The Pennsylvania State University
The Graduate School

REUSE DISTANCE MODELS FOR ACCELERATING SCIENTIFIC COMPUTING WORKLOADS ON MULTICORE PROCESSORS

A Dissertation in
Computer Science and Engineering
by
Jeonghyung Park

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2015

The dissertation of Jeonghyung Park was reviewed and approved* by the following:

Padma Raghavan
Associate Vice President for Research
Director of Institute for CyberScience
Distinguished Professor, Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Mahmut Kandemir
Professor, Computer Science and Engineering

Kamesh Madduri
Assistant Professor, Computer Science and Engineering

Christopher Duffy
Professor, Civil and Environmental Engineering

Raj Acharya
Professor, Computer Science and Engineering
Department Head, Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

As the number of cores increase in chip multiprocessor microarchitecture (CMP) or multicores, we often observe performance degradation due to complex memory behavior on such systems. To mitigate such inefficiencies, we develop schemes that can be used to characterize and improve the memory behavior of a multicore node for scientific computing applications that require high performance.

We leverage the fact that such scientific computing applications often comprise code blocks that are repeated, leading to certain periodic properties. We conjecture that their periodic properties and their observable impacts on cache performance can be characterized in sufficient detail by simple '$\alpha + \beta \sin(\theta)$' models. Additionally, starting from such a model of the observable reuse distances, we develop a predictive cache miss model, followed by appropriate extensions for predictive capability in the presence of interference.

We consider the utilization of our reuse distance and cache miss models for accelerating scientific workloads on multicore system. We use our cache miss model to determine a set of preferred applications to be co-scheduled with a given application to minimize performance degradation from interference. Further, we propose a reuse distance reducing ordering that improves the performance of Laplacian mesh smoothing. We reorder mesh vertices based on the initial quality for each node and its neighboring nodes so that we can improve both temporal and spatial localities. The reordering results show that 38.75% of performance improvement of Laplacian mesh smoothing can be obtained by our reuse distance reducing ordering when running on a single core. 75x of speedup is obtained when scaling up to 32 cores.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work was supported in part by the National Science Foundation through awards 1319448 and 1439058.

Dedication

I would like to thank my thesis advisor, Dr. Padma Raghavan, for her guidance, patience, and support she has shown me during my time here at Pennsylvania State University. I am grateful for the one-on-one time she has given and our conversations that have expanded my mind and opened to the gate for overcoming problems I met in my methodology. Without her support, I would have never made it. I thank my committee members, Dr. Mahmut Kandemir, Dr. Kamesh Madduri, and Dr. Christopher Duffy, for their insightful commentary on my work. Finally, I would like to thank my father and my mother. My father was strong role model that taught me the joy of research in computer science. My mother taught me the value of hard work. I am grateful for their indefinite support over the years.

Chapter 1

Introduction

Current trends call for core counts as high as 60$\sim$90 at a multicore node in the near future [2]. Consequently, it is desirable to apply greater parallelism to enable efficient utilization of processing power and memory resources [3,4]. However, the performance of irregular high performance computing applications (HPC) on such multicores tend to scale poorly. We observe that many irregular applications cause unexpected memory accesses and these become the bottleneck of performance scalability [5]. Further, when multiple applications are run concurrently on larger number of cores, they compete for shared resources such as on-chip caches, main memory, memory bandwidth, and the I/O bus. Such computing resource contention often have negative impacts on overall performance of applications. This degradation becomes even more pronounced in applications with higher performance computing requirements. To mitigate unexpected performance degradation for irregular high performance computing applications on multicores, understanding memory subsystem behavior is crucial. More specifically, efficient cache utilization for irregular applications must be considered in order to achieve performance scalability under a fixed problem size.

For resolving significant performance issues and tuning application's performance, we focus on memory behavior of on-chip caches and NUMA domains for shared memory, many-core systems. In current HPC systems, there is considerable variation between vendors, for instance, Intel and AMD chip designs range from 4 to 12 cores per die and have significant differences in cache design [6,7]. This variation makes it challenging to predict application's performance on these various computer architecture systems without costly runs of applications. Further, for a performance prediction to be feasible it is required to be compact and efficient. Thus, we seek mathematical model driven approach to predict memory behavior of applications.

In this thesis, we consider the methodology for modeling memory behavior of HPC applications by using reuse distances. It is known that there exists a relationship between reuse distance and cache miss rates for applications on multicore systems [8]. Applications with lower reuse distances show better performance. Figure 1.1 shows the performance variation in regards of reuse distances. Once we develop the reuse distance models, we use it to optimize performance.

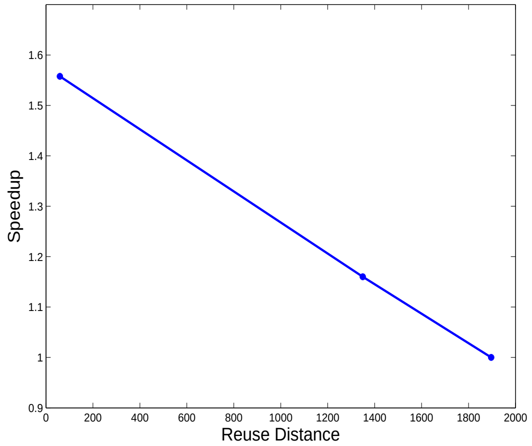
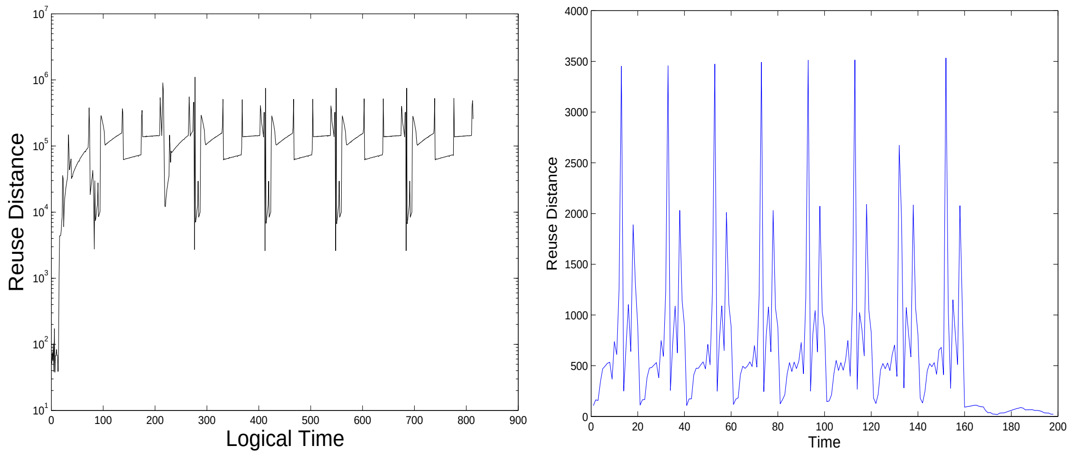


Figure 1.1: Speedups obtained from difference reuse distances for irregular mesh smoothing application. As the reuse distances increased, less performance gainings are achieved.

To mathematically model the memory behavior of HPC applications through reuse distances, we use the fact that many scientific applications show periodic behavior. Our main contributions are based on exploiting the fact that HPC applications often exhibit periodic behavior. This is primarily from the fact that in many HPC codes, code segments in basic blocks are performed repeatedly. We expect that their iterative computations will lead to certain periodic patterns in measures such as reuse distances and cache misses that we can model compactly for predictive purposes. As an example, the observed reuse distance profile is shown below in Figure 1.2(a) for parallel multigrid (MG). Mesh smoothing application considered in Chapter 5 also shows the periodic behavior in its observed reuse distance profile shown in Figure 1.2(b). We would like to consider how the behavior of this and other applications can be modeled as a simple periodic function.



(a) parallel multigrid

(b) Mesh optimization

Figure 1.2: The reuse distance profile for the multigid (MG) application (a). A certain periodic behavior is observable. The reuse distance profile for the mesh optimization application discussed in Chapter 6 also shows a periodic pattern (b).

Our contributions in this thesis are not meant to faithfully reproduce the complicated time evolution patterns that can be observed of reuse distances and cache misses. Instead, our goal is to come up with concise, reduced-order models which can capture essential features of memory behavior including average reuse distances and average cache misses. We use the periodic patterns in their observable reuse distances to develop models for their memory behavior in terms of cache miss rates. Additionally, we develop a predictive cache miss model based on our conjecture that these periodic patterns will continue to persist even in the presence of interference in the shared caches when two applications are co-scheduled. Further, we will improve the performance of HPC application by using our predictive reuse distance and cache miss models.

We start by providing notation, background material on simple periodic functions and Laplacian mesh smoothing, related research on reuse distance and cache interference miss analysis, and mesh reordering schemes in Chapter 2. We provide our main contributions in Chapters 3, 4, and 5. Our main contributions in this thesis can be summarized as follows:

- **Reuse Distance and Cache Miss Models.** In Chapter 3, we propose a simple $\alpha + \beta \sin(2\pi t\phi)$ model of reuse distance that can be obtained from observed values. We assume that a small set of data can be obtained to correlate, in general, reuse distance and cache misses for a given multicore. We propose how to develop a model for cache misses from the reuse distance model when an application runs by itself without interference.

- **Predictive Cache Miss Models in the Presence of Interference.** In Chapter 4, we propose models of cache misses under interference for any pair of applications. In our approach, all data needed to compute correlations can be obtained using synthetic code segments and without running actual applications. Actual runs are only needed to observe reuse distances for an application when it is running alone, thus, enhancing the potential value of our predictive model for informing co-scheduling. We verify our models for NAS benchmarks and we show that the error rate in our predictive models for cache misses when two applications interfere each other is on average 3.13%.

- **Accelerating Scientific Workload Using Reuse Distance and Cache Miss Models.** In Chapter 5, we show that how to utilize our reuse distance and cache miss models for improving performance. We first show that the predictive cache model can be used to generate rankings of a preferred partner for a given application to inform how application pairs could be co-scheduled for best performance. We then propose reuse distance reducing ordering for Laplacian mesh smoothing. This reordering scheme provides up to 38.75% of performance improvement for Laplacian mesh smoothing on a single core. When we scale up to 32 cores, 75x speedup with reuse distance ordering is obtained as compared with a serial baseline (with original ordering mesh). Predictive cache miss models developed in Chapter 3 were utilized to predict cache performance for Laplacian mesh smoothing when the reuse distance reducing ordering is applied. Prediction error rates for our predictive cache miss models were low at 5.36% on average.

Finally, we present concluding remarks and potential directions for future research in Chapter 6.

Chapter 2

Background

In this chapter, we present a brief review of the background material. We first introduce and define terms and notation that we will use in the remainder of this dissertation. Further, we provide a brief review of sine functions constructed by an Fast Fourier Transformation (FFT). Additionally, we present a review of related research about reuse distance analysis and characterization of cache interference. We then, provide a brief explanation of Laplacian mesh smoothing that is used for mesh quality improvement in irregular HPC applications. Finally, we discuss related works for mesh reordering to improve the mesh smoothing performance.

2.1 Notations

We consider a set of n applications $A_1, A_2, ...A_n$, each of which may execute on its own on a multicore processor or execute with any one of the other applications, and thus as a pair of applications sharing and contending for memory resources. We assume that for an application A_i when it is running on a multicore, we can observe its memory accesses to calculate the reuse distance and cache miss rate for each access as they evolve over time. The reuse distance is defined as the number of memory references between two successive accesses to the same memory reference. The cache miss rate is defined as the number of misses that occur in a shared cache divided by the total number of cache accesses.

We use OR_{A_i}, and OC_{A_i} to denote these *observed reuse distance* and cache miss profiles, respectively, for the case when *the application is running by itself*, i.e., without another co-scheduled application. Further, we expect to observe such sequences when A_i and A_j are co-scheduled together. In the co-scheduled case, because of interference, we expect the profiles to be different for each application compared to when it is run on its own. Further, we expect both applications to be impacted in different ways. To denote the *observed reuse distances for application A_i because of interference by A_j*, we use $OR_{(A_i|A_j)}$. Similarly, the *observed cache misses for the interfered application A_i* is denoted $OC_{(A_i|A_j)}$.

We will be developing modeled approximate representations of some of the observable measures as part of our main contributions in Chapter 3 and Chapter 4. We use R_{A_i} to denote the sine model for representing the periodic behavior in OR_{A_i}. We use symbols with an overline, $\overline{C}_{A_i}$ to denote the *model for predicting cache misses from R_{A_i}*, the model of observed reuse distances. Similarly, the *modeled reuse distance for A_i* in the presence of interference by co-scheduled application A_j is denoted by $\overline{R}_{(A_i|A_j)}$. The *modeled cache miss rate for A_i* in the presence of interference by co-scheduled application A_j is denoted by $\overline{C}_{(A_i|A_j)}$.

2.2 Simple Periodic $\alpha + \beta\sin(2\pi t\phi)$ Functions and the Role of FFTs

It is known that time-domain data that exhibit periodic behavior can be represented as a simple sine function in the form of $\alpha + \beta\sin(2\pi t\phi)$. If an observed sequence of a measure, such as the reuse distance, is known for example, through the trace of an application running on a multicore or from compiler or runtime analysis, then it is conceivable that if it has periodic behavior, it could be approximated by such a sine representation. In general, the process of determining the existence of periodicity in such data involves a suitable fast fourier transformation (FFT) [9].

Consider data with 100 elements in Figure 2.1(a) where a simple, regular, periodic behavior is clearly observable. Now an FFT can be applied to these data, to get an frequency-domain representation in terms of dominant magnitudes and their frequencies as shown in Figure 2.1(b). Observe that the two highest magnitudes are $M_1 = 2$ and $M_2 = 0.5$; further, they occur at frequencies $I_1 =$

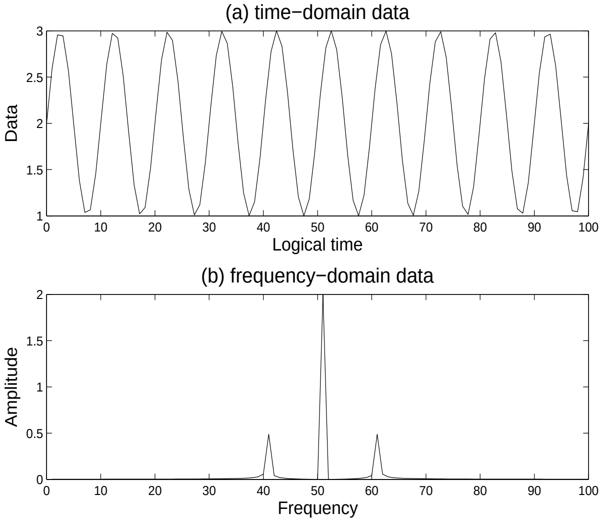


Figure 2.1: Time variance data trace for 100 elements generated by a function $y = 2 + sin(2\pi t \times 10)$ and its frequency domain data converted by FFT.

50, and $I_2 = 60$ respectively. One way construct a sine function of the form $\alpha + \beta \sin(2\pi t\phi)$ is to set $\alpha = M_1$, $\beta = M_2 \times 2$, and $\phi = abs(I_1 - I_2)$ to get $2 + 1 \times \sin(2\pi t \times 10)$. In this simple example, we recover the original form with accuracy. However, in general such a construction will give only an approximation. Further, the original data may be noisy and may have linear or non-linear trends that may need to be identified and removed before applying an FFT to characterize the periodic behavior. However, this basic approach is often appropriate with suitable variations to derive periodic function models from data in a wide variety of applications.

2.3 Reuse Distance Analysis for Characterizing Cache Interference

We now summarize how our results are loosely related to two prior streams of results on (i) reuse distance analysis and (ii) characterization of contention in the memory subsystem of CMPs/multicores when applications are co-scheduled. Further, we comment on how our contributions are significantly different than these earlier results.

Many researchers have applied reuse distance analysis to capture the cache performance of applications running on a single core system [10–12], and on multicore systems [13–16]. These results typically involve generating reuse distance histograms and using them to predict cache miss rates. However, these results do not take into account interference from co-scheduling of applications as we do in this paper.

More recently, as multiple applications are run simultaneously on CMPs/multicores, there is increasing interests in studying performance degradation effects due to shared resource contention. Zhuravlev et al. classify levels of cache contention in regard to increases in execution times [17]. Tang et al. used cache misses, memory bandwidth and prefetchers to indicate an application's contention characteristics [18] with further extensions in [15, 19, 20]. Additionally, Zhao et al. propose a two-phase regression approach involving cache misses and memory bandwidth [21]. These characterization results differ substantively from the results in this paper which provide closed form trigonometric function models of cache miss rates with interference from co-scheduling. Further, our models derive solely from the observable reuse distance profile of an application running alone.

The differentiating significance of our contributions in this paper is that we bring predictive capability through modeling for the specific class of scientific applications that are iterative, and thus, with periodicity in their memory access patterns. We observe that some others have also considered utilizing periodic behavior, although for very different applications such as identifying basic blocks [22] and reuse signatures [23].

2.4 Laplacian Mesh Smoothing

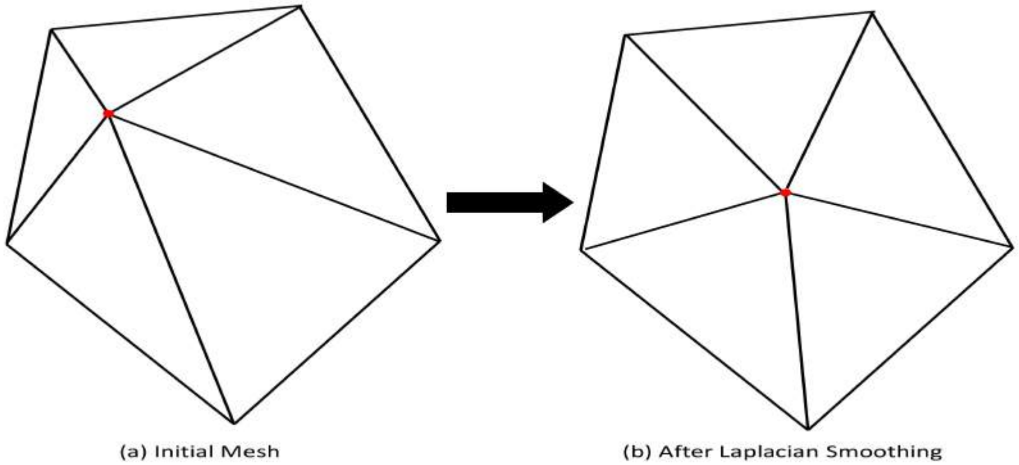
Mesh smoothing application is performed to improve the quality of the mesh so that accurate PDE solution can be obtained within a short execution time [24]. In mesh smoothing procedure, we first compute the initial mesh quality for a given mesh and do mesh smoothing to improve the mesh quality [25]. After smoothing, we compute the mesh quality again and if the overall mesh quality reached a desirable level, then we stop smoothing.

We use edge-length ratio [26], i.e., the ratio of minimum and maximum length edges, as a mesh quality metric for computing the mesh quality in this study. The mesh quality for each vertex can be represented as an average quality metric value of triangles that are attached on the vertex. The mesh quality for entire region of the mesh can be computed by averaging all mesh quality values obtained from each vertex. The range of edge-length ratio mesh quality values is $0 \sim 1$. If the quality value for a triangle is closed to 1, we can say the triangle has a good shape, i.e., closed to equilateral triangle that we desire. The goal of mesh smoothing application will be maximizing the average quality values for each vertex.

For improving the quality of the mesh, we perform Laplacian smoothing to replace a vertex position using neighbored vertices coordinates. Suppose there is a vertex v we want to move and N neighbored vertices surrounding the vertex. If we represent the position for i^{th} neighbored vertex as p_i, the new position for $\overline{p}_v$ will be

$$\overline{p}_v = \frac{1}{N} \sum_{i=1}^{N} p_i.$$

Figure 2.2 shows the initial mesh and the output mesh of Laplacian smoothing.



(a) Initial Mesh

(b) After Laplacian Smoothing

Figure 2.2: Laplacian smoothing performed on initial mesh. The vertex position inside the mesh was changed.

We would like to improve the performance of the Laplacian mesh smoothing by reordering the initial mesh. Each vertex will be reordered based on the initial mesh quality for each vertex and details will be described in Chapter 6.

Algorithm 1 *Algorithm for Laplacian Mesh Smoothing*

```
procedure Laplacian Smoothing($V$, $T$)
    $V \leftarrow$ mesh vertex data
    $T \leftarrow$ mesh triangle data
    quality = 0
    for $i \leftarrow 1, V$ do
        compute initial mesh quality for V[i]
        quality = quality + $q_{V[i]}$
    end for
    Initial quality = $\frac{1}{V}$quality
    while Final quality < goal quality do
        for $i \leftarrow 1, V$ do
            Laplacian Smoothing
        end for
        quality = 0
        for $i \leftarrow 1, V$ do
            compute final mesh quality for V[i]
            quality = quality + $q_{V[i]}$
        end for
        Final quality = $\frac{1}{V}$quality
    end while
end procedure
```

2.5 Mesh Reordering for Improving Mesh Smoothing Performance

Several researchers suggested mesh reordering scheme for improving mesh smoothing application performance. For example, Strout [27] and Hovland et al. [28] developed a Feasible Newton mesh optimization algorithm and benchmark. Their algorithm and benchmark employed both data and iteration reordering in order to improve cache performance. One finding of their research was that reordering of the input data can increase or decrease the number of iterations taken by the inexact Newton method and can affect its success or failure [29]. The reordering applied was a reordering of the vertices and elements in the mesh by applying a breadth-first search and reversing the order in which the vertices were visited. When data and iteration ordering were performed on the relevant hypergraphs, the reorderings were found to significantly decrease the number of cache misses in all phases of code execution and resulted in significantly faster code [28]. However, less amount of reuse distance reducing were observed with their reordering schemes compared to our proposed reordering scheme. Experimental evaluation will show these results in Chapter 6. Shontz and Knupp [30] considered mesh vertex reordering techniques to reduce the total time required to improve the mesh quality. Vertex ordering was performed for the first iteration (static) and every iterations (dynamic). To reduce extra reordering time for dynamic cases, Park et al. [31] considered a priori vertex reordering which performs the vertex reordering only once when mesh smoothing is started. Unlike this study, we consider cache performance when a reordering technique is applied to mesh smoothing. We can gain the performance improvement since we reduce the reuse distances for mesh smoothing applications and the reduced reuse distance can improve cache performance [32].

Chapter 3

Reuse Distance and Cache Miss Models

In this chapter, we develop a methodology to predict the cache miss rates from reuse distance models for HPC applications. We use the fact that our HPC applications often show periodic patterns in their memory behavior. We will show that the periodic behavior can be utilized to characterize the cache performance using reuse distances when it is running alone. This predictive cache miss models are tested for actual applications represented by NAS benchmarks. Experimental evaluation for our predictive cache miss models will be provided. Finally, we provide the summary of this chapter.

3.1 Modeling Cache Performance in an HPC application using $\alpha + \beta \sin(2\pi t\phi)$ function

In this section, we develop $\alpha + \beta \sin(2\pi t\phi)$ models that seek to characterize periodic behavior that we expect is observable in regard to reuse distance and cache miss profiles of HPC applications. Starting from a brief description of our overall methodology, we develop reuse distance and cache miss models to predict and utilize cache miss rates in an application when it is running alone, i.e., without interference.

3.1.1 Methodology

Our approach is based on studying the relationship between reuse distance and cache miss rates using a simple synthetic benchmark that we call *CodeP* which is described in later paragraphs. *CodeP* would be run on multicore system and data will be generated to find correlations on that particular system between miss rates and reuse distances. These data and regression analysis will be used to estimate values of parameters in the models that we propose later in this section. Our main conjecture is that even though these parameters are obtained using synthetic code data, they would also be equally valid for actual high performance data applications. This conjecture is tested later in Section 3.2. Below we provide the details of our synthetic benchmark and how the data are collected.

CodeP Synthetic Benchmark We would like to define a logical time unit to map to a series of discrete events. Let t be the logical time that corresponds to N memory references. Then, the reuse distance for time t is the average reuse distance observed in N memory accesses. Similarly, the number of cache misses for time t is given by the total number of cache misses observed in N memory accesses. Cache miss rates can be computed by dividing cache misses for time t by N. In this study, we use $N = 20,000$.

We develop a synthetic benchmark with built-in periodic behavior of memory accesses. It provides time-dependent data of measures such as the reuse distances and cache misses that we use to illustrate and develop our main contribution in Section 3.1. We call it *CodeP* and it generates R unique memory references that are repeated F number of times for a reuse distance equal to R and a set of observations of length $F \times R$.

The $F \times R$ memory references generated by *CodeP* should be higher than the value of N that we use for memory references corresponding to a logical time interval of length t. To generate observations using *CodeP*, multiple runs will be made for different values of F and R such that $F \times R$ is much bigger than N by factors of 10 or more. These observations will be used for regression of analysis to estimate parameters for $\alpha + \beta \sin(2\pi t\phi)$ models.

Algorithm 2 *CodeP*

```
procedure $CodeP(R, F)$
    $R \leftarrow$ length of unique memory reference
    $F \leftarrow$ number of repetitions
    for $i \leftarrow 1, F$ do
        for $j \leftarrow 1, R$ do
            access memory[j]
        end for
    end for
end procedure
```

3.1.2 Predictive Models

We start by developing a model based on observed values of reuse distances when a single application runs on a multicore without any interference. Next, we consider how this model could be used to derive a model for predicting cache misses.

In the rest of this section, we use simple illustrative examples based on $CodeP$, our synthetic benchmark. Further, we consider instances with small reuse distances in the range of $10 \sim 50$, and small frequencies in the range of $10 \sim 100$ matched with the gem5 simulation of a conventional two level cache hierarchy with a very small 2KB cache. To estimate the parameters for the models that we will develop in next section, we use multiple runs of $CodeP$ and their observed cache misses to generate the data set that we will use to estimate correlation.

R_{A_i}: A Model for Reuse Distance from Observations for Application A_i. We now characterize periodic behavior in observed reuse distance profiles as an $\alpha + \beta \sin(2\pi t\phi)$ model.

Consider an observed reuse distance profile OR_{A_i} for $codeP(30, 10)$ as shown in Figure 3.1 by a solid line with a step-like shape. By applying an FFT to OR_{A_i}, we can obtain an frequency-domain representation in terms of dominant magnitudes and their frequencies: $FFT(OR_{A_i}) = (M_1^R, M_2^R, I_1^R, I_2^R)$. We can now construct a model R_{A_i} as follows:

$$R_{A_i} = M_1^R + M_2^R \times 2\sin(2\pi t \times abs(I_1^R - I_2^R)). \tag{3.1}$$

Here in the $\alpha + \beta \sin(2\pi t\phi)$ notation, $\alpha = M_1^R$ represents the average reuse distances

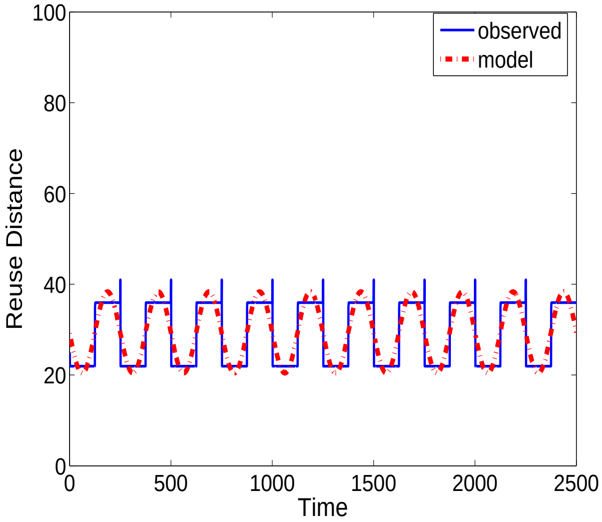


Figure 3.1: Observed reuse distance profile(solid line) and its sine function model(dash-dot line) for $CodeP(30,10)$ application.

observed in OR_{A_i}, and $\beta = M_2^R$ represents the deviation of the reuse distance from the average reuse distance value while $\phi = abs(I_1^R - I_2^R)$ is the frequency. For the observed data shown by the solid line in Figure 3.1, this method yields the model $R_{A_i} = 29.39 + 8.94 \sin(2\pi t \times 10)$, which is also shown in Figure 3.1 as a dash-dot line. Note that from our model, the average reuse distance is 29.39 which is within 0.02% error of the actual reuse distance of 30 from $CodeP(30,10)$. Further, observe that although the model does not capture the corners in Figure 3.1 of the original data, we conjecture that it may be sufficiently accurate to bring predictive ability for co-scheduling applications.

$\overline{C}_{A_i}$: A Predictive Model for Cache Misses derived from R_{A_i}. Given an observed cache miss profile OC_{A_i} of application A_i, we could develop its corresponding model C_{A_i} as we had developed R_{A_i} from the observed OR_{A_i}. However, this would require the trace data observed by running application A_i on a given multicore. To limit such experimental runs and corresponding observations, we would like in general to be able to derive a predictive model for cache misses $\overline{C}_{A_i}$ using only the R_{A_i} model of its reuse distance. Additionally, we would need a small set of "training data" to characterize how reuse distances correspond to cache misses for a given multicore. If this method is successful, then we could easily derive predictive models for cache miss rates for an application on a given multicore starting from only its reuse distance profile. Further, in some cases, the reuse distance profile of an application can be derived through our analysis. In such cases, no observations are needed with the application to predict its cache behavior, thus adding to predictive capabilities.

For illustrative purposes, consider data that show average cache miss rates corresponding to average reuse distances from a gem5 simulation of a multicore with two level caches of very small size at 2KB. Figure 3.2 shows these data as points and the linear model from regression between reuse distance and cache misses as a line.

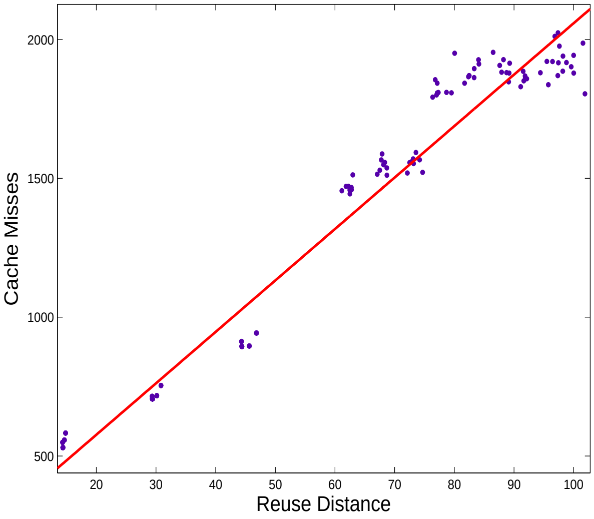


Figure 3.2: Correlation between reuse distance and cache misses for a multicore with 2KB cache. Consider for example, cache misses of 1,500 corresponding to a reuse distance of 60, 24 times the reuse distance value.

Consider the reuse distance model that we derived earlier for $CodeP(30, 10)$, namely $R_{A_i} = 29.39 + 8.94 \sin(2\pi t \times 10)$. We conjecture that the miss rates will correspond to the average reuse distances when scaled by a factor of 24 from Figure 3.2, and we expect that the dominant frequencies for both reuse distance and cache misses to be the same to give $\overline{C}_{A_i} = 705.69 + 107.28 \sin(2\pi t \times 10)$. Although $\overline{C}_{A_i}$ is a profile, we can easily recover the cache miss and rates. Recall that t corresponds to N memory references, we use $N = 20,000$, i.e., now the average cache miss rates is given by $705.69/20,000$ (α/N) at 3.53%. This approach is illustrated for $CodeP(30, 10)$ in Figure 3.3, where the dash-dot line is the predicted

$\overline{C}_{A_i}$ and the actual observed misses OC_{A_i} are shown by a solid line.

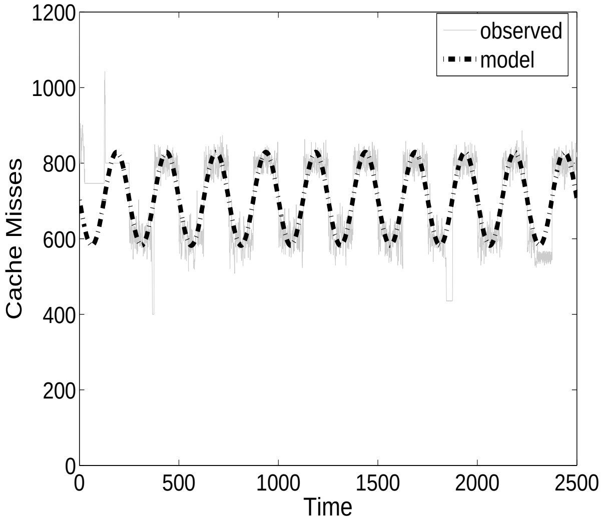


Figure 3.3: Observed cache miss profile (solid line) and its sine function model(dash-dot line) for $CodeP(30, 10)$ application running on a system with cache size 2KB.

Summary. Our predictive models are developed for an application A_i using only *its observed reuse distance* combined with certain model parameter estimates obtained by regression analysis of observed reuse distances and cache misses for its synthetic $CodeP$. Observed cache miss rates including OC_{A_i} are used solely to evaluate the quality of corresponding models, namely $\overline{C}_{A_i}$.

We have presented our key ideas in terms of very simple $\alpha + \beta \sin(2\pi t\phi)$ models. However, we expect that our method can be easily generalized to produce more complex representations of periodic functions when appropriate.

3.2 Experimental Results

In this section, we examine the predictive capabilities of our models and evaluate model accuracy. We start with a description of our experimental setup. We then verify our models for predicting cache miss rates when an application is running alone, i.e., with no interference.

3.2.1 Experimental Setup

System and Simulation. We used the gem5 simulation framework [33] to collect detailed reuse distances, cache misses, and interference statistics for evaluating our $\alpha + \beta \sin(2\pi t \phi)$ models. We test two different types of computer system architectures.

We simulated a conventional computer architecture system that has a two level cache hierarchy. L1 cache was set up as private and L2 cache and all lower level memory hierarchy components were set up as shared. The L1 cache size is 32KB for instruction cache, 32KB for data cache and L2 cache size is 512KB for unified cache of both instruction and data. All caches have a 64-byte line size and utilize an LRU replacement policy. Private L1 caches are 8-way set associative and shared L2 cache has 16-way set associativity.

We also considered a Network-on-Chip (NoC) architecture to test the validity of our predictive cache miss models for S-NUCA. The network topology was set up as a mesh with 4 mesh rows. A private 32KB L1 data cache, a private 32KB instruction cache and a shared 512KB L2 cache were set up for NoC architecture simulation. The associativity for L1 and L2 are 8 and 16 respectively.

Test Suite. Seven HPC applications are selected from the NAS Parallel Benchmark suite [34] for testing our methodology, namely, Conjugate Gradient (CG), Multigrid (MG), Integer Sort (IS), Lower Upper diagonal (LU), Scalar Penta-diagonal (SP), Block Tri-diagonal (BT), and Fast fourier Transform (FT), with class A workloads. The descriptions for each application are as follow:

- Conjugate Gradient (GC) - Computes an estimation of the smallest eigenvalue of a large sparse symmetric positive-definite matrix.

- Multigrid (MG) - Solves a three-dimensional discrete Poisson equation.

- Integer Sort (IS) - Sorts the particles to the appropriate place in a particle method application.

- Lower Upper Diagonal (LU) - Solves a regular sparse LU triangular system using a symmetric successive over-relaxation method.

- Scalar Penta-diagonal (SP) - Solves a nonlinear PDE using scalar pentadiagonal kernel.
- Block Tri-diagonal (BT) - Solves a nonlinear PDE using block tridiagonal kernel.
- Fast Fourier Transform (FT) - Solves a three-dimensional PDE using discrete fast Fourier Transform.

Labels assigned for each application are shown in Table 3.1.

Benchmark	IS	MG	CG	SP	BT	FT	LU
Label	A_1	A_2	A_3	A_4	A_5	A_6	A_7

Table 3.1: Set of applications used for our model verification.

Model Parameter Estimation. We need to estimate values of parameter k_1 in for the cache miss model of the form $\overline{C}_{A_i} = k_1 R_{A_i}$ without interference. We use multiple runs of *CodeP* to first obtain the dataset that is then used to fit a line using regression as discussed in Section 3.1.

k_1
0.03445

Table 3.2: Model parameters. k_1 is used in $\overline{C}_{A_i} = k_1 R_{A_i}$.

3.2.2 Verification: Predicting Cache Misses from Models of Reuse Distance, i.e., $\overline{C}_{A_i} = k_1 R_{A_i}$.

We first test our predictive cache miss model when there is no interference during the execution of a given application. Table 3.3 shows our reuse distance and cache miss models for all applications. Figure 3.4(a), 3.4(b), 3.4(c), 3.4(d), 3.4(e), 3.4(f), and 3.4(g) show the details of reuse distance model results when compared to the observed reuse distance profiles for IS(A_1), MG(A_2), CG(A_3), SP(A_4), BT(A_5), FT(A_6), and LU(A_7). Our modeled average reuse distances match to observed values with errors of 0.0008%, 0.00076%, 0.01%, 0.001%, 0.00066%, 0.00071%,

$\alpha + \beta \sin(2\pi t\phi)$ Models			
Reuse Distance (10^3)			
	α	β	ϕ
R_{IS}	69.177	25.747	12
R_{MG}	43.994	7.392	42
R_{CG}	126.719	7.628	853
R_{SP}	65.395	22.974	208
R_{BT}	17.948	10.302	21
R_{FT}	11.807	3.581	26
R_{LU}	57.196	15.683	669
Cache Miss			
	α	β	ϕ
$\overline{C}_{IS}$	2383.15	887.75	12
$\overline{C}_{MG}$	1515.62	254.683	42
$\overline{C}_{CG}$	4365.5	262.8156	853
$\overline{C}_{SP}$	2252.88	791.46	208
$\overline{C}_{BT}$	618.34	354.93	21
$\overline{C}_{FT}$	406.78	123.35	26
$\overline{C}_{LU}$	1970.41	540.303	669

Table 3.3: Reuse distance models derived from FFT conversion of observed reuse distance profiles, $R_{A_i} = FFT(OR_{A_i})$, and cache miss models derived from $\overline{C}_{A_i} = k_1 R_{A_i}$, where $k_1 = 0.03445$, for all applications when they are run alone, with no interference.

and 0.00095%, respectively for IS, MG, CG, SP, BT, FT, and LU, thus, verifying them as being highly accurate.

Figure 3.5(a), 3.5(b), 3.5(c), 3.5(d), 3.5(e), 3.5(f), and 3.5(g) show the details of our predictive cache miss models when compared to the observed cache miss profiles for IS(A_1), IS(A_2), MG(A_3), CG(A_4), SP(A_5), BT(A_6), and LU(A_7). The observed cache miss rates for IS, MG, CG, SP, BT, FT, and LU are 11.895%, 7.57%, 22.73%, 12.86%, 3.24%, 2.49%, and 9.79% respectively compared to the modeled cache miss rates of 11.915%, 7.58%, 21.83%, 11.26%, 3.09%, 2.03%, and 9.85% (obtained by dividing the dominant amplitude by $N = 20,000$). These values are quite accurate with an average error of 2.01%.

3.2.3 Verification: Predicting Cache Misses from Models of Reuse Distance, i.e., $\overline{C}_{A_i} = k_1 R_{A_i}$ in Real Architecture System.

We now test our predictive cache miss models for computing cache misses from reuse distance models of the Laplacian mesh smoothing when the application is running alone on a real multicore system. Recall that we estimate values of parameter k_1 in the cache miss model of the form $\overline{C}_{A_i} = k_1 R_{A_i}$. We used 40 runs of $CodeP$ synthetic benchmark to obtain the dataset that is used to find correlation between reuse distances and cache misses in a given multicore system. We estimated the parameter for our predictive cache miss model as $\overline{C}_{A_i} = 0.003909 R_{A_i} + 110.27$. We added a constant to our predictive cache miss model for covering various dataset generated by multiple $CodeP$ runs on Intel Westmere multicore system.

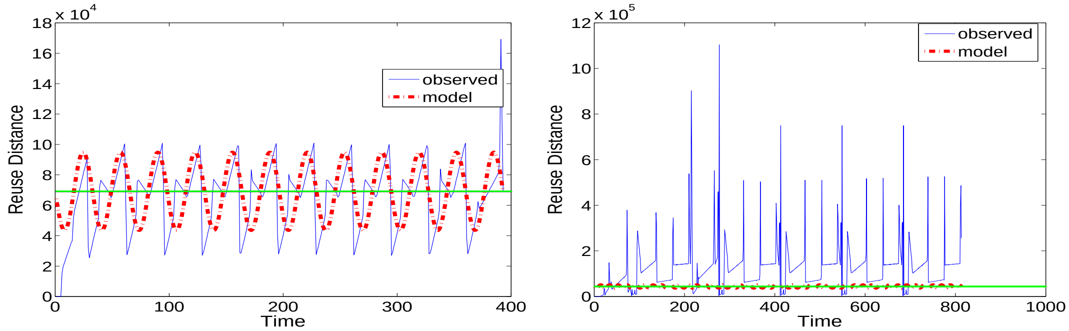
Table 3.4 shows reuse distance models we obtained using our simple periodic function, $\alpha + \beta \sin(2\pi t \phi)$, where $\phi = 8$ was obtained in this test. Figure 3.6 shows cache miss rate results obtained from our predictive cache miss models. Overall prediction error rates are 5.36%.

3.3 Summary

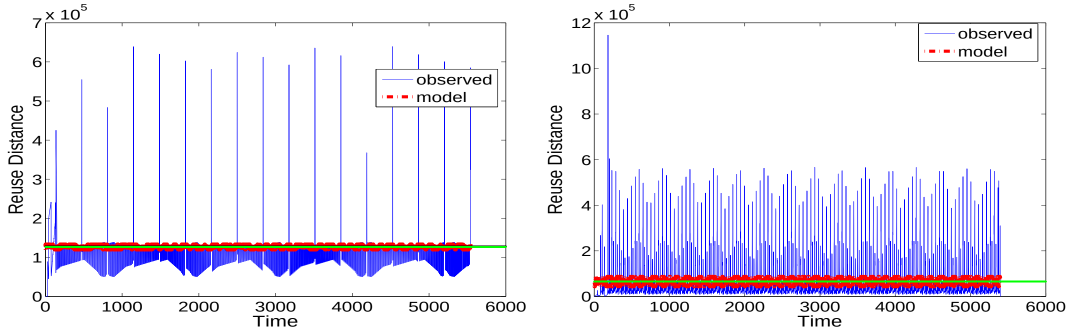
We have developed a predictive cache miss model for HPC applications running on a multicore with no interference. Our approach is based on our conjecture that the repeated basic block structure in scientific codes on HPC multicores would result in periodic behavior of their reuse distances. We showed that a simple $\alpha + \beta \sin(2\pi t \phi)$ model of the observed reuse distance for each application could be utilized to build a predictive cache miss model for a particular application when it is running alone. The prediction error rates were low at 2.01% on average. From this predictive cache miss models for the application when running by itself with no interference, we will develop a predictive cache miss models for an application in the presence of interference in the next chapter.

$\alpha + \beta \sin(2\pi t\phi)$ Models			
		Reuse Distance	
	Ordering	α	β
$R_{carabiner}$	ORI	1896.46	1331.72
	BFS	1349.27	1068.06
R_{crake}	ORI	1898.97	1012.62
	BFS	1272.41	816.08
R_{dialog}	ORI	1708.97	635.26
	BFS	977.93	479.65
R_{lake}	ORI	2478.62	849.39
	BFS	1379.73	675.88
$R_{riverflow}$	ORI	2794.31	1839.15
	BFS	1835.05	1351.63
R_{ocean}	ORI	4425.42	2448.97
	BFS	2906.54	1897.39
R_{stress}	ORI	1557.93	735.61
	BFS	932.42	510.54
R_{valve}	ORI	1919.92	1696.31
	BFS	1242.08	679.38
R_{wrench}	ORI	1937.08	784.04
	BFS	1146.79	583.13

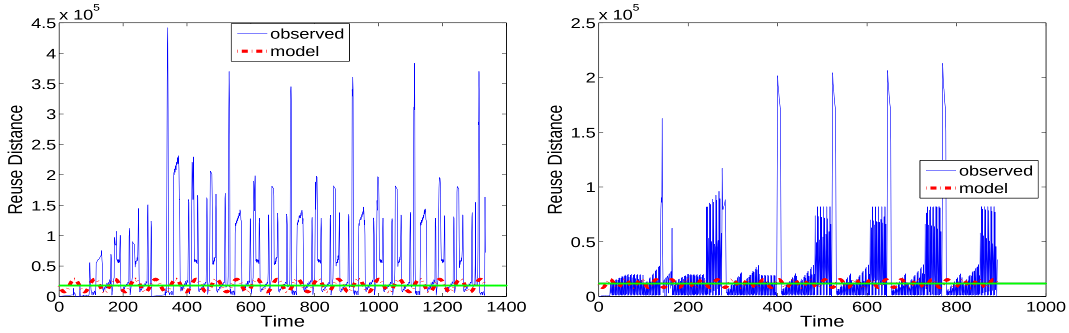
Table 3.4: Reuse distance models of Laplacian mesh smoothing for carabiner, crake, dialog, lake, riverflow, ocean, stress, valve, and wrench meshes when original ordering and BFS ordering are applied.



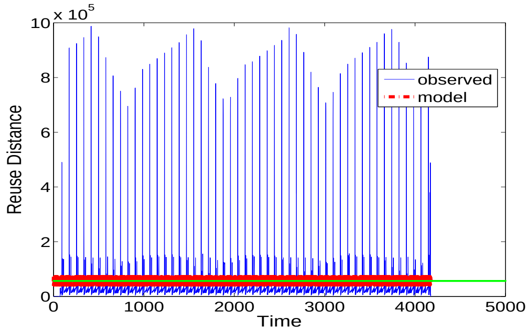
(a) IS (b) MG



(c) CG (d) SP

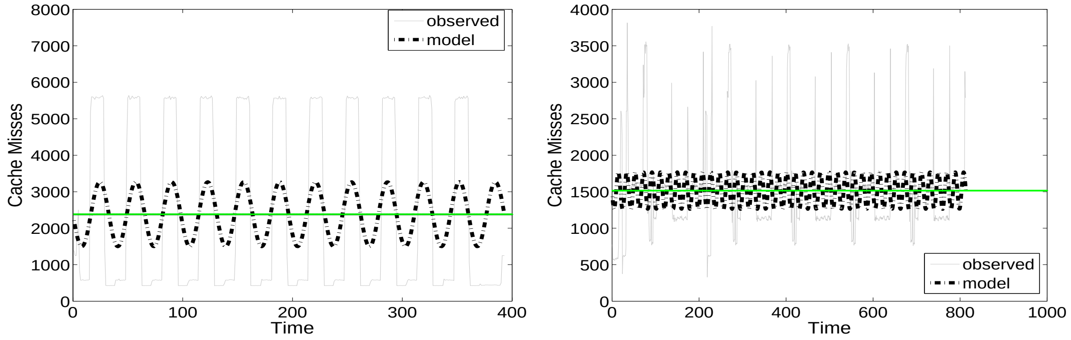


(e) BT (f) FT

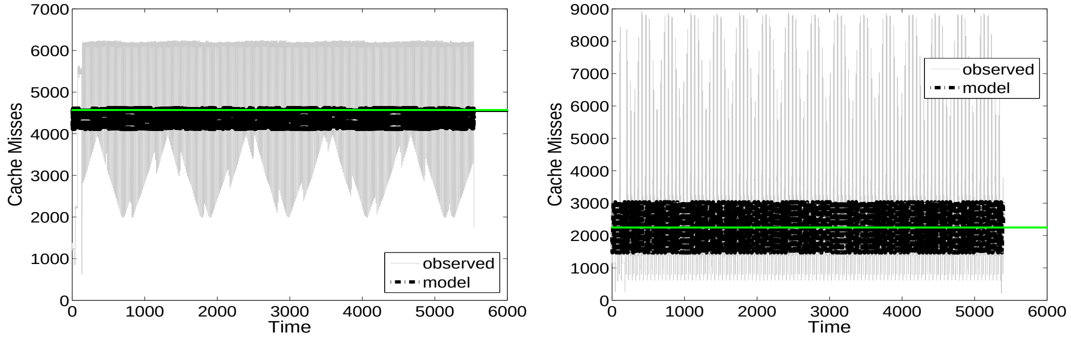


(g) LU

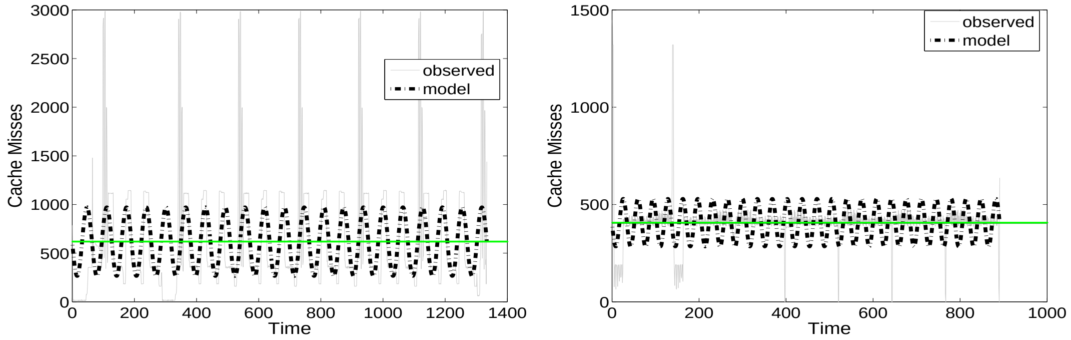
Figure 3.4: Observed reuse distance profiles OR_{A_i} and their reuse distance models $R_{A_i} = FFT(OR_{A_i})$ for (a) IS(A_1), (b) MG(A_2), (c) CG(A_3), (d) SP(A_4), (e) BT(A_5), (f) FT(A_6), and (g) LU(A_7). Average reuse distances shown as lines for IS, MG, CG, SP, BT, FT, and LU are 69118.3, 44028.6, 127999.9, 65328.8, 17937.1, 11799.5, and 57141.8 when observed and 69177.06, 43994.78, 126719.9, 65395.64,



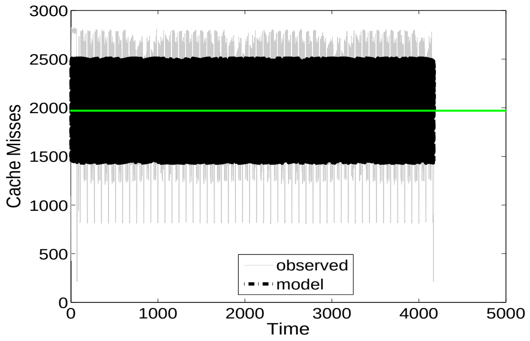
(a) IS (b) MG



(c) CG (d) SP

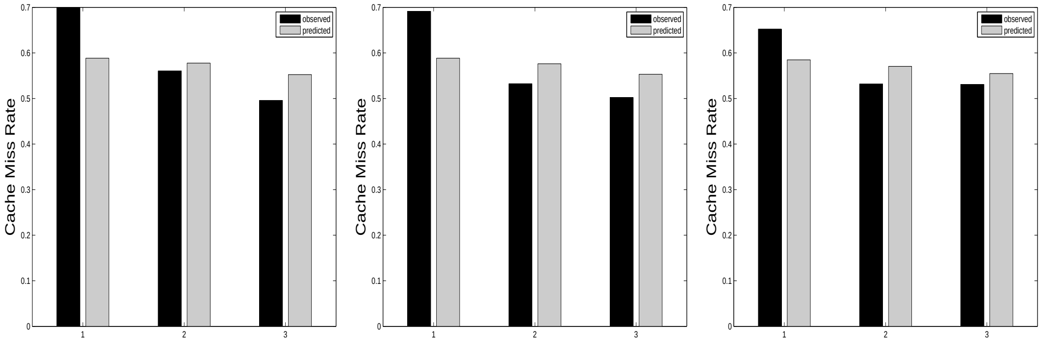


(e) BT (f) FT

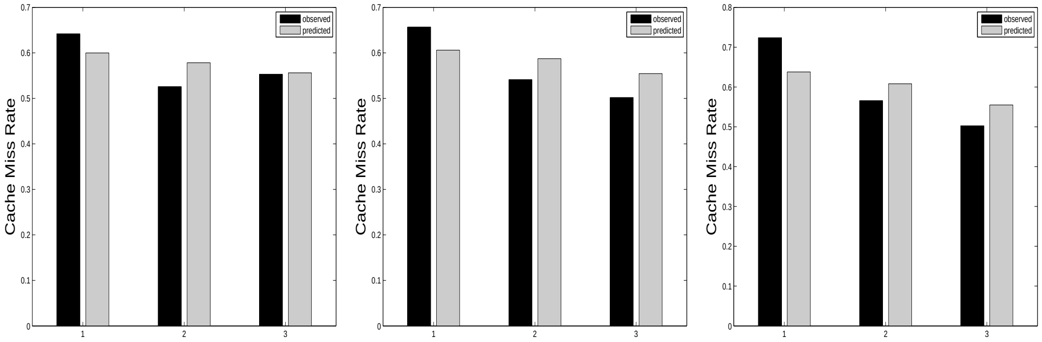


(g) LU

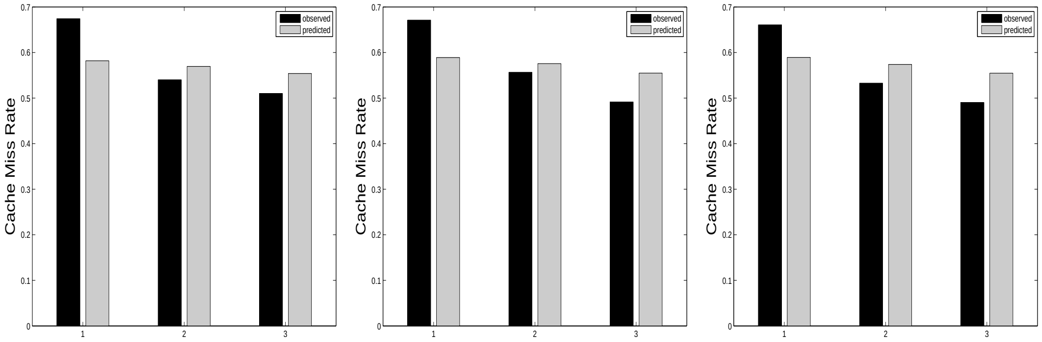
Figure 3.5: Observed cache miss profiles OC_{A_i} and their cache miss models $\overline{C}_{A_i} = 0.03445R_{A_i}$ for (a) IS(A_1), (b) MG(A_2), (c) CG(A_3), (d) SP(A_4), (e) BT(A_5), (f) FT(A_6), and (g) LU(A_7). The dominant amplitudes are shown as lines for both observed and predicted (α) values. Observe that these lines are nearly coincident



(a) carabiner (b) crake (c) dialog



(d) lake (e) riverflow (f) ocean



(g) stress (h) valve (i) wrench

Figure 3.6: Our predictive cache miss model results of Laplacian mesh smoothing for (a) carabiner, (b) crake, (c) dialog, (d) lake, (e) riverflow, (f) ocean, (g) stress, (h) valve, and (i) wrench meshes when ORI, BFS, and RDR ordering are applied. Overall prediction error rates are 5.36%.

Chapter 4

Can Cache Miss Models be Predictive in the face of Interference?

In this chapter, we develop a methodology to predict the cache miss rates from reuse distance models for HPC applications when they are co-scheduled. Similar to the previous chapter, we use the fact that our HPC applications often show periodic patterns in their memory behavior. We will show that the periodic behavior can be used for developing predictive cache miss models for the application in the presence of interference, i.e., two applications are running simultaneously. This predictive cache miss models are tested for actual applications represented by NAS benchmarks. Experimental evaluation for our predictive cache miss models will be provided. Finally, we provide the summary of this chapter.

4.1 Modeling Interference in Co-scheduled HPC Applications Using $\alpha + \beta \sin(2\pi t\phi)$ function

In this section, we develop a series of models to predict and utilize cache miss rates in the presence of interference from co-scheduled applications. Starting from a brief description of our overall methodology, we develop reuse distance and cache miss models to predict and utilize cache miss rates in an application when it is interfered by co-scheduled application.

4.1.1 Methodology

We first studied the relationship between reuse distance and cache miss rates using *CodeP* described in previous chapter. *CodeP* would be run on multicore system and data will be generated to find correlations on that particular system between miss rates and reuse distances. Further, multiple pairs of *CodeP* instances would be run on the system to observe impacts on the reuse distances from interference. These data and regression analysis will be used to estimate values of parameters in the models that we propose later in this section. Our main conjecture is that even though these parameters are obtained using synthetic code data, they would also be equally valid for actual high performance data applications. This conjecture is tested later in Chapter 4.

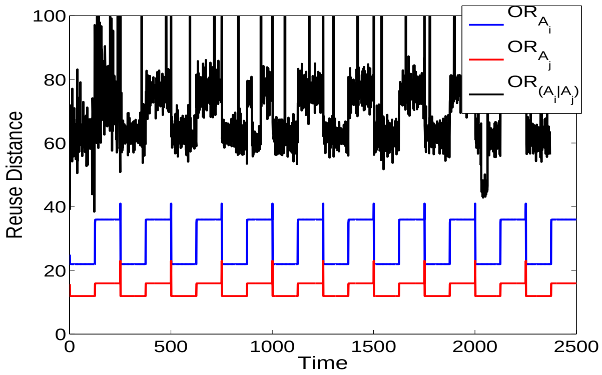
4.1.2 Predictive Models

From the predictive cache miss model we developed in previous chapter, we extend these ideas to develop predictive cache miss models for reuse distances and cache behavior in the presence of interference when two applications are co-scheduled on a multicore.

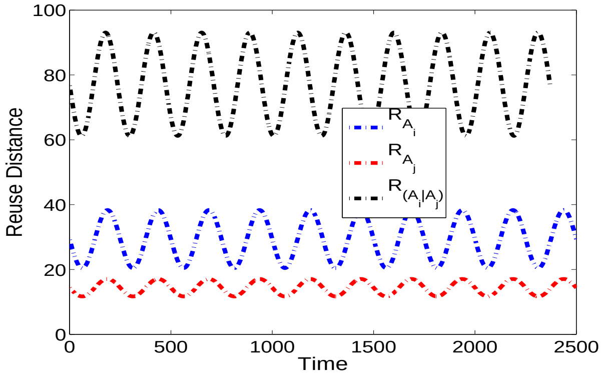
In the rest of this section, we use simple illustrative examples based on *CodeP*, our synthetic benchmark. The parameters for the models that we will develop in next section are estimated from the data set generated by multiple runs of *CodeP* to find correlation.

$\overline{R}_{(A_i|A_j)}$: A Predictive Model of Reuse Distance of Application A_i with Interference by Application A_j. We now consider how R_{A_i} and R_{A_j}, namely reuse distance models of A_i and A_j when they run alone and without interference, can be used to obtain models in the presence of interference when they are co-scheduled. We thus seek to derive $\overline{R}_{(A_i|A_j)}$ of A_i in the presence of interference by A_j based on the assumption that periodic behavior is still preserved although its parameters will be affected by interference.

Consider the illustrative example shown in Figure 4.1. A_i corresponds to $CodeP(30, 10)$ and A_j corresponds to $CodeP(10, 10)$. Their observed reuse dis-



(a) Observed reuse distance profiles



(b) Modeled reuse distance profiles

Figure 4.1: Observed reuse distance profiles (a) for $CodeP(30, 10)$, namely A_i, for $CodeP(10, 10)$, namely A_j, and for A_i in the presence of interference by A_j. The corresponding modeled profiles are shown below in (b).

tances, namely OR_{A_i} and OR_{A_j} when running alone and for A_i with interference from A_j ($OR_{(A_i|A_j)}$) are shown in Figure 4.1(a). We can see that periodic behavior is preserved in $OR_{(A_i|A_j)}$. We therefore conjecture that a model for the interfered reuse distance of A_i can be given by a linear combination of the form $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$ where c_1, c_2, and c_3 are suitable parameters. We consider determining the parameters by seeking correlations between interfered and uninterfered instances. Figure 4.2 shows the regression model for a dataset obtained from single and pairwise runs of our $CodeP$ synthetic benchmark. The surface gives $0.9659R_{A_i} + 1.389R_{A_j} + 0.8509$. Figure 4.1(b) shows these modeled values for $\overline{R}_{(A_i|A_j)} = 77.14 + 15.93 \sin(2\pi t \times 10)$.

In the illustrative example above, we estimated model parameters for reuse distances of interfered pairs of *CodeP* runs using regression models of observed reuse distance profiles and cache miss rates for *CodeP*. Consequently, there is no predictive capability here. However, we expect to use model parameters estimated from *CodeP* runs to also model arbitrary application pairs under interference. Now, the only observed values for these arbitrary applications will be there reuse distances when running alone. It remains to be seen if our conjecture is indeed valid or not, and correspondingly, if $\overline{R}_{(A_i|A_j)}$ will bring predictive capability.

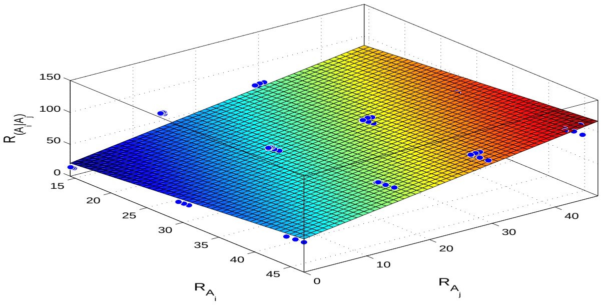


Figure 4.2: Correlation between reuse distances for two applications and the interfered reuse distance in a given multicore system.

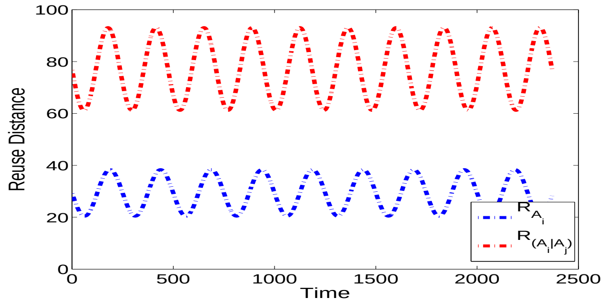
$\overline{C}_{(A_i|A_j)}$: A Predictive Model of Cache Misses of Application A_i with Interference by Application A_j. We would now like to get a predictive cache miss model $\overline{C}_{(A_i|A_j)}$ under interference. We conjecture that it can be obtained much as we had earlier obtained cache miss rates from reuse distances without interference, namely by scaling by k_1. Thus, $\overline{C}_{(A_i|A_j)} = k_1\overline{R}_{(A_i|A_j)}$. Figure 4.3 shows our overall approach for deriving this predictive cache miss model of application A_i with interference by application A_j.

$$\begin{tikzcd}
OR_{A_i} \arrow[r] & R_{A_i} \arrow[r] \arrow[dr] & \overline{C}_{A_i} = k_1 R_{A_i} \arrow[r] & \overline{C}_{(A_i|A_j)} = k_1\overline{R}_{(A_i|A_j)} \\
OR_{A_j} \arrow[r] & R_{A_j} \arrow[r] & \overline{R}_{(A_i|A_j)} \arrow[ur] &
\end{tikzcd}$$

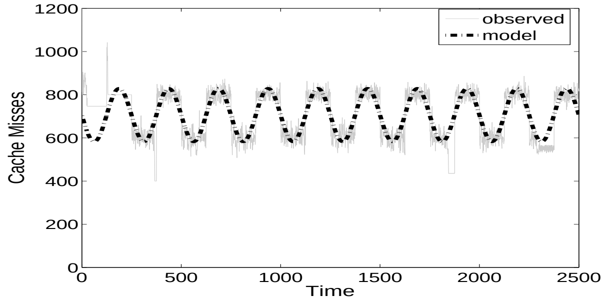
Figure 4.3: Summary of our proposed approach to develop a predictive model of cache misses of application A_i with interference by application A_j when they are co-scheduled.

Figure 4.4 shows the observed and modeled cache miss profiles for $CodeP(30, 10)$ (A_i) when interfered by $CodeP(10, 10)$ (A_j). In this example, $\overline{C}_{(A_i|A_j)} = 1851.58+$

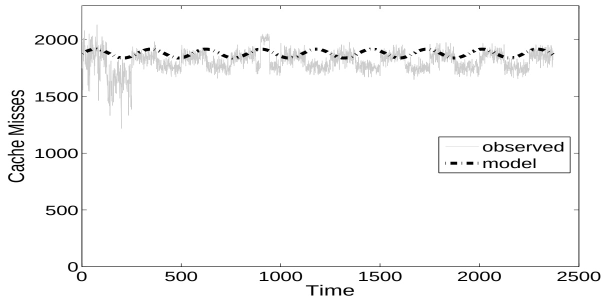
$42.19 \sin(2\pi t \times 10)$ compared to $\overline{C}_{A_i} = 705.69 + 107.28 \sin(2\pi t \times 10)$ indicates nearly a doubling of miss rates.



(a) Modeled reuse distances for A_i without and with interference



(b) Cache misses for A_i



(c) Cache misses for A_i under interference from A_j

Figure 4.4: $CodeP(30, 10)$ and $CodeP(10, 10)$ are used as A_i and A_j. Reuse distance models are shown in (a) for R_{A_i} and $\overline{R}_{(A_i|A_j)}$, observed cache misses OC_{A_i} and modeled cache misses $\overline{C}_{A_i}$ are shown in (b), and finally, when co-scheduled in the presence of interference, the observed cache misses $OC_{(A_i|A_j)}$ and the modeled cache misses $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$ are shown in (c).

4.2 Experimental Results

4.2.1 Experimental Setup

System and simulation setups are exactly the same as the previous chapter. Seven HPC applications selected from the NAS Parallel Benchmark suite are co-scheduled for testing our methodology.

Model Parameter Estimation. We need to estimate values of parameter k_1 in for the cache miss model of the form $\overline{C}_{A_i} = k_1 R_{A_i}$ without interference and which is also used in $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$. Further, we need to estimate parameters c_1, c_2, and c_3 for the reuse distance model with interference of the form $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$. We use multiple runs of *CodeP* to first obtain the dataset that is then used to fit a line and surface using regression as discussed in Section 4.1.

k_1	c_1	c_2	c_3
0.03445	1.178	0.1245	-9593

Table 4.1: Model parameters. k_1 is used in $\overline{C}_{A_i} = k_1 R_{A_i}$, $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$. Further, c_1,c_2, and c_3 are used in $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$.

4.2.2 Verification: Predicting Reuse Distances in the Presence of Interference, i.e., $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$.

We now predict the reuse distances of A_i when interference occurs due to co-scheduled application A_j as a linear combination: $\overline{R}_{(A_i|A_j)} = 1.178 R_{A_i} + 0.1245 R_{A_j} - 9593$.

Table 4.2 shows the average reuse distances results for A_i in the presence of interference derived from our predictive reuse distance model. For example, when IS(A_1) is interfered by FT(A_6), the least amount of increase of reuse distance is obtained; BT, MG, LU, SP, and CG reflect an increasing ordering of their impacts on reuse distance. The range of average reuse distance for IS in the presence of interference is 73367.65–87674.2. Other applications' pairs are also ordered based on average reuse distances increased by interference from co-scheduling.

A_i	Interference least ⟷ most						Range of Interfered Average Reuse Distance(10^3)
IS	FT	BT	MG	LU	SP	CG	73.367∼87.674
MG	FT	BT	LU	SP	IS	CG	44.467∼58.009
CG	FT	BT	MG	LU	SP	IS	141.153∼148.295
SP	BT	FT	MG	LU	IS	CG	69.677∼83.219
BT	FT	MG	LU	SP	CG	IS	18.671∼27.327
FT	BT	MG	SP	LU	IS	CG	12.458∼20.093
LU	FT	BT	MG	SP	IS	CG	59.254∼73.561

Table 4.2: Predicted reuse distances in the presence of interference using $\overline{R}_{(A_i|A_j)} = 1.178R_{A_i} + 0.1245R_{A_j} - 9593$ for each application when paired with one of the remaining six in order of increasing levels of interference. The ranges of interfered reuse distances are also shown.

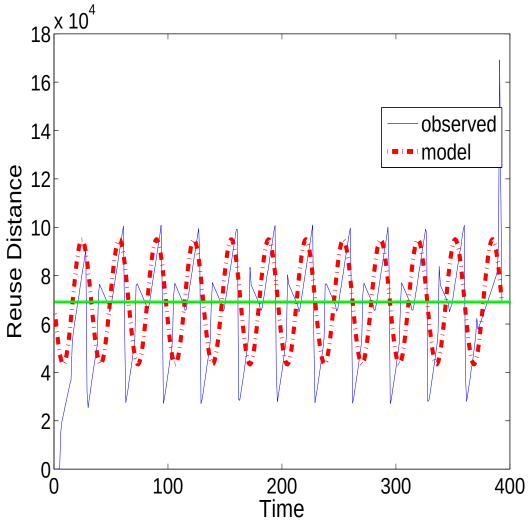
A_i	A_j	Reuse Distance Model (10^3)
IS	FT	$\overline{R}_{(IS\|FT)} = 73.367 + 30.776\sin(2\pi t \times 12)$
	CG	$\overline{R}_{(IS\|CG)} = 87.674 + 31.280\sin(2\pi t \times 12)$
CG	FT	$\overline{R}_{(CG\|FT)} = 141.153 + 9.432\sin(2\pi t \times 26)$
	IS	$\overline{R}_{(CG\|IS)} = 145.259 + 12.192\sin(2\pi t \times 26)$
SP	BT	$\overline{R}_{(SP\|BT)} = 69.677 + 28.346\sin(2\pi t \times 208)$
	CG	$\overline{R}_{(SP\|CG)} = 83.219 + 28.013\sin(2\pi t \times 208)$

Table 4.3: Reuse distance models for IS(A_1), CG(A_3) and SP(A_4) in the presence of interference. Pairs shown in the reuse distance models represent the least and the most increased reuse distances for IS, CG, and SP.

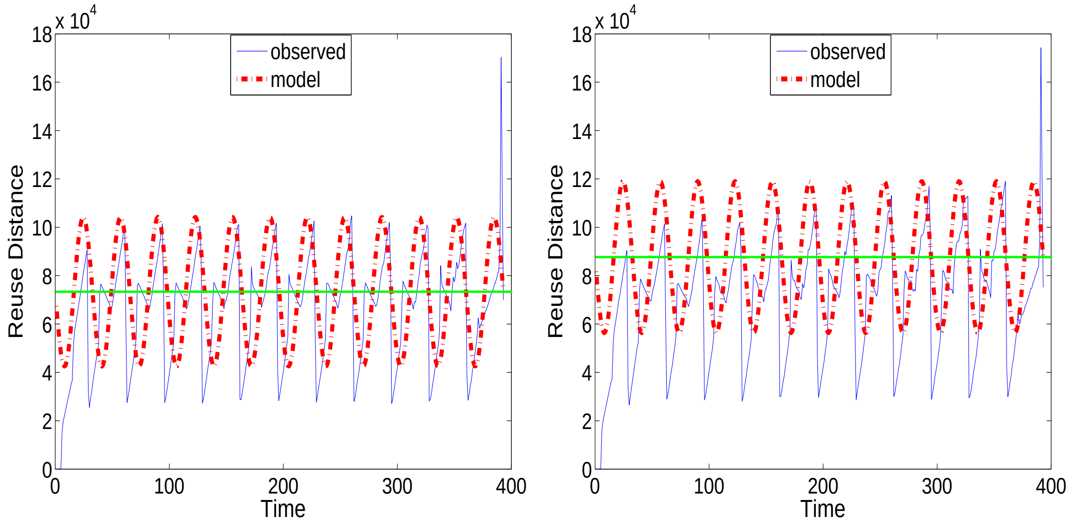
Table 4.3 shows the reuse distance models for IS, CG, and SP in the presence of interference from co-scheduled applications. For each application, the least interfered and the most interfered case results are shown.

Figure 4.5 show the reuse distance models for IS(A_1) in the presence of interference from FT(A_6) and CG(A_3). When IS is interfered by co-scheduled FT, the observed average reuse distance increases to 73909.96, as compared to 73367.65 predicted by our reuse distance model $\overline{R}_{(IS|FT)}$. When IS is interfered by co-scheduled CG, the most increase of average reuse distance, 87003.18 is observed, which is close to the predicted value of 87674.2. Compared to the observed reuse distances for IS in the presence of interference from FT and CG, the average reuse distances computed by our reuse distance models are sufficiently accurate with 0.7% error on average.

Figure 4.6 show the reuse distance models for CG(A_3) in the presence of inter-



(a) IS with no interference

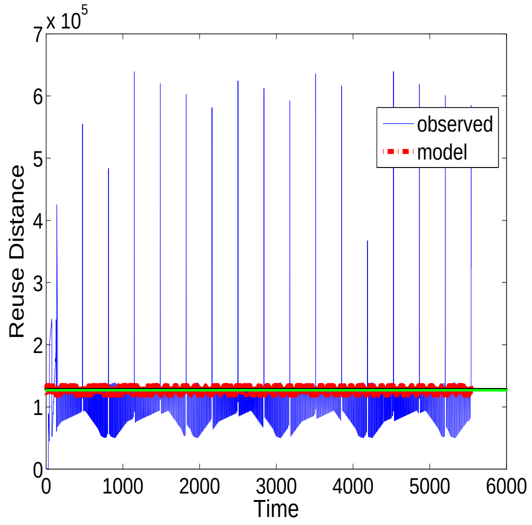


(b) IS interfered by FT

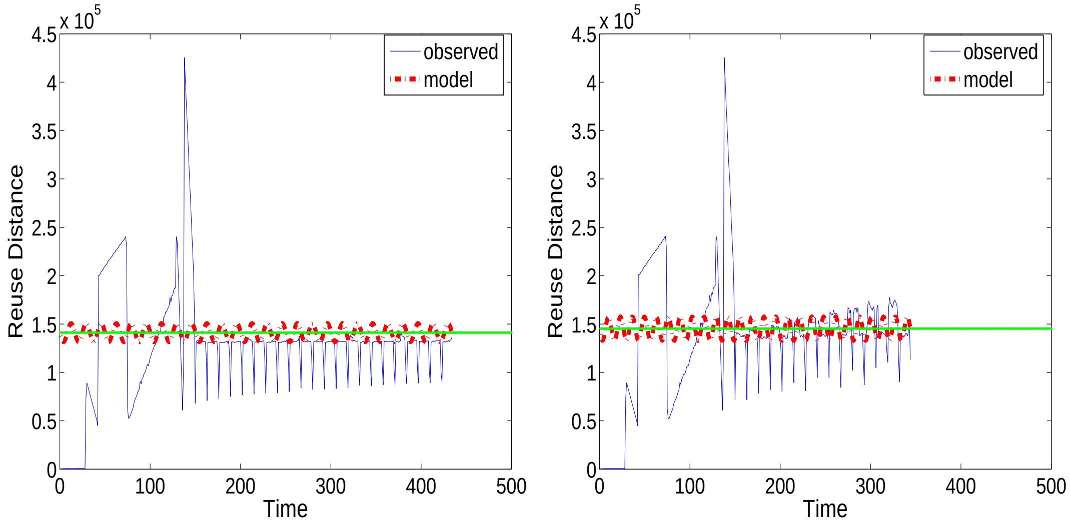
(c) IS interfered by CG

Figure 4.5: Observed reuse distance profile OR_{A_i} and the reuse distance model $R_{A_i} = FFT(OR_{A_i})$ for IS(A_1)(a).Observed reuse distance profiles $OR_{(A_i|A_j)}$ and the reuse distance models $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$, where $c_1 = 1.178$, $c_2 = 0.1245$, and $c_3 = -9593$, in the presence of interference from FT(A_6)(b) and CG(A_3)(c).

ference from FT(A_6) and IS(A_1). When CG is interfered by co-scheduled FT, the observed average reuse distance increases to 137964.64, as compared to 141153.1 predicted by our reuse distance model $\overline{R}_{(CG|FT)}$. When CG is interfered by co-



(a) CG with no interference

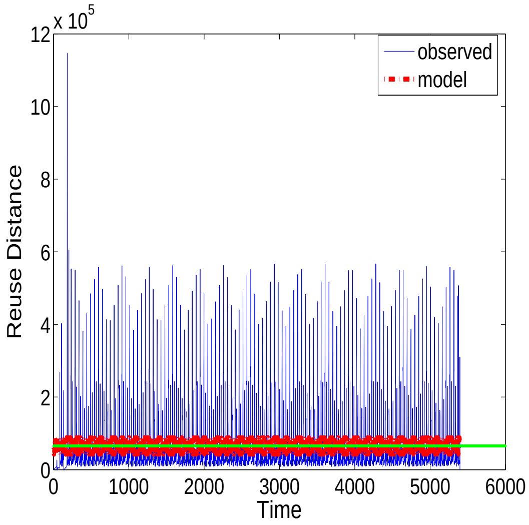


(b) CG interfered by FT

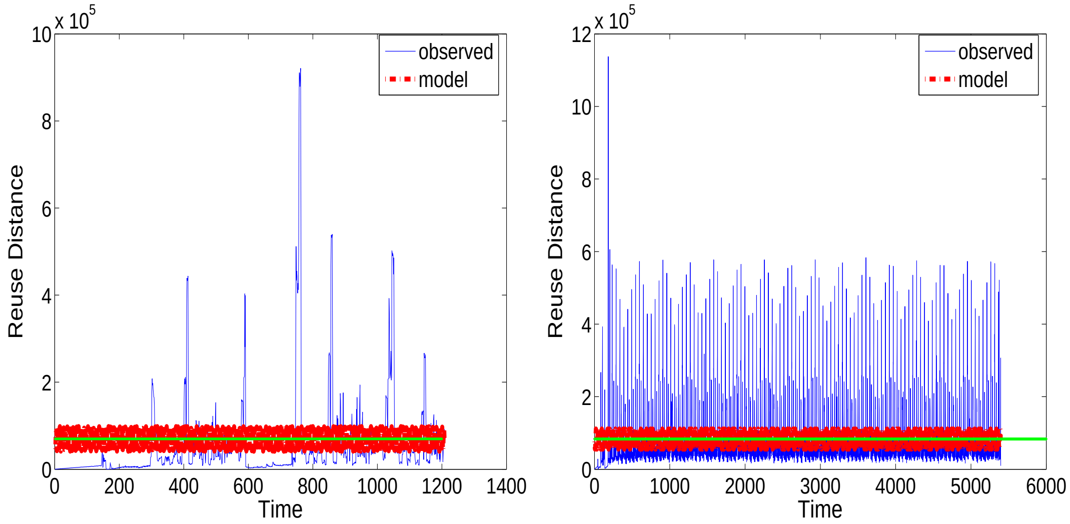
(c) CG interfered by IS

Figure 4.6: Observed reuse distance profile OR_{A_i} and the reuse distance model $R_{A_i} = FFT(OR_{A_i})$ for CG(A_3)(a). Observed reuse distance profiles $OR_{(A_i|A_j)}$ and the reuse distance models $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$, where $c_1 = 1.178$, $c_2 = 0.1245$, and $c_3 = -9593$, in the presence of interference from FT(A_6)(b) and IS(A_1)(c).

scheduled IS, the most increase of average reuse distance, 147072.13 is observed, which is close to the predicted value of 145259.58. Compared to the observed reuse distances for CG in the presence of interference from FT and IS, the average reuse distances computed by our reuse distance models are sufficiently accurate with 1.5% error on average.



(a) SP with no interference



(b) SP interfered by BT

(c) SP interfered by CG

Figure 4.7: Observed reuse distance profile OR_{A_i} and the reuse distance model $R_{A_i} = FFT(OR_{A_i})$ for SP(A_4)(a). Observed reuse distance profiles $OR_{(A_i|A_j)}$ and the reuse distance models $\overline{R}_{(A_i|A_j)} = c_1 R_{A_i} + c_2 R_{A_j} + c_3$, where $c_1 = 1.178$, $c_2 = 0.1245$, and $c_3 = -9593$, in the presence of interference from BT(A_5)(b) and CG(A_3)(c).

Figure 4.7 show the reuse distance models for SP(A_4) in the presence of in-

A_i	Interference least ⟷ most						Range of Interfered Cache Misses
IS	FT	BT	MG	LU	SP	CG	2527.51∼3020.37
MG	FT	BT	LU	SP	IS	CG	1505.55∼1998.42
CG	FT	BT	MG	LU	SP	IS	4862.73∼5108.78
SP	BT	FT	MG	LU	IS	CG	2400.4∼2866.91
BT	FT	MG	LU	SP	CG	IS	448.56∼941.43
FT	BT	MG	SP	LU	IS	CG	225.69∼692.23
LU	FT	BT	MG	SP	IS	CG	2067.65∼2534.17

Table 4.4: Predicted cache misses in the presence of interference using $\overline{C}_{(A_i|A_j)} = 0.03445\overline{R}_{(A_i|A_j)}$ for each application when paired with one of the remaining six in order of increasing levels of interference. The ranges of interfered cache misses are also shown.

terference from BT(A_5) and CG(A_3). When SP is interfered by co-scheduled BT, the observed average reuse distance increases to 69104.57, as compared to 69677.7 predicted by our reuse distance model $\overline{R}_{(SP|BT)}$. When SP is interfered by co-scheduled CG, the most increase of average reuse distance, 82081.64 is observed, which is close to the predicted value of 83219.69. Compared to the observed reuse distances for SP in the presence of interference from BT and CG, the average reuse distances computed by our reuse distance models are sufficiently accurate with 1.1% error on average.

4.2.3 Verification: Predicting Cache Misses from Reuse Distance in the Presence of Interference, i.e., $\overline{C}_{(A_i|A_j)} = k_1\overline{R}_{(A_i|A_j)}$.

We now test our model for predicting cache miss rates when an application is interfered by another co-scheduled application. Our methodology includes starting from $R_{A_i} = FFT(OR_{A_i})$, deriving $\overline{C}_{A_i} = k_1 R_{A_i}$, and $\overline{R}_{(A_i|A_j)}$ to obtain $\overline{C}_{(A_i|A_j)} = k_1\overline{R}_{(A_i|A_j)}$.

Table 4.4 shows the predicted cache miss results for A_i in the presence of interference. For application A_i, the co-scheduled applications are shown in order of the increases in miss rates they cause; the range of cache misses are also provided.

Table 4.5 shows the cache miss models for IS, CG, and SP in the presence

A_i	A_j	Cache Miss Model
IS	FT	$\overline{C}_{(IS\|FT)} = 2527.51 + 1060.25\sin(2\pi t \times 12)$
	CG	$\overline{C}_{(IS\|CG)} = 3020.37 + 1077.62\sin(2\pi t \times 12)$
CG	FT	$\overline{C}_{(CG\|FT)} = 4862.73 + 324.95\sin(2\pi t \times 26)$
	IS	$\overline{C}_{(CG\|IS)} = 5108.78 + 420.03\sin(2\pi t \times 26)$
SP	BT	$\overline{C}_{(SP\|BT)} = 2400.39 + 1007.18\sin(2\pi t \times 208)$
	CG	$\overline{C}_{(SP\|CG)} = 2866.92 + 843.283\sin(2\pi t \times 208)$

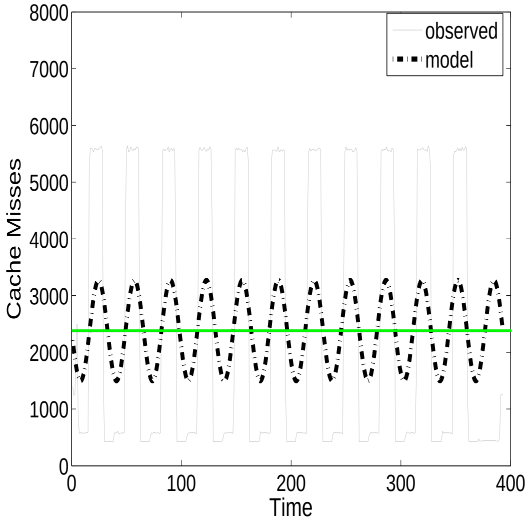
Table 4.5: Cache miss models for IS(A_1), CG(A_3), and SP(A_4) in the presence of interference. Pairs shown in the cache miss models represent the least and the most increased cache misses for IS, CG, and SP.

of interference from co-scheduled applications. Applications which cause the least and the most interferences are paired. The cache miss models for IS in the presence of interference from FT and CG, $\overline{C}_{(IS|FT)}$ and $\overline{C}_{(IS|CG)}$, are shown in Figure 4.8. When IS is interfered by FT and CG, the cache miss rates computed from our cache miss models are 12.64% and 15.1%, as close to the observed cache miss rates of, 12.67% and 16.8%, respectively.

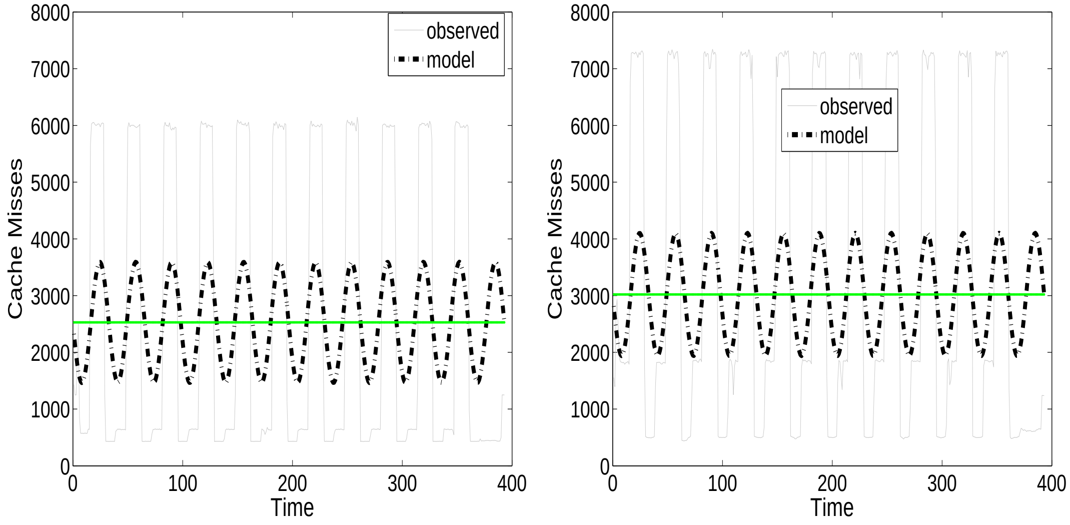
Figure 4.9 shows the cache miss models for CG in the presence of interference from FT and IS, $\overline{C}_{(CG|FT)}$ and $\overline{C}_{(CG|IS)}$. When CG is interfered by FT and IS, the cache miss rates computed from our cache miss models are 24.31% and 25.54%, as close to the observed cache miss rates of, 21.07% and 24.74%, respectively.

Figure 4.10 shows the cache miss models for SP in the presence of interference from BT and CG, $\overline{C}_{(SP|BT)}$ and $\overline{C}_{(SP|CG)}$. When SP is interfered by BT and CG, the cache miss rates computed from our cache miss models are 12% and 14.33%, as close to the observed cache miss rates of, 11.92% and 17.03%, respectively.

Figure 4.11(a) and Figure 4.11(a) show the predicted cache miss rates when interference occurs in the original applications tested on the conventional computer architecture and NoC architecture set up in gem5 Simulation framework. These results are for all seven NAS Benchmarks. These summarizing verification results support the validity of our predictive models. The overall error rates for our predictive cache miss rate model is 3.13% for conventional architecture and 3.98% for NoC architecture.



(a) IS with no interference



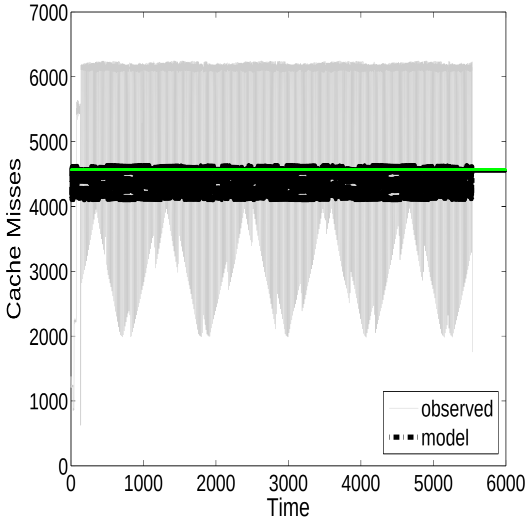
(b) IS interfered by FT

(c) IS interfered by CG

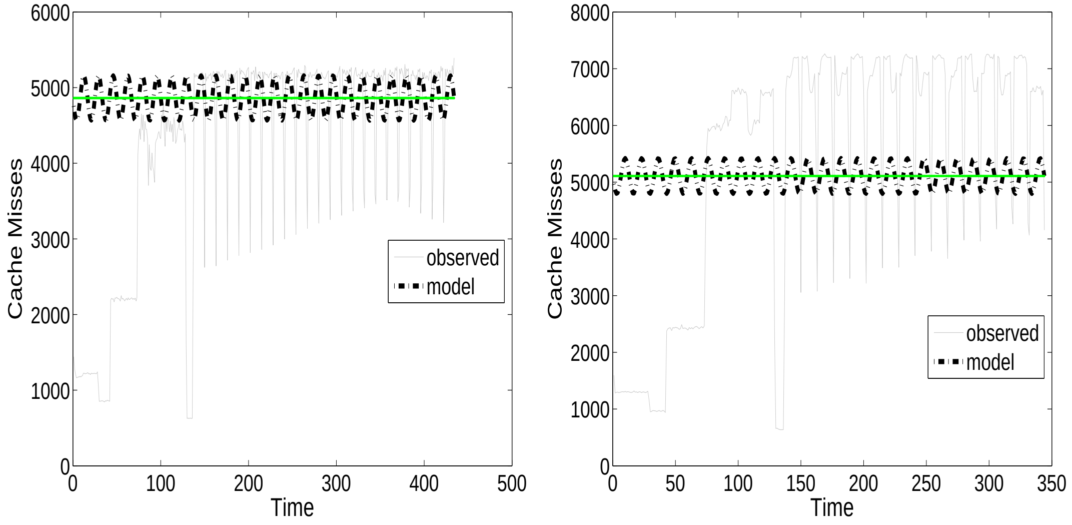
Figure 4.8: Observed cache miss profile OC_{A_i} and the cache miss model $\overline{C}_{A_i} = k_1 R_{A_i}$, where $k_1 = 0.03445$, for IS(A_1)(a). Observed cache miss profiles $OC_{(A_i|A_j)}$ and the cache miss models $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$ in the presence of interference from FT(A_6) (b) and CG(A_3) (c).

4.3 Summary

We have developed a predictive cache miss model in the presence of interference due to co-scheduling of application pairs on a multicore. Our approach is based on



(a) CG with no interference

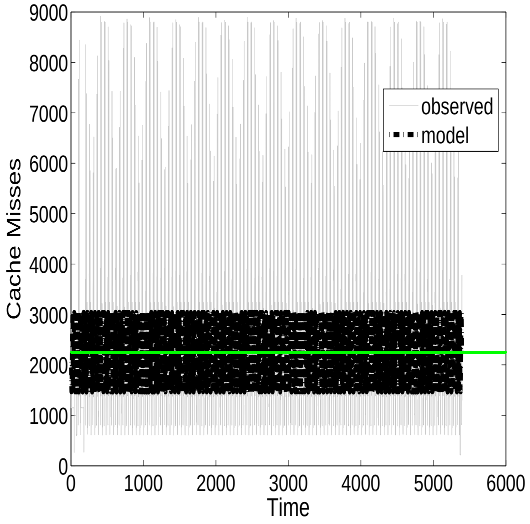


(b) CG interfered by FT

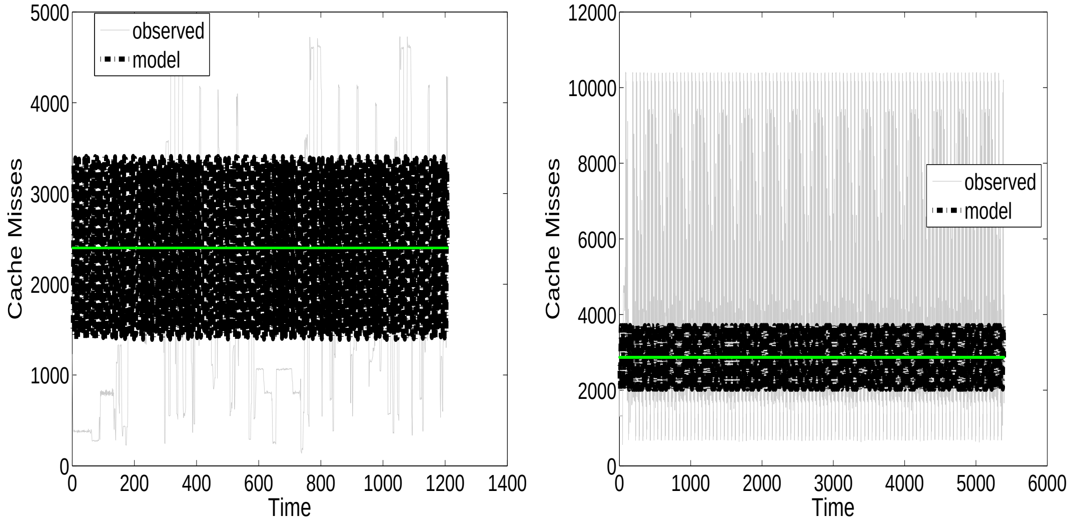
(c) CG interfered by IS

Figure 4.9: Observed cache miss profile OC_{A_i} and the cache miss model $\overline{C}_{A_i} = k_1 R_{A_i}$, where $k_1 = 0.03445$, for CG(A_3)(a). Observed cache miss profiles $OC_{(A_i|A_j)}$ and the cache miss models $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$ in the presence of interference from FT(A_6) (b) and IS(A_1) (c).

our conjecture that the periodic behavior of applications reuse distances happens because the basic code segment block in HPC applications performs repeatedly. From the simple $\alpha + \beta \sin(2\pi t\phi)$ model of the observed reuse distance for each application, we compose appropriately to finally yield a model for predicting cache



(a) SP with no interference

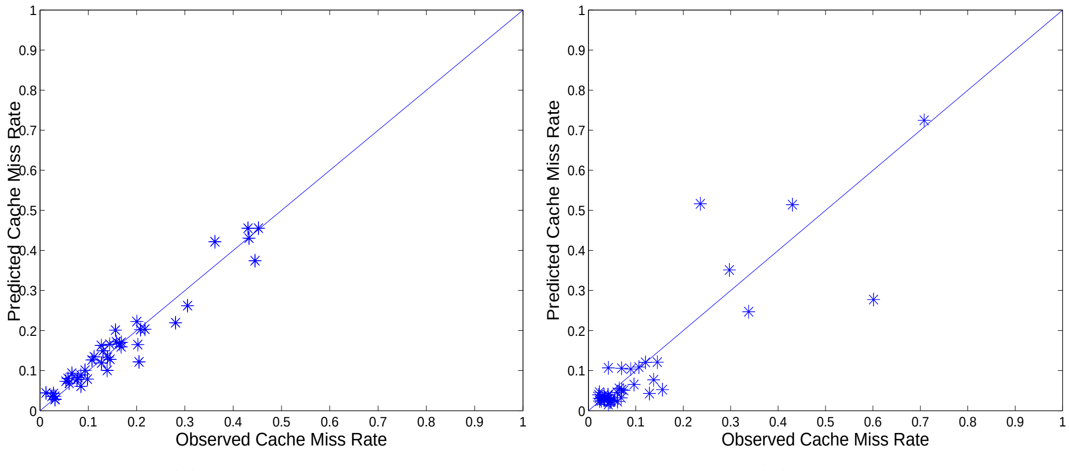


(b) SP interfered by BT

(c) SP interfered by CG

Figure 4.10: Observed cache miss profile OC_{A_i} and the cache miss model $\overline{C}_{A_i} = k_1 R_{A_i}$, where $k_1 = 0.03445$, for SP(A_4)(a). Observed cache miss profiles $OC_{(A_i|A_j)}$ and the cache miss models $\overline{C}_{(A_i|A_j)} = k_1 \overline{R}_{(A_i|A_j)}$ in the presence of interference from BT(A_6) (b) and CG(A_3) (c).

misses for a particular application in the presence of interference. Similar to the methodology we developed in Chapter 3, our parameter estimation for these models involves regression with data observed from about a hundred runs of a simple synthetic code on the target system. When these models were verified using the



(a) Conventional (b) NoC

Figure 4.11: Prediction results for our cache interference miss model on conventional computer architecture (a) and NoC architecture system (b). Overall prediction error rates are 3.13% for conventional architecture with a two level cache hierarchy system and 3.98% for NoC architecture.

NAS benchmarks and gem5 simulation, we observed low error rates of less than 3.13% on average for a conventional multicore with a two-level cache hierarchy and 3.98% for an NoC architecture.

We expect that our models can be extended to include more complex periodic behavior of applications including those with linear trends. Additionally, we expect that our approach can be extended to three co-scheduled applications by composing the model for a pair with the model for a single application. However, parameter estimation will likely need to be more sophisticated to control error rates to find approximations that have sufficient predictive capability. Further, we see the need to add stochastic elements if other factors such as core variation or transient effects of resource contention are to be represented. Finally, we expect that our models could be used in additional useful applications, such as for studying the scalability of multicore cache hierarchies. We anticipate studying these aspects in future work.

Chapter 5

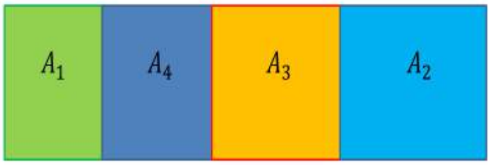
Accelerating Scientific Workloads using Reuse Distances and Cache Miss Models

We now show how to utilize our predictive reuse distances and cache miss models developed in Chapters 3 and 4 for improving the performance of scientific applications. We first test the effectiveness of our cache miss models under interference for informing the choice of application pairs to minimize performance degradation when they are co-scheduled. Additionally, we propose a reuse distance reducing ordering to improve the performance of Laplacian mesh smoothing. We summarize experimental results to demonstrate the utility of reuse distance and cache miss models developed in the preceding chapters.

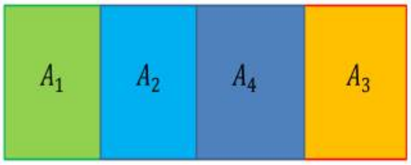
5.1 Co-scheduling using Ranking Results of our Predictive Cache Miss Models under Interference

We would like to consider how to utilize our predictive cache miss models for co-scheduling applications efficiently. Suppose there are n applications A_1, A_2, ... A_n that have to be co-scheduled. We randomly pick an application, for example, A_1, and we predict the cache miss rates when A_1 is co-scheduled with other applications such as $\overline{C}_{(A_1|A_2)}$, $\overline{C}_{(A_1|A_3)}$ and so on. We then rank the paired applications based on the predicted cache miss rates. We select the best partner for A_1 that is ranked as top and this application pair will be co-scheduled. Among the rest of applications, we randomly pick an application again and rank the pair applications based on our predictive cache miss models. This processes are repeated until all applications are scheduled. These application pairs reduce performance degradation due to resource contention, and thus, we can obtain the performance improvement in their execution times.

Figure 5.1 shows how to co-schedule applications efficiently using our ranking system. For example, if there are three candidate applications A_j, A_k, A_l that could be co-scheduled with the application A_i, we can predict $\overline{C}_{(A_i|A_j)}$, $\overline{C}_{(A_i|A_k)}$ and $\overline{C}_{(A_i|A_l)}$ and rank them based on the predicted cache miss rates. The best partner for A_i is the application that causes the least increase in interfered cache miss rates among all.



(a) Worst case for co-scheduling A_1 ,A_2,A_3, and A_4



(b) After applying our raking based Co-scheduling for A_1 ,A_2,A_3, and A_4

Figure 5.1: Total execution times when four applications A_1, A_2, A_3, and A_4 are co-scheduled. When applications are scheduled in a random way (a), we obtained the worst performance. When applications are co-scheduled with our ranking system (b), we first co-schedule applications A_1 and A_2 since the pair of A_1 and A_2 show the least increase of cache misses. The best co-scheduling performance can be obtained through our ranking results of predictive cache miss models.

5.1.1 Experimental Results

We demonstrate using our $\overline{C}_{(A_i|A_j)}$ models to rank applications so that we can determine application pairs for co-scheduling. By comparing the predicted cache miss rates and choosing the pair applications which show least negative impact on cache interference, we can do co-scheduling with minimum performance degradation. Figure 5.2 shows the predicted cache miss rates for IS, MG, CG, SP, BT, FT, and LU applications for choosing the pair applications to be co-scheduled.

Table 5.1 shows the ranking results for IS, MG, CG, SP, BT, FT, and LU to determine the best partner for co-scheduling. By comparing the predicted with the actual observed rankings, we can see that the first and the last match consistently. This shows the pictorial predictive capability for gaining application pairs for high performance.

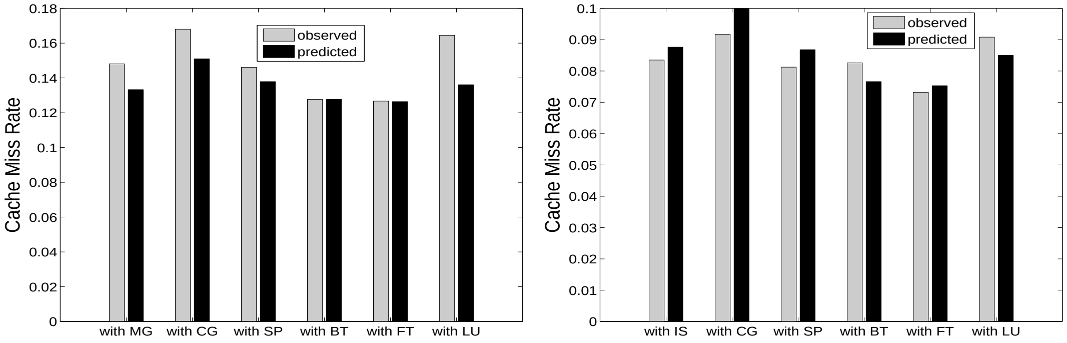
We test our ranking system for co-scheduling six applications, IS, MG, SP, BT, FT, and LU efficiently. Figure 5.3 shows the execution time results when IS, MG, SP, BT, FT, and LU are co-scheduled using our ranking system. We select the application pair FT and BT that show the least increased cache misses among all. We then apply our ranking system to the remaining applications IS, MG, LU, and SP and select the pair of applications IS and MG among them. Finally, the remaining applications, LU and SP, are co-scheduled. We are able to obtain the performance improvement when IS, MG, SP, BT, FT, and LU applications are co-scheduled using our ranking system. Execution time with our ranking system was 5863.46 sec, as compared to the worst case scenario, 7439.45 sec.

5.1.2 Summary of Co-scheduling

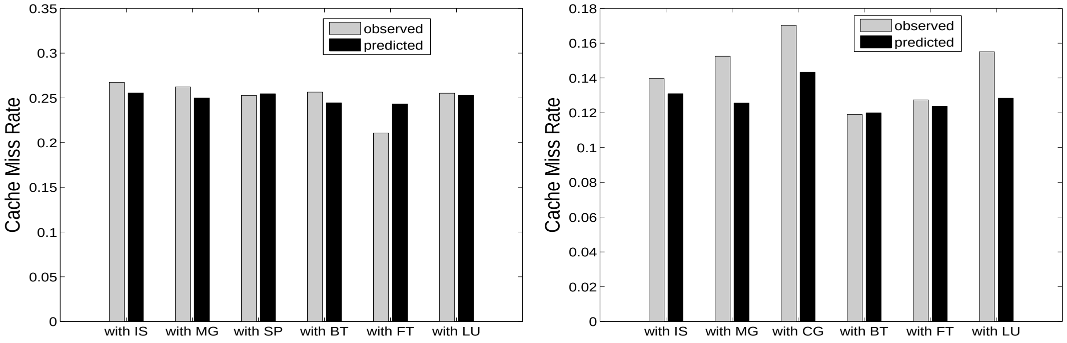
We showed the usage of our predictive cache miss models to improve the performance of HPC applications. The predicted best application pairs in regard to least degradation in performance when co-scheduled, matched with observed performance in our tests, supporting how our models could be utilized in a practical setting. This ranking system can be more precise if more complicated linear models are built for predicting cache miss rates. As a future work, we will improve the ranking results to meet all ranking orders between predicted and observed performance.

For	Co-scheduling Rank					
IS	MG	CG	SP	BT	FT	LU
Predicted	3	**6**	5	**2**	**1**	4
Observed	4	**6**	3	**2**	**1**	5
For	Co-scheduling Rank					
MG	IS	CG	SP	BT	FT	LU
Predicted	5	**6**	4	2	**1**	3
Observed	4	**6**	2	3	**1**	5
For	Co-scheduling Rank					
CG	IS	MG	SP	BT	FT	LU
Predicted	**6**	3	5	2	**1**	4
Observed	**6**	5	2	4	**1**	3
For	Co-scheduling Rank					
SP	IS	MG	CG	BT	FT	LU
Predicted	5	3	**6**	**1**	**2**	4
Observed	3	4	**6**	**1**	**2**	5
For	Co-scheduling Rank					
BT	IS	MG	CG	SP	FT	LU
Predicted	**6**	2	**5**	4	**1**	3
Observed	**6**	3	**5**	2	**1**	4
For	Co-scheduling Rank					
FT	IS	MG	CG	SP	BT	LU
Predicted	5	2	**6**	3	**1**	4
Observed	4	3	**6**	2	**1**	5
For	Co-scheduling Rank					
LU	IS	MG	CG	SP	BT	FT
Predicted	**5**	3	**6**	4	**2**	**1**
Observed	**5**	4	**6**	3	**2**	**1**

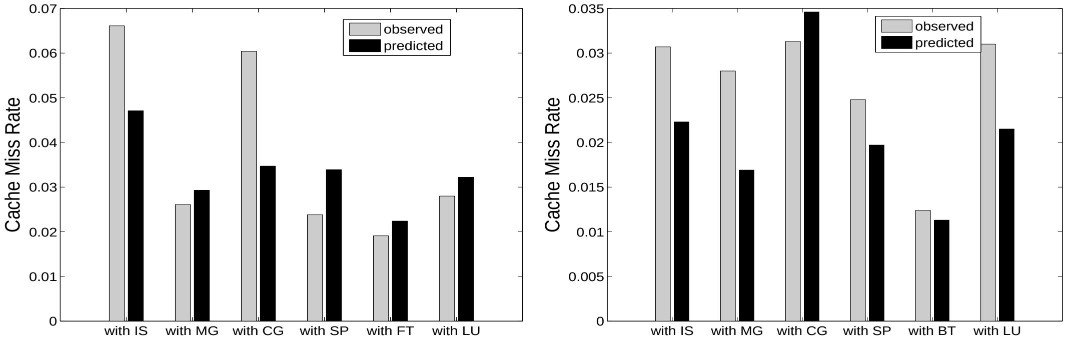
Table 5.1: Predicted ranking of applications for co-scheduling with IS, MG, CG, SP, BT, FT, and LU.



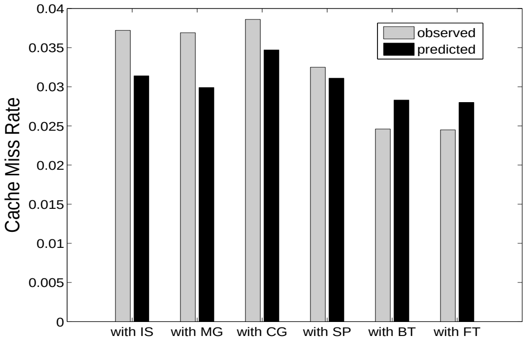
(a) IS (b) MG



(c) CG (d) SP

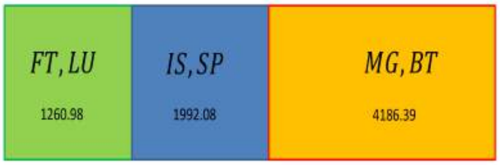


(e) BT (f) FT

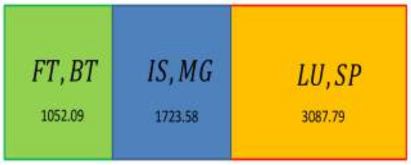


(g) LU

Figure 5.2: Prediction results for our cache interference miss models for (a)IS, (b)MG, (c)CG, (d)SP, (e)BT, (f)FT, and (g)LU. From the predicted cache miss rates, we rank the best pair application for each application to co-schedule each other.



(a) Worst case for co-scheduling



(b) After applying our raking system for co-scheduling

Figure 5.3: Total execution times when six applications IS, MG, SP, BT, FT, and LU are co-scheduled. When applications are scheduled in a random way (a), we obtained the worst performance, 7439.45 sec. When applications are co-scheduled with our ranking system (b), we first co-schedule applications FT and BT since the pair of FT and BT show the least increase of cache misses. Next, we co-schedule applications IS and MG for obtaining the least increase of cache misses among the remaining applications. Improved co-scheduling performance can be obtained through our ranking results of predictive cache miss models, 5863.46 sec.

5.2 Reuse Distance Reducing Ordering for Laplacian Mesh Smoothing

In this section, we show how to utilize our reuse distance models to improve the performance of irregular HPC application, namely, the Laplacian mesh smoothing. In HPC systems, when solving partial differential equations using finite element methods for unstructured meshes in parallel on multicore system, high quality meshes are requited [35]. This is due to the fact that mesh quality plays a significant role in both the accuracy of the solution produced by a PDE solver, as well as the solver's execution time [36]. However, when attempting to parallelize the mesh smoothing application to obtain solution of PDE efficiently, we do not obtain full performance scalability due to the irregular memory accesses of the application [37]. Thus, in order to increase the cache utilization and achieve the benefits of high quality meshes for PDE solvers efficiently, we explore the development of a mesh reordering scheme in this section.

In Chapter 3, we have established that reuse distance plays a significant role in cache performance of HPC applications on multicore [32]. Figure 5.4 shows the performance results for Laplacian mesh smoothing when difference reuse distances are applied. There is a huge range of reuse distances for Laplacian mesh smoothing. When the Laplacian mesh smoothing runs on a randomly ordered mesh, the reuse distances for Laplacian smoothing increase from 4449.64 to 90100.06. Our predictive cache miss model shows that cache miss rates for Laplacian mesh smoothing increase from 0.65% to 2.31% when random ordering is applied to the nodes for Laplacian mesh smoothing. However, when the Laplacian mesh smoothing runs with BFS ordered mesh [28], we observe that the reuse distances for Laplacian smoothing decrease from 4449.64 to 2926.52. Further, our predictive cache miss models show that the reduced reuse distance improves the cache performance of Laplacian mesh smoothing. When the reuse distance for Laplacian mesh smoothing is reduced, cache miss rates decreased from 0.65% to 0.6%. We want to make sure how reuse distances can affect the performance of Laplacian mesh smoothing.

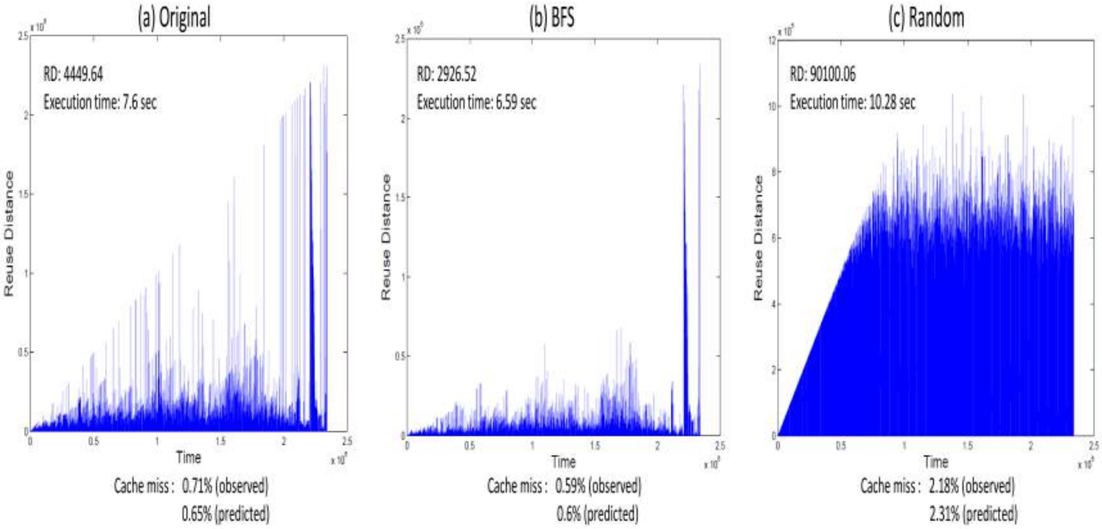


Figure 5.4: Performance results for Laplacian mesh smoothing according to the difference reuse distances with (a) original mesh, (b) BFS ordered mesh, and (c) random ordered mesh.

5.2.1 Factors Affecting Temporal and Spatial Locality

There are two factors that affect the performance of Laplacian mesh smoothing. The first is cache misses around reuses of nodes in Laplacian mesh smoothing. When the Laplacian mesh smoothing needs a node, if the node is already prefetched in the cache, it improves temporal locality, which in turn, reduces cache miss rates. The second is how to stream the nodes for Laplacian mesh smoothing. Suppose that the nodes are spatially stored in the memory. If a node is selected, the node is streamed to the cache along with its neighboring nodes. Since the Laplacian smoothing method processes a node and its neighbors successively, this prefetching of the neighboring nodes into cache improves the applications cache performance. Figure 5.5 shows two partial traces of node visiting observed from Laplacian mesh smoothing when two different reuse distances are applied. When the reuse distances for Laplacian mesh smoothing are reduced by applying BFS ordering, temporal and spatial localities are significantly improved. This example dictates the interplay between temporal and spatial localities on one hand and reducing reuse distances on the other.

... 613,640,701,115,125,613,623,115,121,124,84,173,315,316,342,740,128,173,179,257,315 ...

(a) Original ordering

... 164,166,181,151,152,164,169,181,182,183,184,185,186,187,188,189,151,154,182,185,187,190 ...

(b) BFS ordering

Figure 5.5: Partial traces observed from node visiting for Laplacian mesh smoothing. (a) Traces for Laplacian mesh smoothing with original ordering. (b) Traces for Laplacian mesh smoothing with BFS ordering. When the reuse distances for Laplacian mesh smoothing are reduced by BFS ordering, temporal and spatial localities are significantly improved.

5.2.2 Toward a Reuse Distance Reducing Ordering

We now would like to consider how to reduce the reuse distances for Laplacian mesh smoothing. By the time the Laplacian mesh smoothing terminates, there is a certain order in which nodes have been considered. Consider a synthetic mesh consisting of nodes that have, almost, the same qualities for all nodes, shown in Figure 5.6. Initially, there are certain node traces that the Laplacian mesh smoothing visits during the execution. After that, we permute the nodes in the mesh by applying BFS ordering. The reordered nodes are stored in their updated memory locations. Figure 5.7 shows the mesh reordered by BFS ordering. The nodes in the mesh reordered by BFS ordering store their nodes spatially. This causes the access patterns for Laplacian mesh smoothing to become similar to the memory access of nodes streamed in the cache. Though the accessed node list becomes similar to the streamed node list, there are still spaces for improvement for obtaining optimal reuse distance of Laplacian mesh smoothing.

We now consider improving temporal locality for Laplacian mesh smoothing. When Laplacian smoothing executes, nodes which have bad qualities are visited more frequently than other nodes. Since the Laplacian mesh smoothing is a greedy algorithm, when smoothing the mesh, the Laplacian mesh smoothing visits the

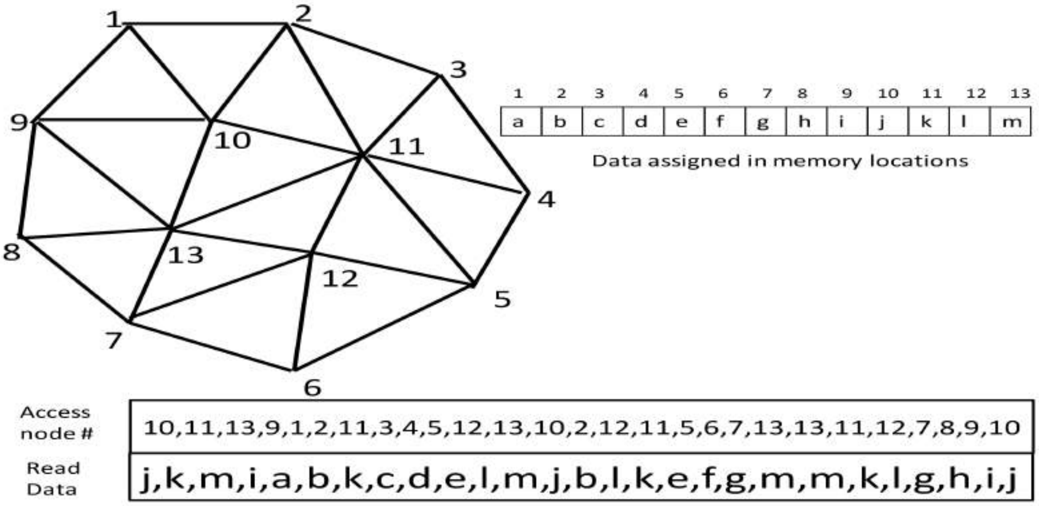


Figure 5.6: Original Mesh that have have almost same qualities for each vertex. The node traces visited by Laplacian mesh smoothing are shown.

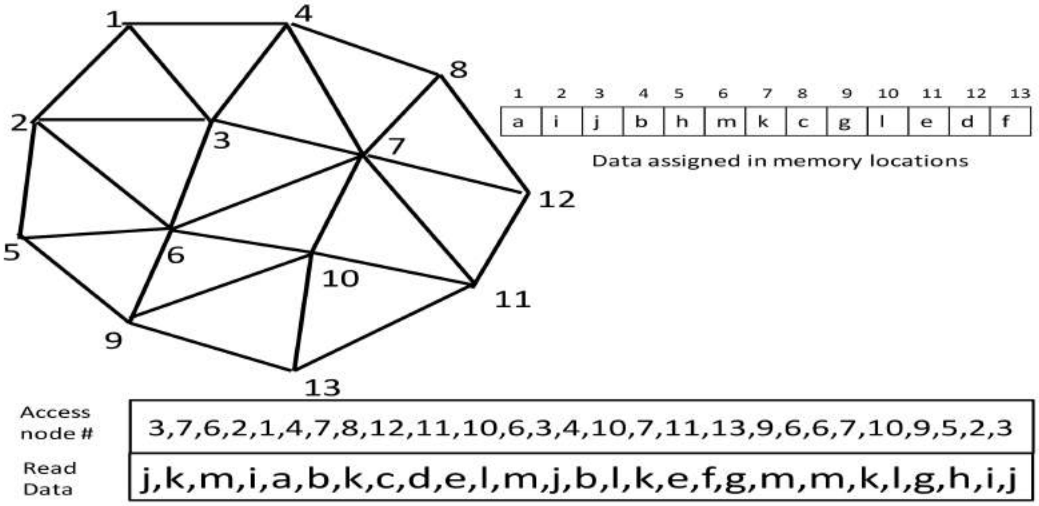
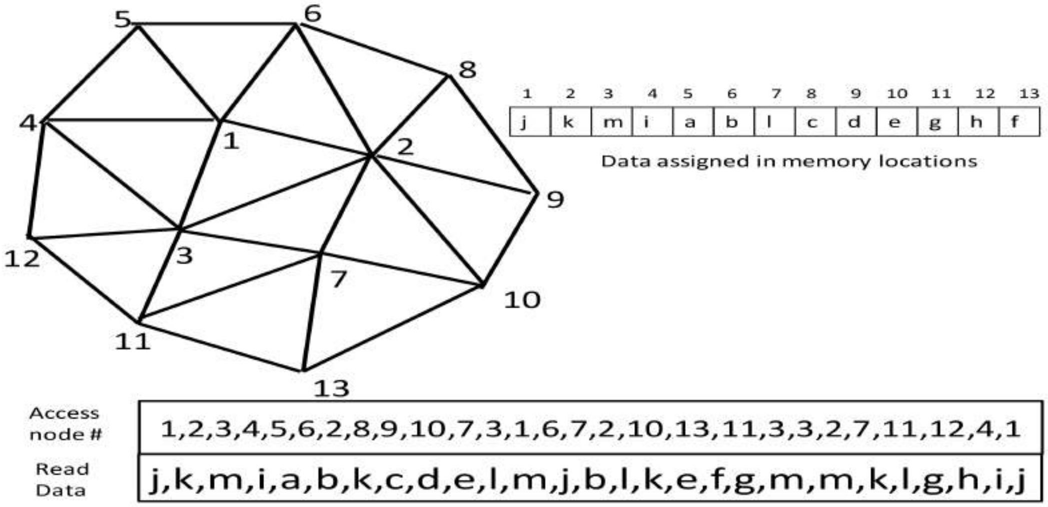


Figure 5.7: BFS ordered Mesh. The node traces showing the visit of Laplacian mesh smoothing have higher spatial locality than original mesh.

node which has bad quality first. Once the smoothing process for the node is over, the Laplacian mesh smoothing selects another node that has the worst quality among nodes nearby the node, i.e., neighboring nodes. We conjecture that the access patterns for Laplacian mesh smoothing can be controlled by the initial qualities of each node in the mesh. If we order nodes and their neighboring nodes based on the qualities they have, the temporal locality will be improved. Figure 5.8

shows the reordered mesh based on the qualities that each node has. Compared to the BFS ordered mesh, the accessed node list becomes even more closed to the streamed node list. Significant reduction of reuse distances for Laplacian mesh smoothing is also observed.



1	2	3	4	5	6	7	8	9	10	11	12	13
j	k	m	i	a	b	l	c	d	e	g	h	f

Access node #	1,2,3,4,5,6,2,8,9,10,7,3,1,6,7,2,10,13,11,3,3,2,7,11,12,4,1
Read Data	j,k,m,i,a,b,k,c,d,e,l,m,j,b,l,k,e,f,g,m,m,k,l,g,h,i,j

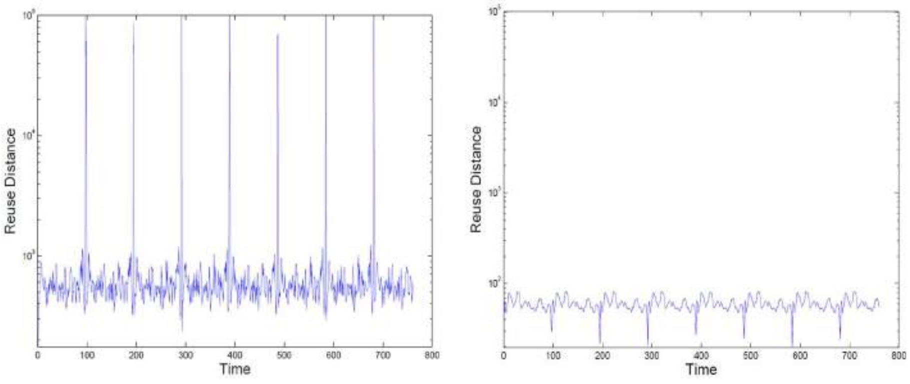
Figure 5.8: Reordered mesh based on the qualities that each node has. The node traces observed from Laplacian mesh smoothing have higher spatial locality and higher temporal locality than original mesh.

By using this conjecture, we propose a mesh reordering scheme that reduces reuse distance of Laplacian mesh smoothing. When mesh smoothing application is executed, the application calculates the initial qualities of each vertex in a mesh, smoothes the mesh vertices, and then computes the quality of the mesh to see the difference between initial and final quality of the mesh. We find that a vertex which has a bad quality in the beginning requires more time to be smoothed compared to other vertices. If such vertices are processed earlier than other vertices, the neighbors of the vertex which has the worst quality are already in the cache and thus, we can improve the spatial locality. The neighboring vertices are also reordered in increasing order. By reordering all the vertices based on the methodology we described above, we are able to obtain the node trace which is very similar to the streamed node list for Laplacian mesh smoothing so that both temporal and spatial localities are improved. The node traces have the reduced reuse distances and consequently we can improve the performance of Laplacian mesh smoothing.

Algorithm 3 *Algorithm for Reuse Distance Reducing Ordering*

```
procedure Reuse Distance Reducing Ordering($V$, $T$)
    $V \leftarrow$ mesh vertex data
    $T \leftarrow$ mesh triangle data
    quality = 0
    for $i \leftarrow 1, V$ do
        compute initial mesh quality for V[i]
        quality = quality + $q_{V[i]}$
    end for
    reordering list = empty
    while $i \leftarrow 1, V$ do
        Pick the vertex V[i] which has the worst quality and put it to the list
        Put the neighbored vertices of V[i] to the reordering list
         based on the increasing order of quality
    end while
    for $i \leftarrow 1, V$ do
        Laplacian Smoothing
    end for
end procedure
```

Figure 5.9 shows the reuse distance profiles for original mesh and the mesh reordered by our reuse distance reducing ordering. Reuse distances are significantly reduced from 1803.83 to 59.48 when our reuse distance reducing ordering is applied.



(a) Original mesh

(b) Reordered mesh

Figure 5.9: Observed reuse distance profiles for original mesh (a) and reordered mesh (b).

5.2.3 Experimental Results

In this section, we evaluate our reuse distance reducing ordering for improving the performance of Laplacian mesh smoothing. We first describe the experimental setup used in this study. We then show the execution times for Laplacian mesh smoothing reduced by our reuse distance reducing ordering. Next, We verify our predictive cache miss models developed in Chapter 3 to confirm that reuse distance can be used to predict cache miss rates for an application when it is running alone. We also test the scalability of the Laplacian mesh smoothing with our reuse distance reducing ordering.

5.2.3.1 Experimental Setup

System Setup. We used an Intel Westmere-EX architecture system to evaluate our reuse distance reducing ordering and to verify our predictive cache miss models developed in Chapter 3. The Intel Westmere-EX architecture equipped with 4 eight-core Intel Xeon E7-8837 processors. It supports 32 concurrent threads. Each core has 32K L1 private cache and 256K L2 private cache, and they share 24MB L3 cache. The machine is an inclusive cache hierarchy. Each processor is directly connected to other three processors via 3.2 GHz QPI links. Figure 5.10 shows the high-level view of the Intel Westmere-EX processor.

For multi-thread running, OpenMP library is used for both systems. Thread affinity is set via KMP_AFFINITY=compact, granularity=fine for pinning each thread to each core. In all cases, thread scheduling is set to be static for simply collecting the application trace of each thread by evenly dividing the vertices. We implemented parallel Laplacian mesh smoothing based on the modular in Mesquite [38].

Test Suite. To determine the impact of reuse distance reducing ordering on the mesh smoothing process, we tested nine meshes shown in Figure 5.11(Coarse approximations are shown). The meshes were generated by Triangle [1] and Table 5.2 gives their configurations.

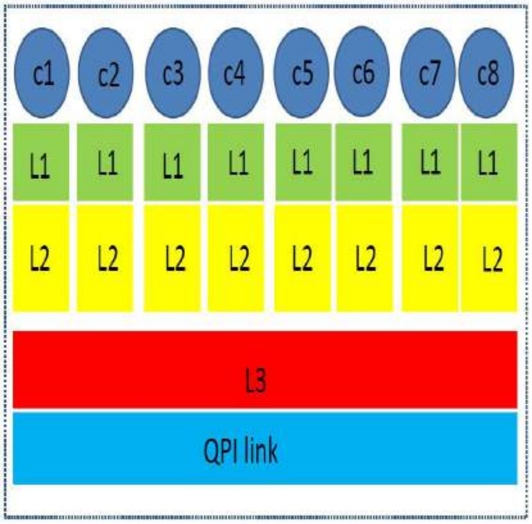


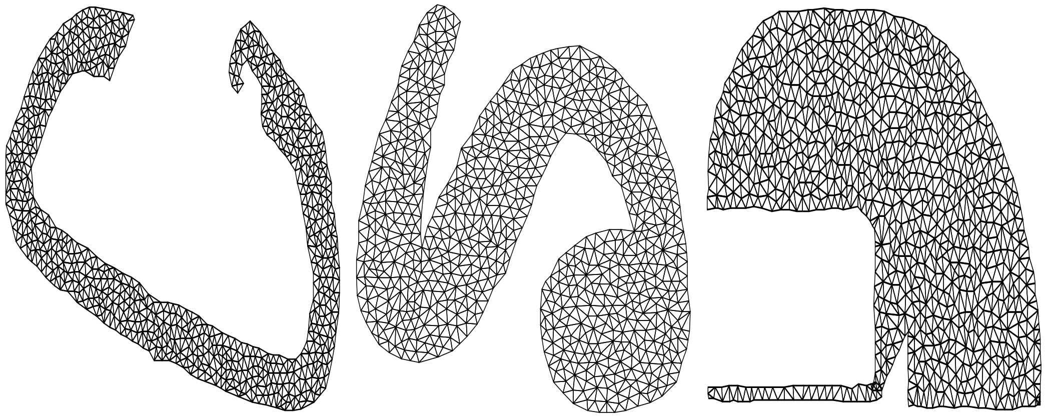
Figure 5.10: High-level views of Intel Westmere-EX processor. Ovals denote to cores and rectangles represent on-chip caches (L1, L2, and L3). The machine has four sockets, which are connected directly via 3.2 GHz OPI links. Each socket has 8 cores with the inclusive cache hierarchy; 32K L1 private cache, 256K L2 private cache, and 24MB shared L3 cache are in the socket, highlighted using dashed box.

Label	Mesh	vertex	triangle
M1	carabiner	328082	652920
M2	crake	298898	595638
M3	dialog	306824	611620
M4	lake	375288	747676
M5	riverflow	332699	661615
M6	ocean	392674	783040
M7	stress	312763	622868
M8	valve	300985	599368
M9	wrench	386757	771097

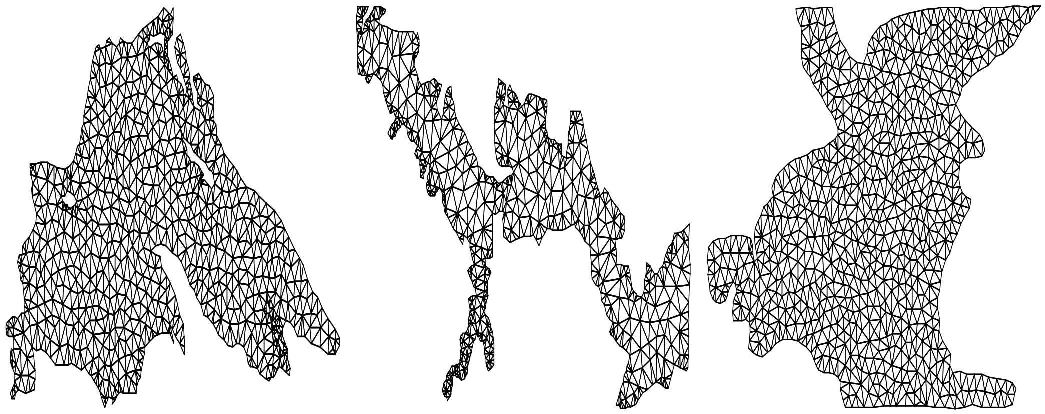
Table 5.2: Input Mesh Configuration

5.2.3.2 Performance: Execution Time and Reduced Reuse Distance

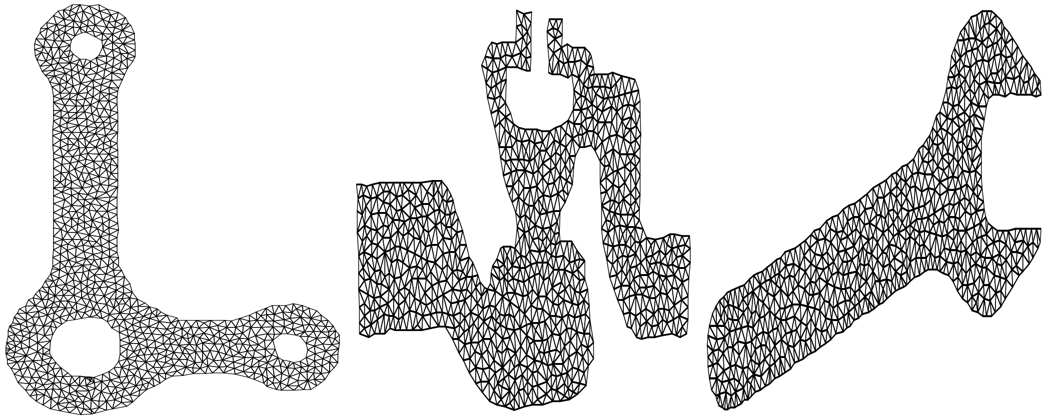
We first test our reuse distance reducing ordering on a serial run of Laplacian mesh smoothing. Figure 5.12 shows the execution time results of Laplacian mesh smoothing when reuse distance reducing ordering was applied. For evaluation purpose, we provide the Breath First Search (BFS) reordering results developed by Strout [28] together. Compared to the original ordering, average execution time was 38.57% faster when we applied reuse distance reducing ordering to the



(a) carabiner (b) crake (c) dialog



(d) lake (e) riverflow (f) ocean



(g) stress (h) valve (i) wrench

Figure 5.11: 2D meshes generated by Triangle [1]. These meshes are coarser, representative versions of the meshes used in the experiments.

Laplacian mesh smoothing.

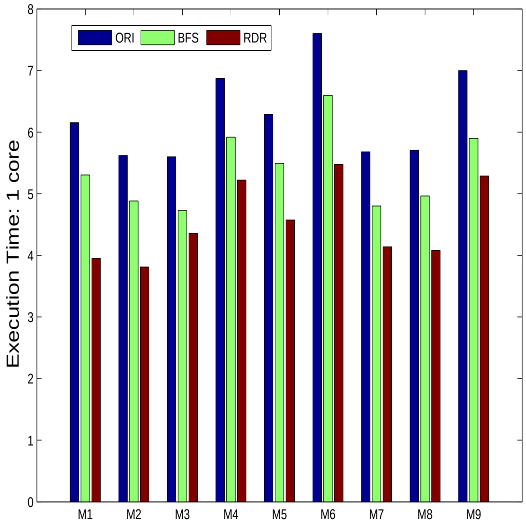


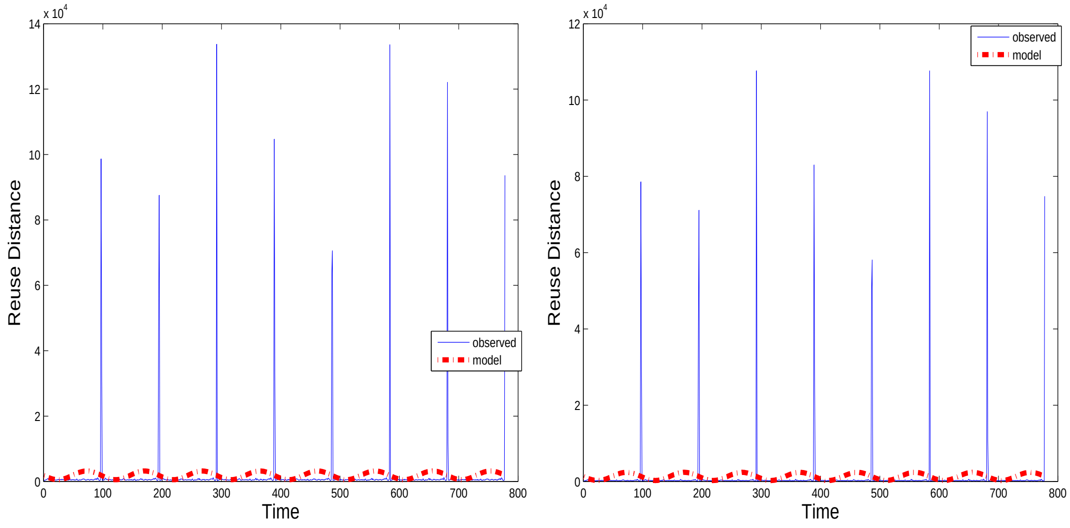
Figure 5.12: Execution time results for Laplacian mesh smoothing when reuse distance reducing ordering was applied. Reuse distance reducing ordering is 38.57% faster than original ordering.

Figures 5.13∼5.21 show the observed reuse distance profiles for original, BFS, and reuse distance reducing orderings of the Laplacian mesh smoothing with carabiner, crake, dialog, lake, riverflow, ocean, stress, valve, and wrench meshes. Reuse distance models using our methodology developed in Chapter 3 are also provided in the Figures. Reuse distances were significantly reduced when we applied our reuse distance reducing ordering to the Laplacian mesh smoothing, thus, less execution times were required.

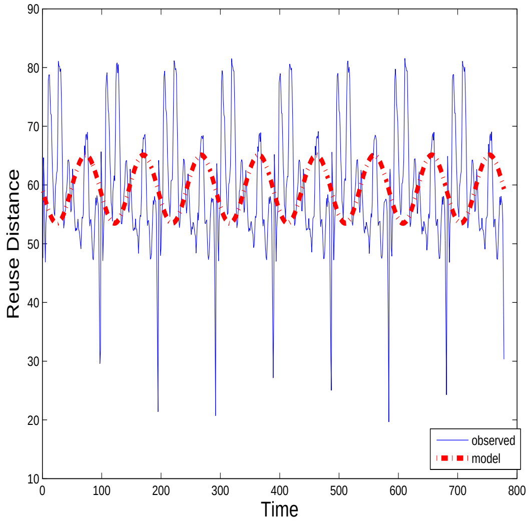
Table 5.3 shows reuse distance models we obtained using our simple periodic function we developed in Chapter 3, $\alpha + \beta \sin(2\pi t\phi)$, where $\phi = 8$ was obtained in this test. These reuse distance models will be used for predicting cache miss rates for Laplacian mesh smoothing application.

5.2.3.3 Cache Performance

We collected the cache performance counter using PAPI 5.1.0.2 [39] to compute cache performance improvement when reuse distance reducing ordering was applied to Laplacian mesh smoothing. Figures 5.22(a)(b) and (c) show the L1, L2, and L3 cache performance for Laplacian mesh smoothing running on a single core



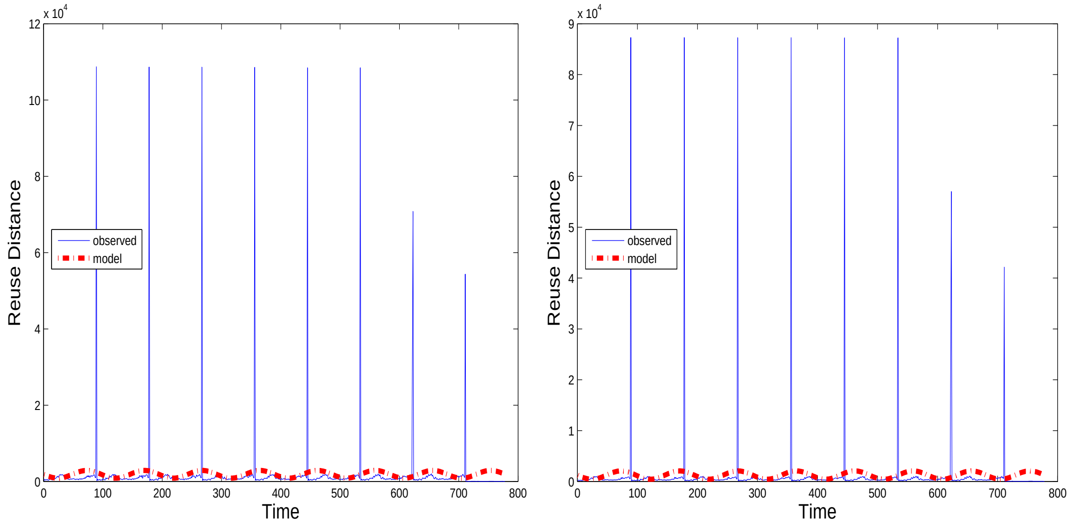
(a) Original ordering (b) BFS



(c) Reuse distance reducing ordering

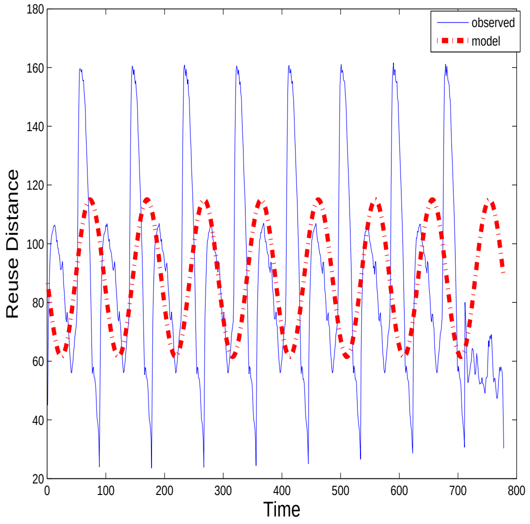
Figure 5.13: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with carabiner mesh.

when reuse distance reducing ordering was applied. After reordering applied to Laplacian mesh smoothing, cache miss rates decreased from 0.67% to 0.5% for L1, from 44.7% to 17.4% for L2, and from 47.7% to 26.1% for L3 on average. Similar cache performance is obtained when the Laplacian mesh smoothing runs on multicores (32 cores).



(a) Original ordering

(b) BFS

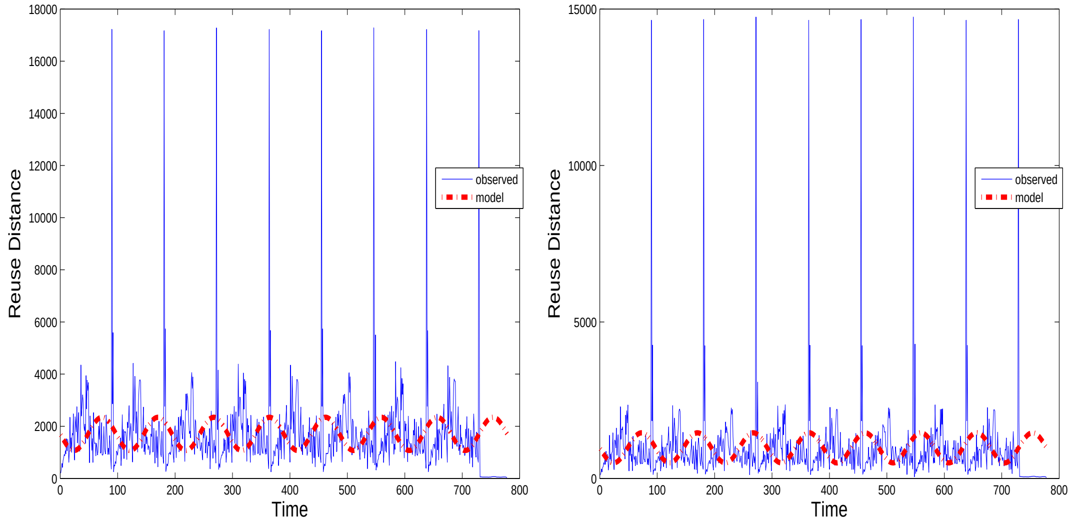


(c) Reuse distance reducing ordering

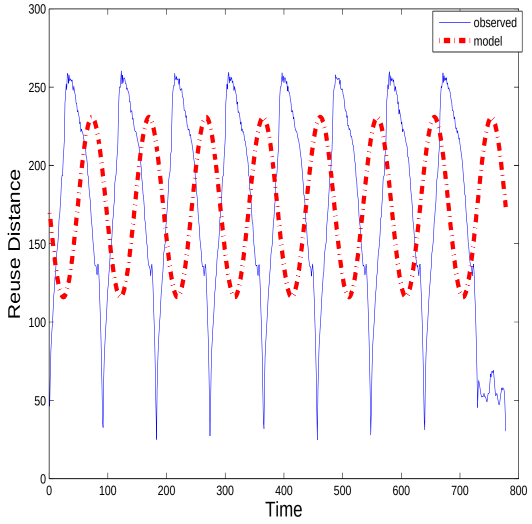
Figure 5.14: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with crake mesh.

5.2.3.4 Performance Scalability

We consider relative speedups per mesh on 1,2,4,8,16,24 and 32 cores. All speedup measurements are relative to serial baseline (Original ordering at 1 core). Aggregate results report geometric mean. We compute speedup relative to the serial baseline as $Speedup(ordering, p) = \frac{T_{ORI}(1)}{T_{ordering}(p)}$, where $T_{ordering}(p)$ is the execution



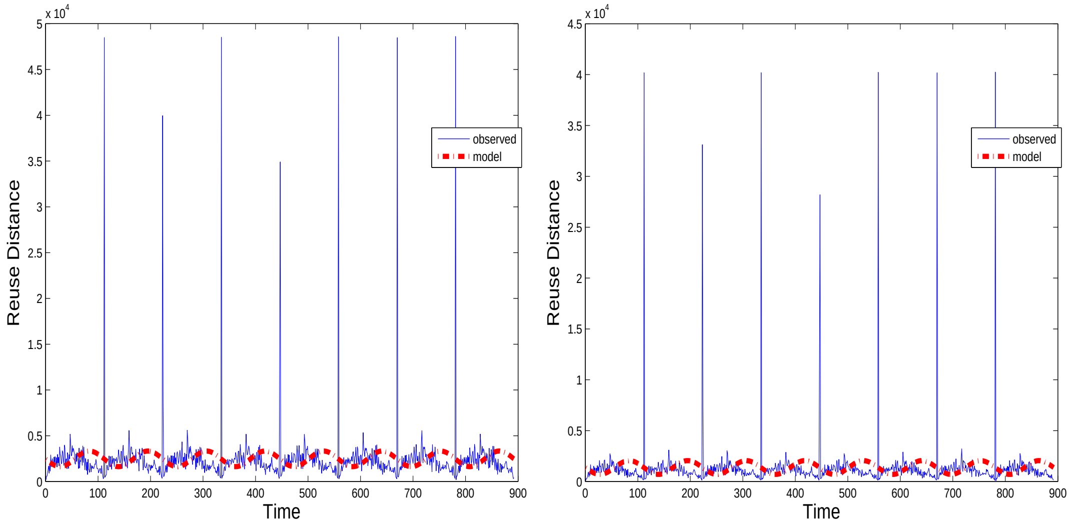
(a) Original ordering (b) BFS



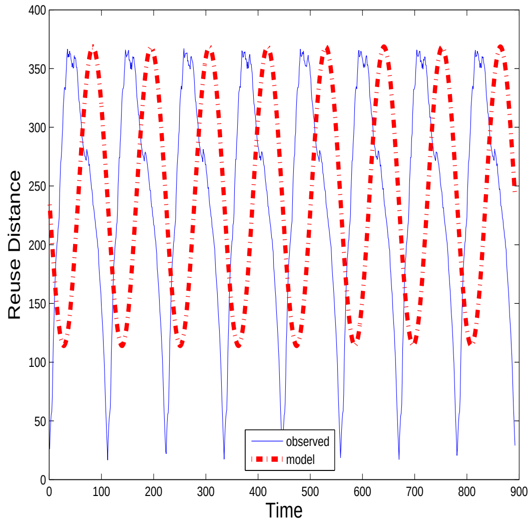
(c) Reuse distance reducing ordering

Figure 5.15: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with dialog mesh.

time with p cores, and *ordering* is either ORI, BFS, or RDR. Figures 5.23(a)˜(g) present the speedup results for each mesh at 1, 2, 4, 8, 16, 24, and 32 cores and Figure 5.23(h) shows mean speedup results of all aggregate meshes for all cores. Speedup of 75.32x with reuse distance reducing ordering over the serial baseline was obtained.



(a) Original ordering (b) BFS

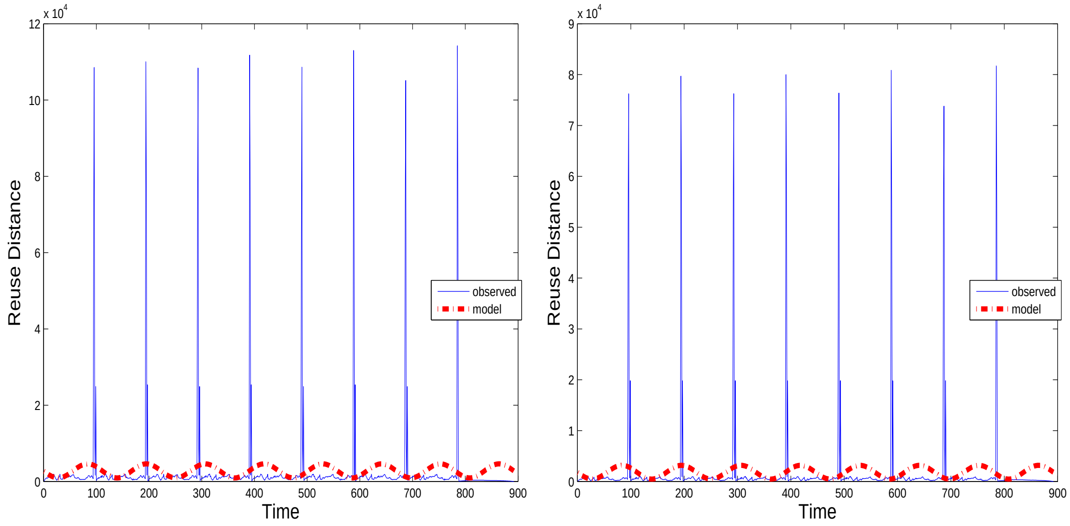


(c) Reuse distance reducing ordering

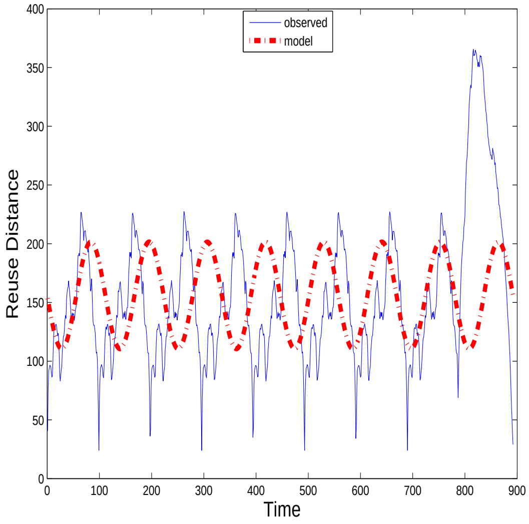
Figure 5.16: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with lake mesh.

5.2.4 Summary of Reuse Distance Reducing Ordering

We have developed and evaluate reuse distance reducing ordering for an irregular application, Laplacian mesh smoothing. We defined a reuse distance reducing ordering scheme that observes the initial mesh qualities for each vertex and reorders



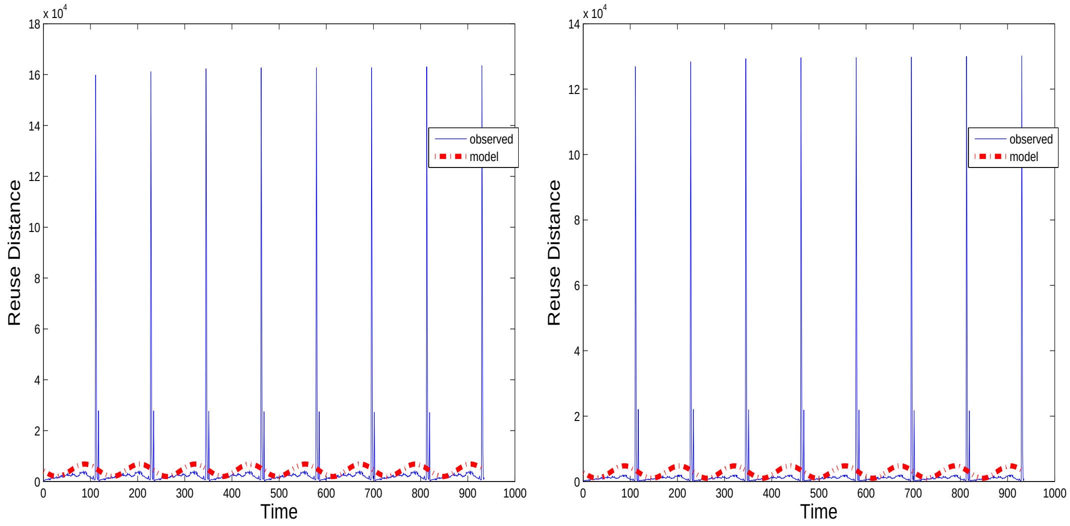
(a) Original ordering (b) BFS



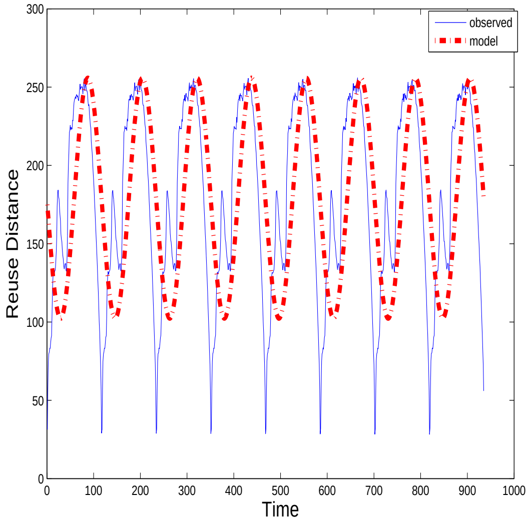
(c) Reuse distance reducing ordering

Figure 5.17: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with riverflow mesh.

the mesh based on the quality in ascending order to improve both temporal and spatial localities for memory accesses of Laplacian mesh smoothing. Additionally, we applied our predictive cache miss models developed in Chapter 3 and utilized to predict the cache performance of Laplacian mesh smoothing with difference reordered meshes. Our parameter estimation for these cache miss models were



(a) Original ordering (b) BFS

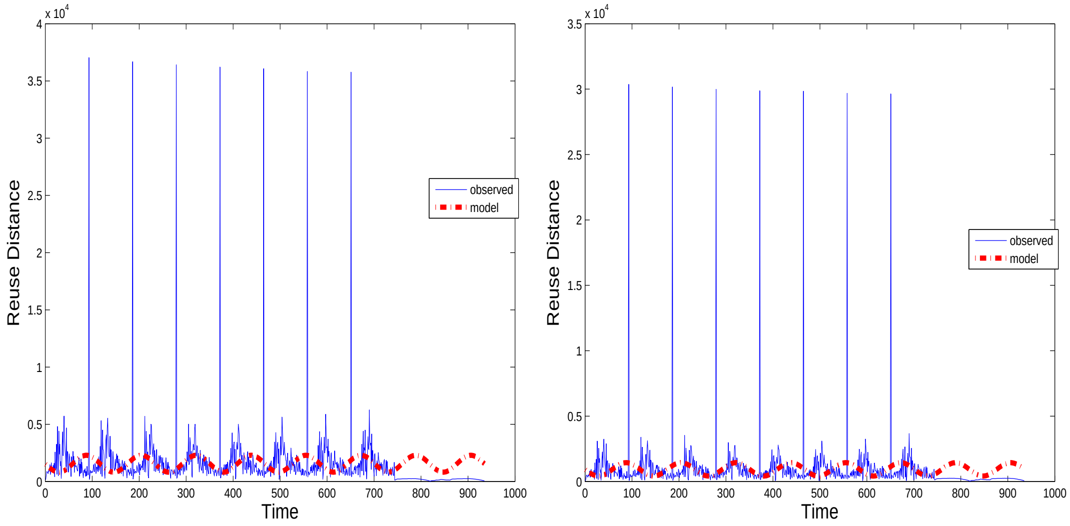


(c) Reuse distance reducing ordering

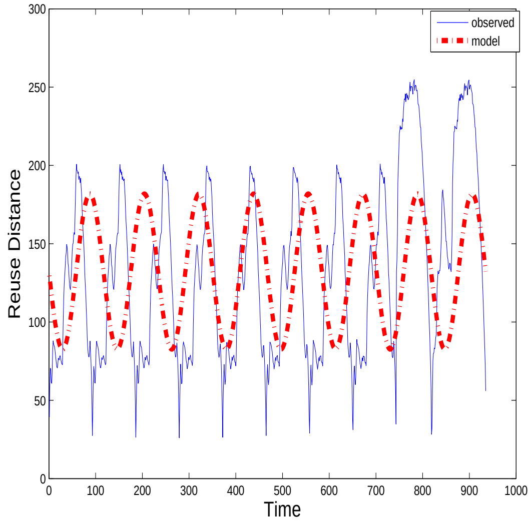
Figure 5.18: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with ocean mesh.

performed on Intel Westmere architecture system.

In summary, our experimental evaluation shows that when reuse distance reducing ordering was applied to Laplacian mesh smoothing, 38.58% of performance improvement was obtained. Cache miss rates decreased as well, from 0.67% to 0.5% for L1, from 44.7% to 17.3% for L2, and from 47.4% to 26.1% for L3 on average



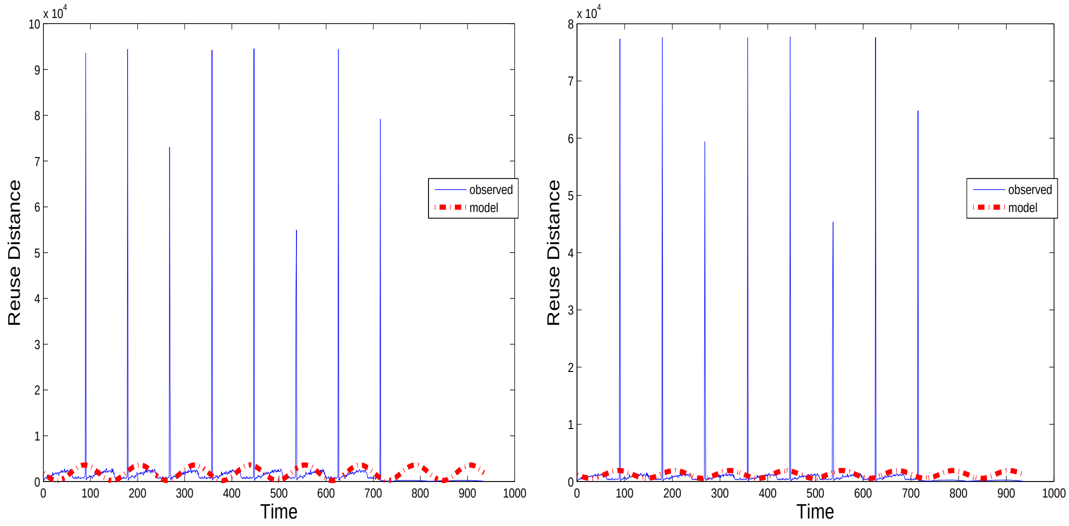
(a) Original ordering (b) BFS



(c) Reuse distance reducing ordering

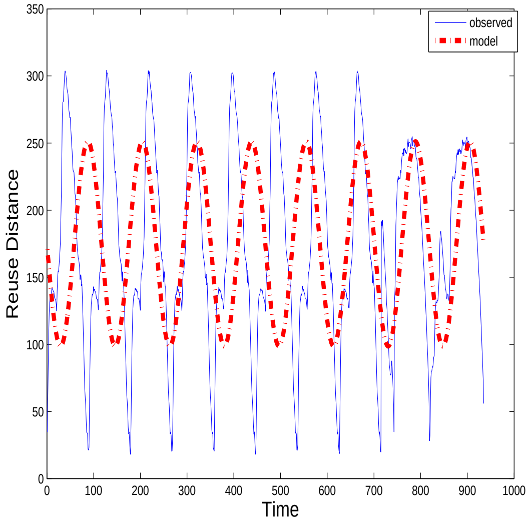
Figure 5.19: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with stress mesh.

when Laplacian mesh smoothing runs on a single core. Similar cache performance is obtained when we run the application on multicores (32 cores). We observed mean speedup of 75x with reuse distance reducing ordering when we scale up to 32 cores. This reuse distance reducing ordering will improve other mesh application performance such as mesh untangling mesh [40], constraint mesh smoothing [41],



(a) Original ordering

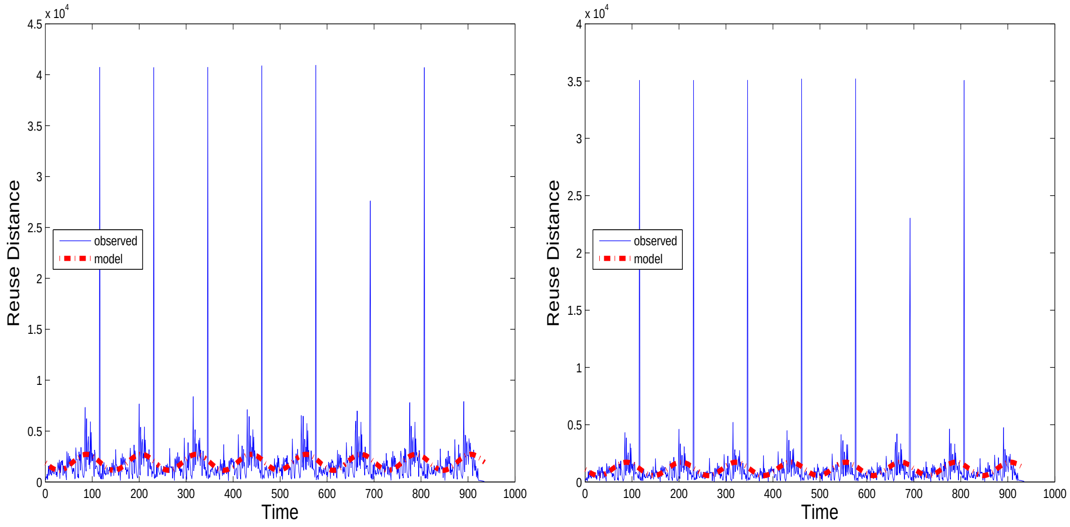
(b) BFS



(c) Reuse distance reducing ordering

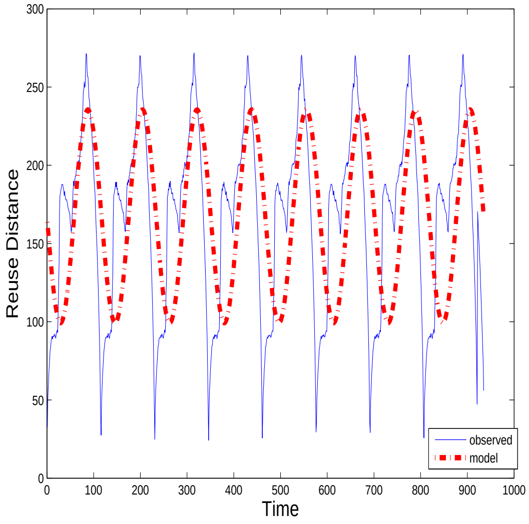
Figure 5.20: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with valve mesh.

and mesh swapping [42] .



(a) Original ordering

(b) BFS

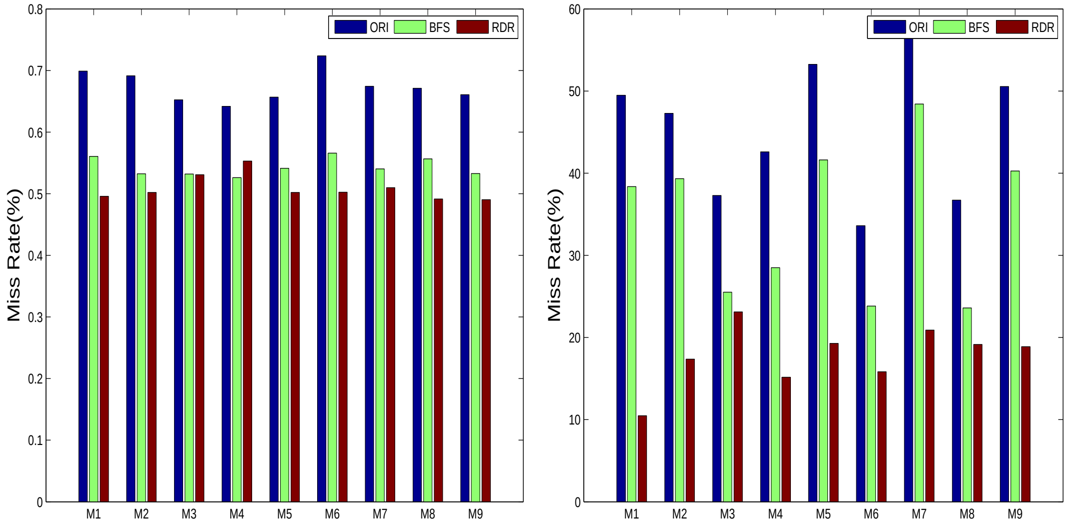


(c) Reuse distance reducing ordering

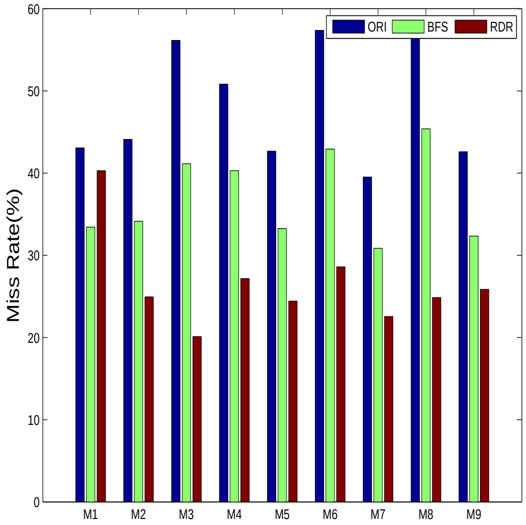
Figure 5.21: Observed reuse distance profiles and reuse distance models for (a) original ordering, (b) BFS ordering, and (c) reuse distance reducing ordering of the Laplacian mesh smoothing with wrench mesh.

$\alpha + \beta \sin(2\pi t\phi)$ Models			
		Reuse Distance	
	Ordering	α	β
$R_{carabiner}$	ORI	1896.46	1331.72
	BFS	1349.27	1068.06
	RDR	59.29	5.79
R_{crake}	ORI	1898.97	1012.62
	BFS	1272.41	816.08
	RDR	88.27	26.65
R_{dialog}	ORI	1708.97	635.26
	BFS	977.93	479.65
	RDR	173.35	57.27
R_{lake}	ORI	2478.62	849.39
	BFS	1379.73	675.88
	RDR	241.27	127.05
$R_{riverflow}$	ORI	2794.31	1839.15
	BFS	1835.05	1351.63
	RDR	156.01	45.33
R_{ocean}	ORI	4425.42	2448.97
	BFS	2906.54	1897.39
	RDR	179.05	76.61
R_{stress}	ORI	1557.93	735.61
	BFS	932.42	510.54
	RDR	132.22	49.51
R_{valve}	ORI	1919.92	1696.31
	BFS	1242.08	679.38
	RDR	174.87	76.04
R_{wrench}	ORI	1937.08	784.04
	BFS	1146.79	583.13
	RDR	167.38	68.02

Table 5.3: Reuse distance models of Laplacian mesh smoothing for carabiner, crake, dialog, lake, riverflow, ocean, stress, valve, and wrench meshes when original ordering, BFS ordering and reuse distance reducing ordering are applied.

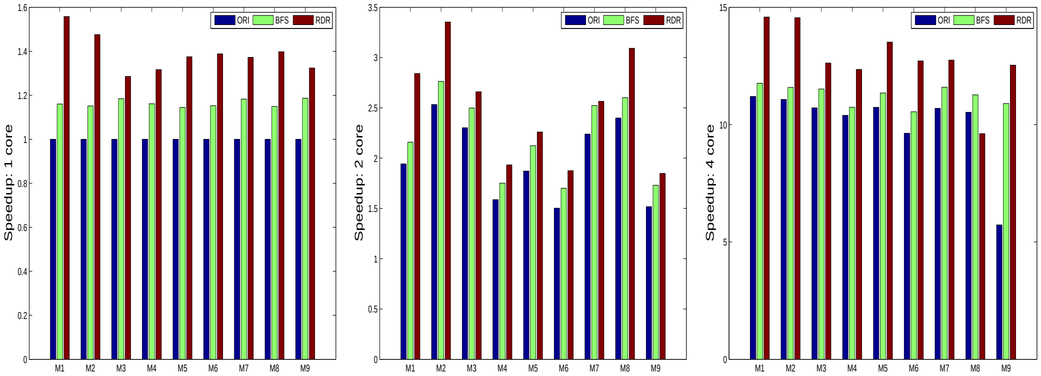


(a) L1 (b) L2

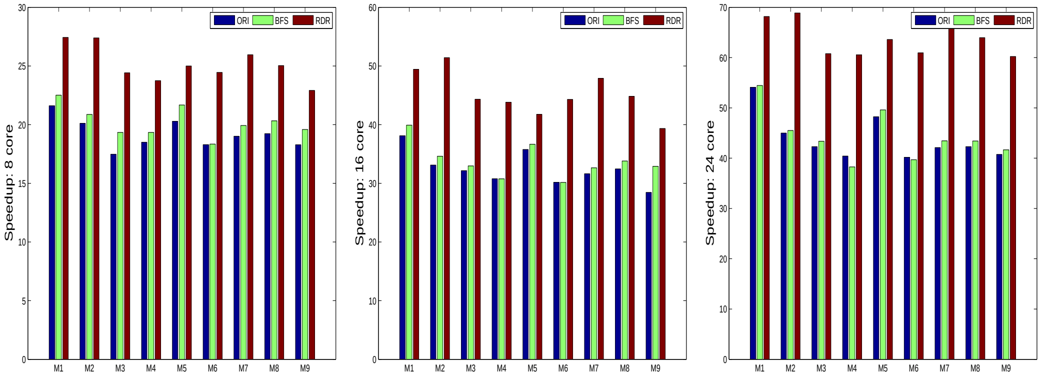


(c) L3

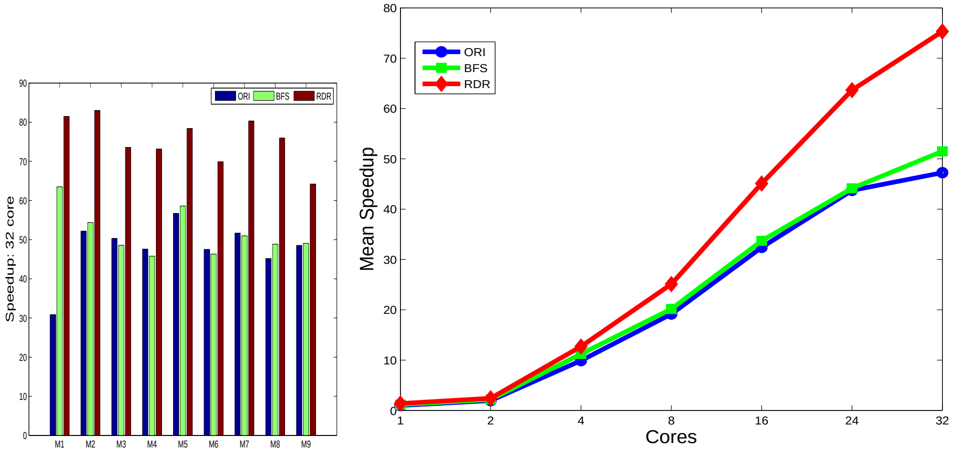
Figure 5.22: Cache performance results when reuse distance reducing ordering was applied to Laplacian mesh smoothing for (a) L1, (b) L2, and (c) L3. Cache miss rates decreased from 0.67% to 0.5% for L1, from 44.7% to 17.3% for L2, and from 47.4% to 26.1% for L3 on average.



(a) 1 core (b) 2 cores (c) 4 cores



(d) 8 core (e) 16 cores (f) 24 cores



(g) 32 cores (h) Mean speedup versus $T_{ORI}(1)$.

Figure 5.23: Observed speedup relative to the serial ORI baseline. Reuse distance reducing ordering (RDR) provides significant performance improvement.

Chapter 6

Conclusions and Future Work

This thesis considers the development of reuse distance models of irregular high performance computing (HPC) applications to enable the prediction of cache miss rates for a single application or pairs of co-scheduled applications. Further, we consider the utilization of these models for improving performance of HPC applications in the presence of interference due to co-scheduling of application pairs on multicore systems. Additionally, we develop and test a reuse distance reducing ordering to improve the performance of an irregular HPC application, namely, the Laplacian mesh smoothing application.

Our main contributions are based on our conjecture that the repeated basic block structure in scientific codes on HPC multicores would result in periodic behavior of their reuse distances. We test our conjecture to show that a simple $\alpha+\beta\sin(2\pi t\phi)$ model of the observed reuse distances for an application can capture the periodic behavior of the application. This reuse distance model is scaled to yield a model for predicting cache miss rates for the application using the linear model obtained by multiple runs of synthetic benchmark, *CodeP*. The proposed predictive cache miss models achieved prediction error rates as low as 2.01% on average.

Further, we derive a cache miss model for a certain application in the presence of interference. We conjecture that the periodic behavior for the application still exists when interfered by another co-scheduled application. We derive reuse distance profiles of an application when it is interfered by another co-scheduled application through linear combination of two applications' reuse distance models.

These predicted reuse distance profiles are then used to derive our predictive cache miss models. The parameters for our predictive cache miss models were estimated by regression with datasets obtained from a hundred runs of a simple synthetic code *CodeP* on the target system. When verifying these models using the NAS benchmarks and gem5 simulation, we observe error rates of less than 3.13% on average for a conventional multicore with a two-level cache hierarchy, and 3.98% for an NoC architecture. We show that when the application with low reuse distances is interfered by another application with high reuse distances, performance degradation is more prominent. This is due to the fact that the memory accesses of the application with lower reuse distances are easily interfered by other co-scheduled applications.

Using these predictive cache miss models, we select the best application pairs for co-scheduling to improve their performance. We rank the application pairs based on increasing cache miss rates due to co-scheduling and pick one application pair which shows the least performance degradation when co-scheduled. When we test our ranking system to NAS benchmarks on gem5 simulation, the predicted best application pairs in regard to least degradation in performance when co-scheduled, matched with observed performance. The worst application pairs which showed the most performance degradation due to co-scheduling also matched the observed results. The variations of cache miss rates for the remaining application pairs are not enough to allow for detecting their ranking precisely. We would like to see how sensitive to co-schedule for all application pairs using our ranking system results.

Additionally, using the reuse distance models we developed, we propose a reuse distance reducing ordering for improving performance of an irregular HPC application, Laplacian mesh smoothing. We reorder the meshes based on the initial mesh quality so that the vertex which has the worst quality can undergo Laplacian mesh smoothing before than other vertices. This reordering reduces the reuse distance of Laplacian mesh smoothing, and thus, improves the performance. Our experimental results show that when reuse distance reducing ordering is applied to Laplacian mesh smoothing, we gain 38.58% of performance improvement when running on a single core. In the experimental tests, our predictive cache miss models achieve prediction error rates as low as 5.36%. The thesis shows that the overall cache performance is improved when reuse distance reducing ordering is applied to Laplacian mesh smoothing as compared to original ordering. Moreover, 75x of mean speedup is obtained when scaling up to 32 cores.

In the near future, we plan to extend our predictive reuse distance and cache miss models to include more complex periodic behavior of HPC applications. We also plan to expand our co-scheduling approach for two applications, to three co-scheduled applications. This can be achieved by composing the model for a pair with the model for a single application. However, more sophisticated parameter estimation will be required to control error rates to find such approximations. Further, we expect to improve the ranking results to meet all ranking orders between predicted and observed performance. We conjecture that considering distinct reuse distances for modeling memory behavior will provide us better performance of our predictive cache miss models.

Bibliography

[1] Shewchuk, J. (1996) "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148, Springer-Verlag Lecture Notes in Computer Science, pp. 203–222.

[2] Wu, M., M. Zhao, and D. Yeung (2013) "Studying Multicore Processor Scaling via Reuse Distance Analysis," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 499–510.

[3] Donggarra, J., H. Meuer, and E. Strohmaier (1997) "TOP500 Supercomputer sites," in *supercomputer*, vol. 13, pp. 89–111.

[4] Wulf, W. A. and S. A. McKee (1995) "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, **23**(1), pp. 20–24.
URL `http://doi.acm.org/10.1145/216585.216588`

[5] Feo, J., O. Villa, A. Tumeo, and S. Secchi (2011) "Irregular Applications: Architectures &#38; Algorithms," in *Proceedings of the First Workshop on Irregular Applications: Architectures and Algorithm*, IAAA '11, ACM, New York, NY, USA, pp. 1–2.
URL `http://doi.acm.org/10.1145/2089142.2089144`

[6] Singhal, R. (2008) "Inside Intel Next Generation Nehalem microarchitecture," in *Hot Chips*, vol. 20.

[7] Conway, P., N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes (2010) "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *Micro, IEEE*, **30**(2), pp. 16–29.

[8] Frasca, M., A. Chatterjee, and P. Raghavan (2011) "Can models of scientific software-hardware interactions be predictive?" *Procedia Computer Science*, **4**(0), pp. 322 – 331, proceedings of the International Conference on

Computational Science, {ICCS} 2011.
URL `http://www.sciencedirect.com/science/article/pii/S1877050911000925`

[9] Smith, S. (1997) *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, San Diego, CA, USA.

[10] Ding, C. and Y. Zhong (2003) "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Desing and Implementation*, ACM, pp. 245–257.

[11] Schuff, D. L., M. Kulkarni, and V. S. Pai (2010) "Accelerating multicore reuse distance analysis with sampling and parallelization," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ACM, pp. 53–64.

[12] Jiang, Y., E. Z. Zhang, K. Tian, and X. Shen (2010) "Is reuse distance applicable to data locality analysis on chip multiprocessor?" in *Proceedings of the International Conference on Compiler Construction*, Springer, pp. 264–282.

[13] Xu, C., X. Chen, R. P. Dick, and Z. M. Mao (2010) "Cache contention and application performance prediction for multi-core systems," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 203–212.

[14] Wu, M. J. and D. Yeung (2011) "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, pp. 264–275.

[15] Chandra, D., F. Guo, S. Kim, and Y. Solihin (2005) "Predicting interthread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, pp. 340–351.

[16] Berg, E., H. Zeffer, and E. Hagersten (2005) "A Statistical Multiprocessor Cache Model," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, IEEE Computer Society, pp. 89–99.

[17] Zhuravlev, S., S. Blagodurov, and A. Fedorova (2010) "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pp. 129–142.

[18] Mars, J., L. Tang, and M. Soffa (2011) "Directly Characterizing Cross Core Interference Through Contention Synthesis," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pp. 167–176.

[19] Tang, L., J. Mars, and M. Soffa (2011) "Contentiousness vs. Sensitivity: Improving Contention Aware Runtime Systems on Multicore Architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 12–21.

[20] Knauerhase, R., P. Brett, B. Hohlt, L. Tong, and S. Hahn (2008) "Using OS Observations to Improve Performance in Multicore Systems," *Micro, IEEE*, **28**(3), pp. 54–66.

[21] Zhao, J., H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang (2013) "An Empirical Model for Predicting Cross-core Performance Interference on Multicore Processors," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 201–212.

[22] Sherwood, T., E. Perelman, and B. Calder (2001) "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pp. 73–14.

[23] Shen, X., Y. Zhong, and C. Ding (2005) "Phase-Based Miss Rate Prediction Across Program Inputs," in *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing*, pp. 42–55.

[24] Freitag, L., P. Knupp, T. Munson, and S. Shontz (2002) "A comparison of optimization software for mesh shape-quality improvement problems," in *Proc. of the* 11^{th} *International Meshing Roundtable*, Sandia National Laboratories, pp. 29–40.

[25] Park, J. and S. M. Shontz (2011) "An Alternating Mesh Quality Metric Scheme for Efficient Mesh Quality Improvement," *Procedia Computer Science*, **4**(0), pp. 292 – 301.

[26] Knupp, P. M. (2001) "Algebraic Mesh Quality Metrics," *SIAM J. Sci. Comput.*, **23**(1), pp. 193–218.

[27] Munson, T. (2007) "Mesh shape-quality optimization using the inverse mean-ratio metric," *Math. Program.*, **110**, pp. 506–590.

[28] Strout, M. M. and P. D. Hovland (2004) "Metrics and Models for Reordering Transformations," in *Proceedings of the 2004 Workshop on Memory System Performance*, ACM, pp. 23–34.

[29] T. Munson, P. H. (2005) "The FeasNewt benchmark," in *Proceedings of the IEEE International Workload Characterization Symposium*, pp. 150–154.

[30] Shontz, S. and P. Knupp (2008) "The Effect of Vertex Reordering on 2D Local Mesh Optimization Efficiency," in *Proceedings of the 17th International Meshing Roundtable* (R. Garimella, ed.), Springer Berlin Heidelberg, pp. 107–124.

[31] Park, J., P. Knupp, and S. Shontz (2010) "Static vertex reordering schemes for local mesh qualiyt improvement," in *Technical report, Sandia National Laboratories*.

[32] Beyls, K. and E.D'Hollander (2001) "Reuse distance as a metric for cache behavior," in *Proceedings of the Parallel and Distributed Computing and System*, pp. 617–662.

[33] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, **39**(2), pp. 1–7.

[34] Bailey, D., E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga (1991) "The NAS parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pp. 158–165.

[35] Mellor-Crummey, J., D. Whalley, and K. Kennedy (2001) "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings," *Int. J. Parallel Program.*, **29**(3), pp. 217–247.

[36] Shewchuk, J. R. (2002) *What Is a Good Linear Finite Element? - Interpolation, Conditioning, Anisotropy, and Quality Measures*, *Tech. rep.*, In Proc. of the 11th International Meshing Roundtable.

[37] Zhou, M., O. Sahni, M. S. Shephard, C. D. Carothers, and K. E. Jansen (2010) "Adjacency-based Data Reordering Algorithm for Acceleration of Finite Element Computations," *Sci. Program.*, **18**(2), pp. 107–123.

[38] BREWER, M., L. FREITAG, P. KNUPP, T. LEURENT, and D. MELANDER (2003) "The Mesquite Mesh Quality Improvement Toolkit," in *Proc. of the* 12^{th} *International Meshing Roundtable*, Sandia National Laboratories, pp. 239–250.

[39] MUCCI, P. J., S. BROWNE, C. DEANE, and G. HO (1999) "PAPI: A portable interface to hardware performance counters," in *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10.

[40] FREITAG, L. and P. PLASSMANN (2000) "Local optimization-based simplicial mesh untangling and improvement," *Int. J. Numer. Math. Eng.*, **49**, pp. 109–125.

[41] PARTHASARATHY, V. and S. KODIYALAM (1991) "A constrained optimization approach to finite element mesh smoothing," *Finite Elements in Analysis and Design*, **9**, pp. 309–320.

[42] FREITAG, L. and C. OLLIVIER (1997) "Tetrahedral mesh improvement using swapping and smoothing," *Int. J. Num. Meth. Eng.*, **40**, pp. 3979–4002.

Vita

Jeonghyung Park

Jeonghyung Park received her B.S. degree in Computer Science from Korea University in Korea. In 2007, she moved to U.S. and enrolled in the M.S program in Computer Science and Engineering at the Pennsylvania State University. In August 2009, she got her M.S. degree and continued her Ph.D study in the Computer Science and Engineering at the Pennsylvania State University. Since 2009 she has been employed in the Computer Science and Engineering department of the Pennsylvania State University as a teaching assistant and a research assistant. Her research interests are scalable parallel algorithms, performance optimization with numerical methods, and large scale data analysis.