level, currently it supports models for UltraSparc, Alpha, x86, x86-64, PowerPC, IPF, MIPS, and ARM. The hosts SIMICS supports are Linux, Windows, and Solaris. Multiple instances of SIMICS can simulate different processor architectures running different operating systems on a single machine.

SIMICS is scalable and can support single-board, multi-board, multi-core, multi-processor, heterogeneous, distributed, networked, SoC (System-on-Chip) systems, and it uses DML (Device Modeling Language), Python, and C/C++ as the modeling languages for various architectures. SIMICS models targets with objects. Processors, devices, or virtual objects like disk images and memory mappings are all modeled by using objects.

There are several debug tools in SIMICS, and breakpoints can be added on a device access or read from memory components to monitor the system. The simulation can be run in reverse order, so that we can go over errors and see the problems occurred during the simulation.

Virtutech has a tool called SimGen that generates an interpreter for the instruction set of the target processor. To do this, first the high-level description of the instruction set should be written. Patterns for instruction decoding are given to Sim-Gen; it takes the description and generates the interpreter. SimGen can be used if a target processor, which has not been implemented in SIMICS, should be used.

SIMICS is used as our simulation platform, since our purpose is to design and implement multicore systems with different attributes. These attributes are varying core types and numbers, varying on-chip memory components and variable sharing of these memory components among cores.

SIMICS have been installed on a Fedora and Windows operating systems and some basic simulations have been executed on SIMICS. I have read about SIMICS

Table 4.1. Test Cases for Static Component.

Test Cases	Jacobian	Barnes Hut	SVM
Test 1	1	0	0
Test 2	1	1	0
....	....	....	....
Test 17	1	16	0
Test 18	1	0	1
Test 19	1	1	1
....	....	....	....
Test 289	1	16	16

and made some basic exercises on custom model development. I finished the tutorial included in the SIMICS's built-in documentation.

4.2. Static Component Tests

In the static component, for a given number of multi-threaded applications, one application is fixed as the base application and the effect of other applications on the performance of base application is observed. The base application and different thread combinations of other applications share the same cache structure. Threads of other applications disturb the base application's data on L2 cache.

At first attempt, three applications, Jacobian Matrix Calculation, Barnes-Hut method, and Sparse matrix-vector multiply (SpMV) applications are used as benchmark applications. Jacobian Matrix Calculation is fixed as the base application and runs on one thread. It remains as one thread because the effects of Barnes-Hut and SpMV threads are monitored. The degree of multi-threading of other two applications varies from 0-to-16. There are 289 different thread combinations. The test cases for three applications are listed in Table 4.1.

The target multicore machine is a 2-core machine, where each processor has its own private 32 KB L1 (data and instruction) caches and each of the two cores share on-chip 1 MB L2 cache, as we can see in Figure 3.1. Since we monitor L2 cache disturbance, even if we create a 2-core machine, both of them are connected to the same shared L2 cache. The initial thread mapping is left to the operating system because no matter the threads are assigned to the which core, all of them use the same shared L2 cache structure. The important characteristics of the simulated multicore and the simulated cache structure are given in Table 3.2.

All test cases are executed, and the execution time values are monitored for each case. Running 289 test cases takes two weeks in a simulator environment. It is expected that while the total number of threads increase, the execution time should also increases because of the contention on the L2 cache. The execution time graph of the test results is shown in Figure 4.2. As seen in the graph, the x axis represents the increasing number of threads of the Barnes-Hut application, the y axis represents the increasing number of threads of the SpMV application, and the z axis represents the execution time values of different combinations. The execution time is proportional to the increasing number of threads for both x and y axes. There are some inconsistencies in the graph like sharp downs, but this may be caused by the context switch of the Linux scheduler. It may give some priorities to the different threads, and their execution times may be shorter than the others. With this test, it is seen that, the threads of the base application is affected by other applications' threads in the meaning of cache disturbance. This effect can be seen in the execution time of overall program execution.

Selecting the best combinations and looking at these results is not difficult. However, it is not possible and practical for many more applications to try all possible combinations. Therefore, a prediction should be made about how application threads affect each other. This behavior can be characterized with a small number of combinations by using curve fitting techniques. According to this example, we have 289 results (17x17); 10% of them (nearly 30 points) can be selected and the behavior of the rests can be predicted. In this manner, the Surface Fitting Tool in MATLAB is used. An

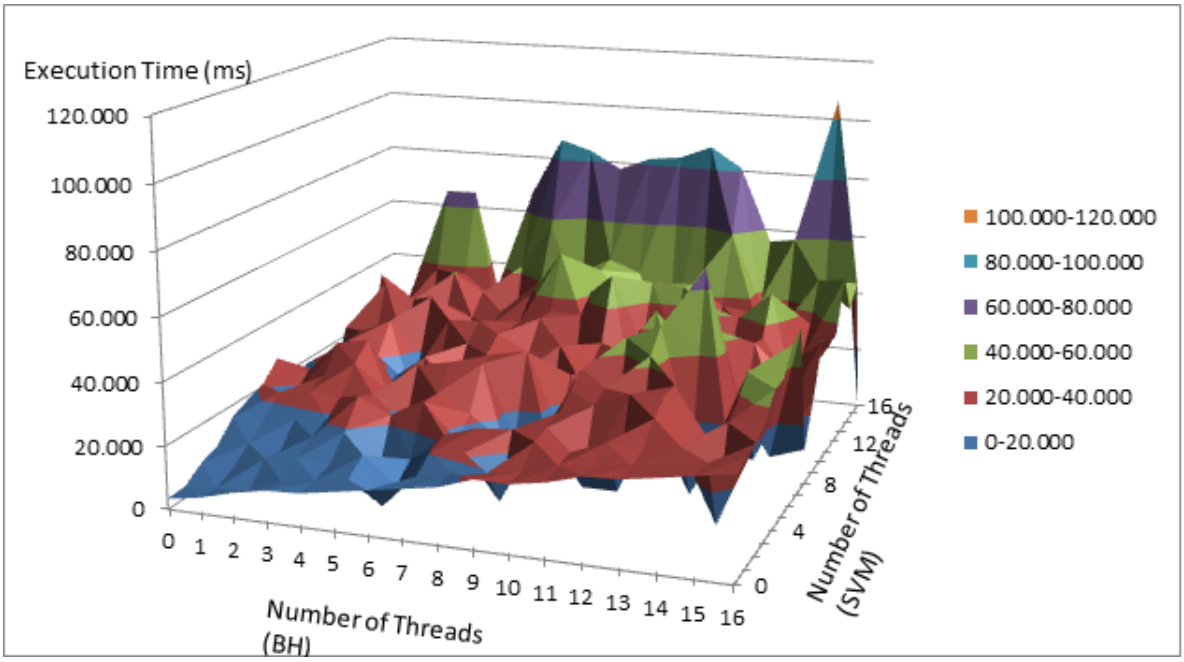

Figure 4.2. Static Version Test Results.

example is explained below.

- Example:

  In this example 10% data is selected in a random manner for the training data, and 90% data is predicted. The predicted values are compared with the actual validation data, which remains as the subtraction of training data from total data. Since there are 289 points in total, 30 points are selected randomly for training data, and 259 points are used for validation data. The target fitting type is listed Curve Fitting Model section in Static Component. The Cubic Interpolant fit type is selected because of the lower error rate compare to other fit types.

  Figure 4.3 shows the Surface Fit Result of this example. The x_test represents the increasing number of BarnesHut application threads, y_test represents the increasing number of SpMV application threads, z_test represents the execution time values for 10% test data, and z_v represents the the execution time values for the 90% validation data. In the graph, the points filled with blue are the test

Table 4.2. Goodness of Fit Statistics.

Goodness of validation: (for validation 90%)
SSE : 32023.8
RMSE : 12.9485
61 validation points outside domain of surface

data and the points blank inside are the validation data. Test data is fully on the curve, but some of the validation data is out of the domain.

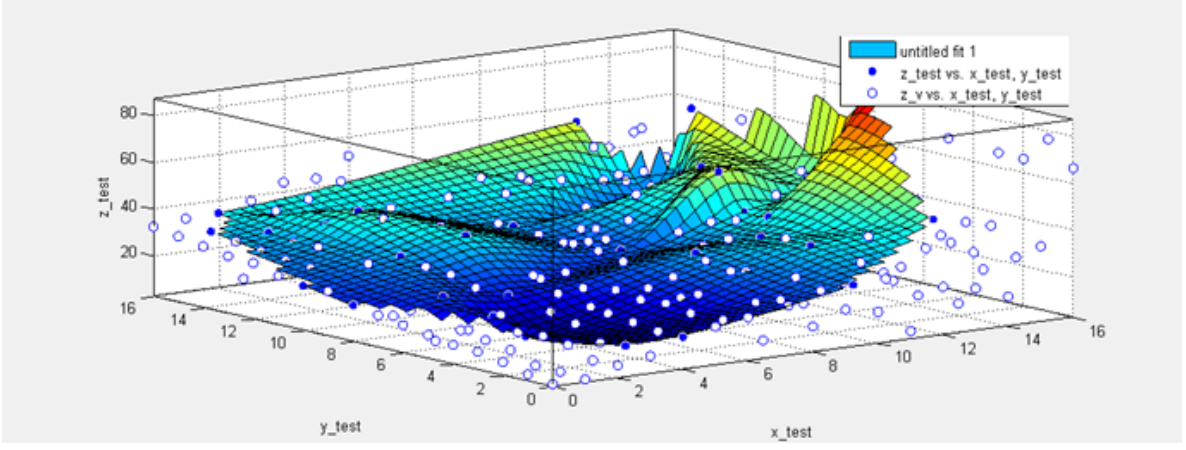

Figure 4.3. Surface Fit Results.

After the fitting of a curve with the given methods, MATLAB returns us Goodness of Fit (GoF) statistics that describes how well it fits a set of observations. The goodness of fit statistic can be seen in Table 4.2. The error rates in this table show how well the rest of data can be predicted with the given test data. The SSE value shows that total deviation for the validation is 32023.8. The SSE value should be closer to 0. The RMSE value that shows the standard deviation of the data is 12.9485. If its value is closer to 0, it indicates a fit that is more useful for prediction. Moreover, we can say that 61 out of 259 points of the validation data are not on the curve.

According to the static component test results, our discussions are as follows;

- The thread of the base application is affected by other applications' threads in the meaning of cache disturbance.
- This effect can be seen in the execution time of overall program execution.
- Applications can be executed based on the thread combinations determined.
- Another alternative is to provide a dynamic prediction-based thread scheduling framework as in the below.

4.3. Dynamic Component Tests

In the dynamic component, three applications, each having 32 threads run on an 8-core machine. which has the same number. In this case, there is no fixed base application. Jacobian Matrix Calculation, Black-Scholes application, and SpMV applications are used as benchmark applications. The degree of multi-threading of all applications is fixed as 32.

The target multicore machine is an 8-core machine, where each processor has its own private L1 (data and instruction) caches and each of the two cores share on-chip L2 cache, as we can see in Figure 3.3. Since L2 cache disturbance is monitored, even if an 8-core machine is created, each of the two cores are connected to the same shared L2 cache. The important characteristics of the simulated multicore and the simulated cache structure are given in Table 3.2

For performing our experiments, a multiprocessor simulation environment, the SIMICS tool-set is used. For Intel x86 Architecture, experiments are conducted on a Linux operating system using a gcc compiler. We have used pthreads implementations of our benchmark applications. The simulation parameters for the CMP architectures used is given in Table 4.3.

Different inputs are used for benchmark applications. Two selected inputs of

Table 4.3. Simulation Parameters for Performing Experiments.

$Parameter$	$Value$
$Numberofcores(n)$	8
$Numberofthreads(p)$	32
$CPUFrequency$	1 GHz
$L1Cache$	32 K/core
$L2Cache$	1 MB/2 cores

SpMV application from the University of Florida Sparse Matrix Collection [34] are considered for computational experiments. The properties of these inputs is shown in Figure 4.3 Two selected inputs, namely in_4K and in_64K, are used for Black-Scholes application. In the Jacobian application, a 500x500 size of matrix is used. The outer for loop size is fixed at 100 for all applications. Each experiment shows the best performance value for both prediction based mapping and random mapping.

Table 4.4. Application Input Matrices for SVM application.

Input Name	Number of rows	Number of columns	Number of nonzeros
Illc1033	1033	320	4732
bcsstk18	11948	11948	80519

In the first experiment, in_4K input is used for the Black-Scholes application and Illc1033 input is used for the SpMV application. Table 4.3 shows the cache statistics for the first experiment. The first column shows the Prediction Based Mapping which is our framework. The second column shows Random Mapping in which the initial thread mapping is done randomly and no optimization is done. The third column shows the Linux Scheduler in which the initial mapping is left to the operating system. The only difference is the initial thread mapping technique between the second and third methods. The first row shows the cache statistics for L1 cache, the second row shows

the cache statistics for L2 cache, and the third row shows the execution time values of each method. According to the test results, L1 and L2 cache statistics for second and third method are mostly the same. However, there is a sharp difference in the execution time values. This difference can be caused by two reasons. The first one is there are a lot of migrations in random mapping, 96 threads are bound to the cores with $pthread\_set\_affinity$ method. As mentioned, even if we migrate some threads to the cores with this command, this situation in the physical environment takes some time. This time can be seen in the execution time value of Random Mapping methodology. The second reason can be that, when the threads are bound to the cores with $pthread\_set\_affinity$ command, the Linux scheduler does not change the core of this thread. The thread stays at this core until the completion of the program. Normally, the Linux scheduler makes context switch among the threads and it can change the core of threads running on. Both of these reasons can influence the execution time value of Random Mapping methodology. We compare our Prediction-Based Mapping Framework with Random Mapping methodology to make a fair comparison since the initial thread mapping is also done randomly in our framework.

According to the test results, both the Random Mapping methodology and Prediction Based Mapping framework show similar performance trends for small inputs. Execution time values are nearly the same for both cases, since they suffer from migration cost. According to the L1 cache statistics, Prediction Based Mapping is nearly 20%. If we look at L1 cache miss rates, our algorithms is nearly 20% better than the others. However, these values say that the hit ratio of the L1 cache is very high. According to the L2 cache statistics, the miss rates are nearly same for all cases. According to the our contributions, it is expected to make improvements on L2 cache disturbance, but these values say that no optimization can be achieved in the meaning of L2 cache.

In the second experiment, the input sizes are increased and in_64K the input is used for the Black-Scholes application, bcsstk18 input is used for the SpMV application. We compare our Prediction-Based Mapping Framework with Random Mapping methodology to make a fair comparison since the initial thread mapping is also done

Table 4.5. L1 and L2 Cache Counts (in thousands) and Execution Time (in seconds) for in_4K and illc1033 inputs.

		Prediction Based Mapping		**Random Mapping**	**Linux Scheduler**
Cache Type	*Statistic Name*	After training	End	End	End
L1 Cache	Total Miss Difference #	223.342	312.122	311.135	311.126
	Total Access Difference #	3.655.994	4.841.180	3.791.047	3.786.851
	Miss Rate (%)		6.45	8.21	8.22
L2 Cache	Total Access Difference #	550.374	824.887	821.542	268.125
	Total Miss Difference #	190.620	266.113	267.844	268.125
	Miss Rate (%)		32.26	32.60	32.73
	Execution Time		4.98	4.90	1.03

randomly in our framework. According to the test results in Table 4.3, the Random Mapping methodology and the Prediction-Based Mapping framework show little different performance trends for large inputs. The execution time value of Prediction Based Mapping framework is 20% worse than for the Random Mapping. According to the L1 cache statistics, Prediction-Based Mapping is nearly 13,5% better than the Random Mapping. According to the L2 cache statistics, the miss rates are nearly same for both cases. With the evaluation of the second experiment, no optimization can be achieved in the meaning of L2 cache statistics.

In the third experiment the cache sizes are decreased and the in_4K input is used for the Black-Scholes application, the illc1033 input is used for the SpMV application. Dedicated L1 cache size is reduced to 8Kb for each core and shared L2 cache size is reduced to 256Kb for each of the two cores. We again compare our Prediction-Based Mapping Framework with Random Mapping methodology. According to the test results in Table 4.3, the Random Mapping methodology and the Prediction-Based Mapping framework show different performance trends for reduced cache sizes. The execution

Table 4.6. L1 and L2 Cache Counts (in thousands) and Execution Time (in seconds) for in_64K and bcsstk18 inputs.

		Prediction Based Mapping		**Random Mapping**
Cache Type	*Statistic Name*	After training	End	End
L1 Cache	Total Miss Difference #	18.119	345.270	344.939
	Total Access Difference #	1.138.340	7.324.389	6.331.472
	Miss Rate (%)		4.71	5.45
L2 Cache	Total Access Difference #	48.855	1.759.458	1.754.831
	Total Miss Difference #	8.160	274.068	280.827
	Miss Rate (%)		15.58	16.00
	Execution Time		6.73	5.36

time value of Prediction-Based Mapping framework is approximately 2.86 times worse than for the Random Mapping. According to the L1 cache statistics, Prediction-Based Mapping is nearly 20% better than the Random Mapping. According to the L2 cache statistics, Prediction-Based Mapping is nearly 6% worse than the Random Mapping. With the evaluation of the third experiment, no optimization can be achieved in the meaning of L2 cache statistics.

The first reason for this could be the connection style of the L2 caches, which is connected for each of the two cores. The prediction model sends us the best eight feasible combinations. When one of the combinations is assigned to a selected core, the threads of the other core which is connected to the same L2 cache structure, continues to disturb the threads of the first selected core. Therefore, each of the two cores can be assumed as a cluster and the prediction results shows the thread combinations of clusters rather than the cores. This change can improve the L2 cache statistics.

Table 4.7. L1 and L2 Cache Counts (in thousands) and Execution Time (in seconds) for 8Kb/core L1 Cache and 256Kb/2 cores L2 Cache.

		Prediction Based Mapping		**Random Mapping**
Cache Type	*Statistic Name*	After training	End	End
L1 Cache	Total Miss Difference #	1.442	316.041	314.376
	Total Access Difference #	973.005	4.755.879	3.786.151
	Miss Rate (%)		6.64	8.30
L2 Cache	Total Access Difference #	5.580	827.200	821.903
	Total Miss Difference #	334	303.266	283.313
	Miss Rate (%)		36.66	34.47
	Execution Time		6.41	2.24

The second reason for these results could be the size of L1 and L2 caches. As mentioned before, the L1 cache hit ratio is very high. This means each thread can access all its data in the first level cache. However, the hit ratio of the L2 cache is not very high and this value can be optimized.

SIMICS does not support simultaneous multi-threading, so each time a context switch occurs, the L1 cache is flushed; therefore, only the data that resides in L2 cache is shared among threads. If each core has its private L2 cache, then this cache is used by only the threads assigned to that core. But if L2 cache is shared by a number of cores, then the data is also shared by all the threads assigned to those cores. If the final mapping is done in this way, the overhead of thread migrations can be decreased. There may be unnecessary thread migrations between the cores that shares the same L2 cache. This would affect the performance of our approach. Apart from that, the final mapping in the core level rather than in the cluster of cores level which shares the same L2 cache. If the connection types of L2 cache structures can be changed or the

final mapping is done in cluster of cores level, L2 cache statistics can be optimized.

4.3.1. Overheads of Our Framework

The most time-consuming overhead of our framework is the thread migration. Since the thread migrations are done in initial mapping, at the end of each epoch time, and in final mapping, our framework greatly suffers from this overhead. As mentioned, even if we migrate some threads to the cores with this command, the actual happening of this situation in physical environment takes some time. In order to decrease the cost of this overhead, we should not migrate threads unless it is necessary. Because of the cost of the migrations, only limited number of threads can be migrated to the other cores at the end of each epoch. This approach does not allow having diverse combinations in the training data set. In order to get rid of migration cost in final mapping phase, a migration-aware mapping strategy is applied. With this approach, we do not migrate threads unless it is necessary. For each core, if there are threads on it, the necessary number of threads are added or removed according to the prediction results.

The second overhead of our framework is the communication between the modules. Our application code is in communication between the g-cache module, and g-cache module is in communication between the curve fitting model. The synchronizations between these modules affect the performance of our framework. The presented results include all the overheads incurred by our framework.

5. CONCLUSIONS AND FUTURE WORK

The focus of this thesis is to find the ways of mapping and dynamically adapting the execution of the multiple multi-threaded applications on CMPs. To do this, a new approach is proposed, which is prediction-based thread scheduling for multicore systems. This approach, analyzes thread behavior of different applications on shared cache, predicts the all possible thread combinations of different applications, and tries to find the best thread combinations of different applications that result with minimum cache disturbance. To achieve that, our framework has two components. The first one is the static component in which the training phase of prediction is done off-line, and the second one is the dynamic component in which the training phase of prediction is done on-line. Both of them use a curve fitting based strategy to determine, for each application, the ideal number of threads running on the same core. In the static component, for a given number of multi-threaded applications, one application is fixed as the base application and it is observed how threads of other applications affect the performance of the base application. The base application and different combinations of other applications' threads share the same cache structure. Threads of other applications disturb the base application's data on L2 cache. This disturbance affects the execution time of the program. With the evaluation of static component, the thread of the base application is affected by other applications' threads in the meaning of shared cache disturbance. This effect can be seen in the execution time of overall program execution. Surface Fitting Toolbox of MATLAB is used to do 3-dimensional curve fitting. Applications can be executed based on the thread combinations determined by MATLAB. Another alternative is to provide a dynamic prediction based thread scheduling framework.

Same tasks of the static component is performed in an automatic way in the dynamic component. Collecting the training data, communication with curve fitting model in MATLAB, receiving predicted results, and determining final mapping strategy according to these values are done at runtime. Any interference with the program is

avoided in any step of the execution. Therefore, dynamic component is more complicated and difficult to implement but, is more practical to use. In dynamic component, for a given number of multi-threaded applications; in this case, there is no fixed base application. The degree of multi-threading is fixed for all applications. In the execution of the program, each core is monitored during periodic time intervals and training data is collected for curve fitting model. After sending training data and receiving the predicted data from curve fitting model, the final thread mapping strategy is determined. In the evaluation of dynamic component, we compare our Prediction-Based Mapping Framework with Random Mapping methodology to make a fair comparison since the initial thread mapping is also done randomly in our framework. According to the experiment results, the performance values are nearly the same for both cases. According to the L1 cache statistics, Prediction Based Mapping is nearly 20% better than Random Mapping. According to the L2 cache statistics, the miss rates are nearly same for both cases. With the evaluation of the experiments, no optimization can be achieved in the meaning of L2 cache statistics so far. On the other hand, there are overheads like thread migration costs and communication costs between the modules, this affects the performance of our framework.

We target to make improvements on L2 cache rather than L1 cache. To achieve that, additional tests will be done with changing some parameters. Current architecture and the sizes and the connection types of cache levels can be changed. Training phase time can be increased to get more diverse combinations to have more accurate prediction results, or it can be decreased to get efficiency on cache level after training phase. The degree of polynomial can be changed that is used in curve fitting model. Instead of doing initial mapping randomly in the current approach, a more intelligent one can be selected and this affects the prediction results. Instead of doing final mapping in the core level, it would be done in the cluster of cores level that share the same L2 cache.

As a future work, instead of using execution time as a performance metric, we can use Instruction per Cycle (IPC) value. In order to get the IPC value from the SIMICS simulator, General Execution-driven Multiprocessor Simulator (GEMS) module will be

used [35].

Running our methodology in a simulation environment sometime takes five days. For that reason, we could not do all these tests in the scope of this thesis. However, additional tests will be done until improvements are made on the L2 cache level. This experiment's results encourage us to achieve this by doing more experiments.

REFERENCES

1. Magnusson, P., M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner, "Simics: A full system simulation platform", *Computer*, Vol. 35, pp. 50–58, 2002.

2. Geer, D., "Chip Makers Turn to Multicore Processors", *IEEE Computer*, 2005.

3. Hammond, L., B. Hubbert, M. Siu, M. Prabhu, M. Chen and K. Olukotun, "The Standford Hydra CMP", *IEEE Micro*, Vol. 20, pp. 71–84, 2000.

4. Kumar, R., D. M. Tullsen, N. P. Jouppi and P. Ranganathan, "Heterogeneous Chip Multiprocessors", *IEEE Computer*, Vol. 38, pp. 32–38, 2005.

5. Jiang, Y. and X. Shen, "Exploration of the Influence of Program Inputs on CMP Co-Scheduling", *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, 2008.

6. Borkar, S., "Thousand core chips: a technology perspective", *In Proc. of DAC*, 2007.

7. Bhadauria, M. and S. A. McKee, "An Approach to Resource-Aware Co-Scheduling for CMPs", *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010.

8. Ding, Y., M. Kandemir, M. J. Irvin and P. Raghavan, "Dynamic Core Partitioning for Energy Efficiency", *Proceedings of the 6th Workshop on High-Performance, Power-Aware Computing (HPPAC), in conjunction with 24nd IEEE/ACM International Parallel and Distributed Symposium, IPDPS-2010*, 2010.

9. Rai, J. K., A. Negi, R. Wankar and K. Nayak, "On Prediction Accuracy of Machine Learning Algorithms for Characterizing Shared L2 Cache Behaviour of Programs

on Multicore Processors", *Proceeding CICSYN '09 Proceedings of the 2009 First International Conference on Computational Intelligence, Communication Systems and Networks.*, 2009.

10. Morad, T., A. Kolodny and U. C. Weiser, "Scheduling Multiple Multithreaded Applications on Asymmetric and Symmetric Cip Multiprocessors", *International Symposium on Parallel Architectures, Algorithms and Programming*, 2010.

11. Jiang, Y., K. Tian and X. Shen, "Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors", *High Performance Embedded Architectures and Compilers - HIPEAC*, pp. 201–215, 2010.

12. Fedorova, A., M. Seltzer and M. D. Smith, "Cache-Fair Thread Scheduling for Multicore Processors", *Technical Report TR-17-06, Division of Engineering and Applied Sciences, Harvard University*, 2006.

13. Chen, S., P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry and C. Wilkerson, "Scheduling Threads for Constructive Cache Sharing on CMPs", *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, 2006.

14. Winter, J. A., D. H. Albonesi and C. A. Shoemaker, "Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures", *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.

15. Tam, D., R. Azimi and M. Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors", *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

16. Song, F., S. Moore and J. Dongarra, "Analytical Modeling and Optimization for Affinity Based Thread Scheduling on Multicore Systems", *Cluster Computing and Workshops. CLUSTER '09. IEEE International Conference on*, 2009.

17. Sibai, F. N., "Simulation and Performance Analysis of Multi-core Thread Scheduling and Migration Algorithms", *Complex, Intelligent and Software Intensive Systems (CISIS), International Conference on*, 2010.

18. Srikantaiah, S., R. Das, A. K. Mishra, C. R. Das and M. Kandemir, "A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors", *In Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SuperComputing), Portland*, 2009.

19. Iverson, M. A., F. Özgüner and L. C. Potter, "Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment", *IEEE Transactions on Computers*, Vol. 48, 1999.

20. "K-Nearest Neighbor Algorithm", 2011, `http://en.wikipedia.org/wiki/K_nearest_neighbors`.

21. Wernsing, J. R. and G. Stitt, "A Scalable Performance Prediction Heuristic for Implementation Planning on Heterogeneous Systems", *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, 2010.

22. Delgado, J., M. Sadjaji, M. Bright and H. A. Duran-Limon, "Performance Prediction of Weather Forecasting Software on Multicore Systems", *PDSEC-10: The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2010.

23. O'Hagan, A., "Curve Fitting and Optimal Design for Prediction", *Journal of the Royal Statistical Society. Series B (Methodological)*, Vol. 40, pp. 1–42, 1978.

24. "MATLAB R2010b Documentation", 2010, `http://www.mathworks.com/help/toolbox/curvefit/bqswfbh-1.html`.

25. "Interpolation", 2011, `http://en.wikipedia.org/wiki/Interpolation`.

26. "Polynomial", 2011, http://en.wikipedia.org/wiki/Polynomial.

27. "Explained Sum of Squares", 2011, http://en.wikipedia.org/wiki/Explained_sum_of_squares.

28. "Parsec Benchmark Suite", 2008, http://parsec.cs.princeton.edu/.

29. Black, S., Fischer, "The pricing of Options and Corporate Liabilities", *Journal of Political Economy*, Vol. 81, pp. 637–659, 1973.

30. Fedorova, A., "Introduction to Cache Simulation with Simics", 2007, http://www.cs.sfu.ca/fedorova/Tech/simics-guides-3.0.26/simics-user-guide-unix/topic87.html/.

31. "Bin Packing Problem", 2011, http://en.wikipedia.org/wiki/Bin_packing_problem.

32. Boyar, J., L. Epstein, L. M. Favrholdt, J. S. Kohrt, K. S. Larsen, M. M. Pedersen and S. Wohlk, "The Maximum Resource Bin Packing Problem", *Journal Theoretical Computer Science*, Vol. 362, 2006.

33. "Vrtutech Simics 4.0.", 2008, http://www.virtutech.com/.

34. Davis, T., "The University of Florida Sparse Matrix Collection", 2010, http://www.cise.ufl.edu/research/sparse/matrices/.

35. "Wisconsin Multifacet GEMS Simulator", 2011, http://www.cs.wisc.edu/gems/.