IMPROVING FAIRNESS AND THROUGHPUT OF A CMP PROCESSOR BY OPTIMIZING THE UTILIZATION OF THE LAST LEVEL SHARED CACHE IN REAL-TIME USING A CONSTRAINED-EXTENDED KALMAN FILTER

by

Ashish Panday

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

Charlotte

2017

Approved by:

____
Dr. Yogendra P. Kakad

____
Dr. Arindam Mukherjee

____
Dr. James M. Conrad

____
Dr. Srinivas Akella

ABSTRACT

ASHISH PANDAY. Improving fairness and throughput of a CMP processor by optimizing the utilization of the last level shared cache in real-time using a constrained-extended kalman filter (Under the direction of DR. YOGENDRA KAKAD)

Cache Partitioning is a technique which maps the data pertaining to each core to a corresponding partition in the cache memory exclusively. Cache partitioning has been shown to improve performance metrics like fairness and throughput in most cases by eliminating inter-core conflict misses in shared cache of modern multi-core processors. Recently real-time management of cache partitioning is being studied to accommodate the variations in the intrinsic behavior of threads running the cores; thereby further improving cache utilization.

This dissertation presents a novel scheme for real-time management of cache partitioning using a constrained-extended Kalman filter. This approach is named Predictive Model based Cache Partitioning (PMCP). The design of PMCP utilizes an evolving approximate model of the nonlinear relationship between observed performance of each thread and the allocated cache partition size. The Gradient Projection method is used to model the performance model which predicts the next partition configuration. It also utilizes the history of the transient behaviors of the active threads to predict the cache partitioning for a low computation and space overhead. The key contribution of the research is that the cache performance curves are generated dynamically and is used to predict partitioning strategies such that cache utilization is optimized. PMCP is evaluated on the GEM5 simulator using the SPEC CPU2006 benchmarks. The results show that the throughput of the system improves by up to 35% with PMCP over shared cache.

DEDICATION

This dissertation is dedicated to my advisor for the last 8 years, Dr. Bharatkumar S. Joshi. He was an Associate Professor and the Graduate Coordinator of the Department of Electrical and Computer Engineering at UNC Charlotte. Unfortunately Dr. Joshi passed away on March 7, 2016.

Dr. Joshi was a highly dedicated and sincere person who cared immensely for all his students and his profession as a professor. His famous words “I am not here to make mediocre engineers” is an example of his dedication which resounds with many of us.

Personally, the inspiration for this research came from Dr. Joshi as he wanted to explore the use of *Control Theory* concepts to optimize performance of computers. It was an honor to have the opportunity to research on this topic with him. As my advisor, Dr. Joshi gave me the freedom and encouragement to explore various approaches that I could use to develop my PhD research. But he was always available to analyze my ideas with me.

Dr. Joshi will always be a role-model for me, professionally and personally, and I will forever be indebted to him for all that he has taught me.

ACKNOWLEDGEMENTS

My thanks and appreciation to Dr. Yogendra P. Kakad for agreeing to accept me as his advisee and persevering with me as I finish my PhD and write my dissertation. I am grateful for the numerous conversations and brainstorming sessions he conducted with me so patiently.

The members of my committee, Dr. Arindam Mukherjee, Dr. James M. Conrad and Dr. Srinivas Akella, have generously given their time and expertise to better my work. I thank them for their contribution and their good-natured support.

I am grateful for the staff members, present and past, of the Electrical and Computer Engineering Department, Stephanie LaClair, Sharron Green, Nikki Redman, Jerri Price, Eddie Hill, Michelle Wallace and Jerena McNeil for all their support.

A special thank you to the Late Dr. Joshua Stokell, Dr. Bushra Khan and Dr. Omera Matoe for the special friendship we shared over the years. I must express my gratitude to my dear friends Jerry Zacharias, Sam Shue, Jeremy Sabo, Lauren Johnson, John Wiley Thompson, Reshmi Mitra, Shakti Shankar, Kushal Datta and Sushma S. Murthy whose friendship, hospitality, knowledge, and wisdom have supported, enlightened, and entertained me over the many years of our friendship. I must acknowledge as well the many friends, colleagues, students and professors who have assisted, advised, and supported my research over the years.

Lastly, the most important people that I’d like to thank my parents, Praveen K. Panday and Neelam Panday, my sister, Natasha Seth, and my brother-in-law, Andy Seth whose unconditional whole-hearted love and support gave me the strength to reach the finish line.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IC	Integrated Circuits
SoC	System on Chip
SiP	System in Package
ITRS	International Technology Roadmap for Semiconductors
IPC	Instructions Per Cycle
QoS	Quality of Service
PMCP	Predictive Model based Cache Partitioning
C-EKF	Constrained - Extended Kalman Filter
TQ	Time Quantum
UCP	Utility based Cache Partitioning
EKF	Extended Kalman Filter
LRU	Least Recently Used
CPU	Central Processing Unit
OS	Operating System
SE	Syscall Emulation
FS	Full System emulation
SLICC	Specification Language for Implementing Cache Coherency
ISA	Instruction Set Architecture
SMU	System Monitoring Unit
HLTP	High Level Transition Period
HLSP	High Level Steady-state Period

$IPCmt_i$	Instructions per cycle multi-threaded (thread *i*)
$IPCst_i$	Instructions per cycle single-threaded (thread *i*)
GP	Gradient Projection

1. INTRODUCTION

Since the advent of computers, the semiconductor industry has been guided by Moore's Law which states that the density of transistors on a chip would double every two years. Consequently the transistor size got smaller and the speed with which the transistors switched from on state to off state also improved. Simultaneously manufacturing larger, more complex integrated circuits using standardized building blocks became cheaper for the semiconductor industry which was then in complete control of the rate of improvement of technology. During the 60s, 70s and 80s, scaling the transistor density directly improved the performance of the system at a very fast rate. In addition, the speed at which the transistors operated was complemented by the speed at which the memory operated.

The last two decades also brought several new paradigms to improve the performance of the system in the form of integrated circuits (ICs) like System on Chip (SoC), System in Package (SiP) and multicore processors. Integrated Circuits allowed cost effective integration of simple building blocks used in the design of several systems. Although these features improved the performance of the system and are currently driving innovation, it has, in many cases, reached the thermal and power limits on these devices. Still the transistor count is keeping up with Moore's Law but it is practically not possible to conjunctly improve the frequency at which the transistors function due to physical limitations on power dissipation and thermal thresholds. In 2015, International Technology Roadmap for Semiconductors (ITRS) [23] presented the roadmap for the future of transistor technology. Several companies have announced that since the horizontal space is limited, efforts are being made to explore the vertical dimension. Even though such a change in paradigm can improve transistor density, power still remains a critical factor and it is predicted that technology will hit a similar limit in the near future. Wu et al. complemented this philosophy [24] and presented a consolidated figure (Figure 1) where the past, the present and the future of the transistor technology is mapped. It is clear that we are at the precipice of reducing the transistors size to increase transistor density in a cost effective way while still maintaining the desired physical properties.

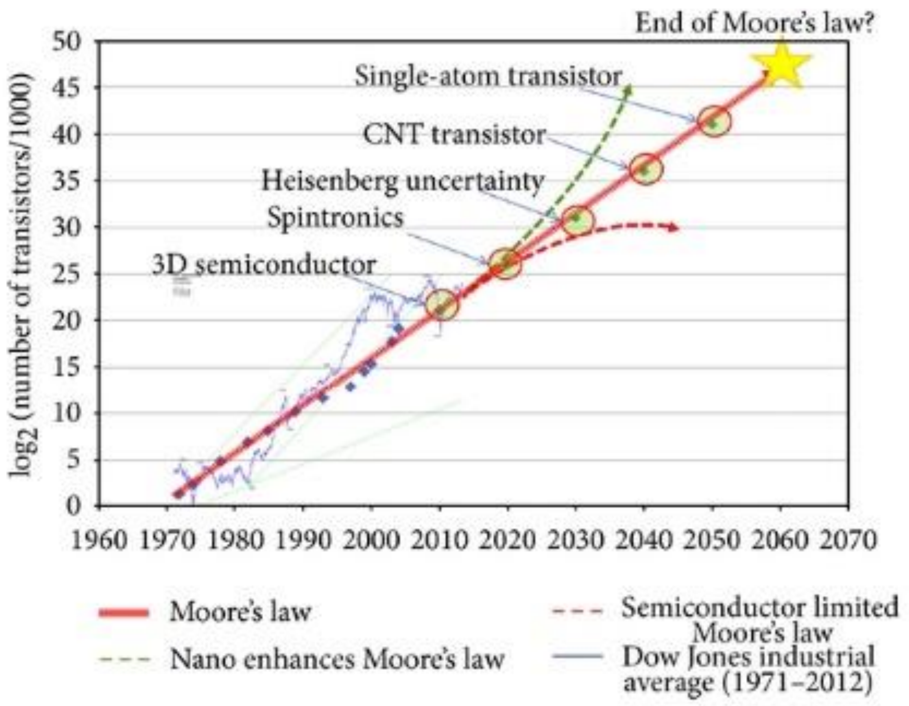


Figure 1: Moore's Law history, future, limited factors, and Nanotechnology-Enhance factors for Moore's Law and compared with Dow Jones industrial average in the same period of time (1971–2011). [24]

Improving the transistor technology which in turn improved the processors' performance in the past. But there are several other factors that affect the performance of the system like memory, bandwidth, interconnect bus and other peripheral devices. This research is focused on optimizing the utilization of these resources specifically the last level shared cache memory. Further discussions in this chapter will therefore be directed towards the effects of memory on performance of the system. It has long been known that it is in fact memory that drives Moore’s Law. The rate at which memory operates directly affects the number of operations that can be performed by the processing unit. Without the improvement in memory it is very difficult to improve the processor performance even if the transistor density is improved. In addition, typically, memory performs significantly slower than the processor.

Historically, the improvement in transistor technology also improved the memory performance and new computer systems were complemented with a new memory technology. New configurations of memory hierarchy and utilization of different types of memory technology also contributed towards a significant improvement in the performance of the system. Indeed, advanced processors and the complementing memory organization and technology has improved the overall performance of the systems. However, due to the limits of Moore’s Law it is challenging to improve the processor and memory technology in a way that is beneficial to the semiconductor industry.

An alternate way to improve system performance, which does not simply rely on improving the transistor technology, can stem from improving the utilization of the existing resources available to the processor. For example, modern multi-core processors, which allow large number of threads to run in parallel, share the limited lower level cache in the memory hierarchy which can cause conflicts between threads resulting in the eviction of useful data of the competing threads. This may lead to degradation of performance metrics for throughput like Instructions Per Cycle (IPC). In some cases where a few competing threads have significantly large footprints in the shared cache may degrade performance of other competing threads [20, 21] which in turn affects the Quality of Service (QoS) and fairness performance metrics like fair-speedup. Perhaps, a restriction policy that can limit the amount of footprint a thread can have in the shared space while organizing the space itself can maintain the QoS for each threads in a judicial manner and improve the overall performance of the processor.

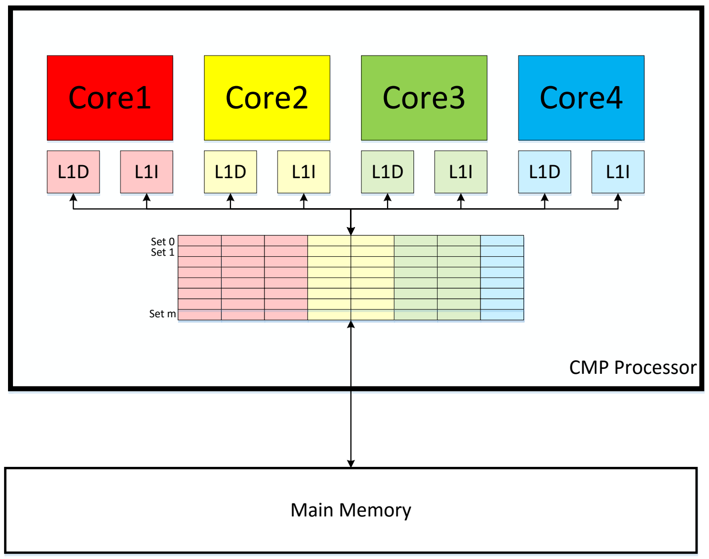


Figure 2: Organization of partitions in a cache partitioned using PMCP

Cache partitioning has been shown to improve the performance of the overall processor in such cases. In cache partitioning, the shared cache space is typically divided into partitions and each thread’s data is mapped to a specific partition exclusively to avoid conflict misses. By deciding the size of the partitions judiciously the QoS can also be improved considerably. Consider a processor with 4 cores and a single threaded process running on each core. Assume that each core has a local L1 cache and all cores share an n-way set associative L2 cache (last level). Assume that cache partitioning was performed on the shared L2 cache. Figure 2 elaborates on this example where the L2 cache space is divided into 4 equal partitions, one for each core. Note that there is one physical shared L2 cache but the cache space is virtually divided into partitions and the data from the threads are organized in their designated partitions.

By assigning exclusive access rights to the threads only to certain partitions, cache partitioning facilitates fairer distribution of resources between the threads. Each thread's data is mapped to a specific partition; thereby eliminating conflict misses. Threads with large strides in data access have a higher probability of exploiting temporal locality since their data is maintained in their assigned partition. However, threads that require larger space to exploit spatial locality may be penalized because of restricted access to only the assigned partitions and not the entire cache space. A well-designed cache partitioning scheme accommodates these trade-off in order to improve the overall system performance.

The primary goal of this research work is to explore the possibility of using *System Theory* to design such a scheme. The shared cache is viewed as an open-loop system whose inputs are the set of values that represent each thread's partition size and the outputs are the number of hits and misses of each core. It is assumed that the system has the ability to identify the inputs and measure the outputs in real-time. The document presents the proposed scheme to close the loop and update the inputs based on measured outputs in real-time. The existing cache partitioning schemes that utilize feedback control are limited by the lack of accuracy in either the state-transition models, which predict the partition sizes, or the performance models that relate size of the available cache to performance metrics (like miss-rates). Some other partitioning schemes are dependent on either prior knowledge about the applications' data access patterns or defined targeted performances. The method described in this research, which is named as Predictive Model based Cache Partitioning (PMCP), provides a unique way to predict partition sizes for each thread with simple models without requiring any knowledge of the applications running on the system.

PMCP utilizes a Constrained-Extended Kalman Filter (C-EKF) as a state estimator to estimate the partition sizes for each thread. Estimators are part of System Theory and have been used to incorporate self-management and on-line estimation in many domains by estimating the value of an unknown parameter using statistical models and measured inputs or outputs. Through C-EKF, PMCP first estimates the state of the system (last level cache) based on *state-transition model*, followed by updating the estimates based on observations. The update process refines the estimates based on weighted difference between a measured miss-rate and miss-rate evaluated using a *performance model*. The weights are decided by evaluating the covariance of the errors in the state-transition model, performance model and overall accuracy of the state estimator. The miss-rates are measured at regular intervals referred as a Time Quantum (TQ) in this document. Once the miss-rates are measured, PMCP estimates the new state of the system iteratively and is expected to converge to a steady state as soon as possible. Only when the steady state is achieved, is the new partitioning applied to the shared cache space. The system runs with the new set of partitions until the end of the current TQ. Miss-rates are measured again at the start of the next contiguous TQ and PMCP estimates the next partitioning scheme based on current partitioning scheme (state from the previous TQ), measured performance (current miss-rates) and state-transition weights (error covariance matrices).

It is also observed that, if the threads do not change their access patterns throughout the experiment, the steady state values over several TQs experience a convergence as well to an overall stable state. With good state-transition and performance models, PMCP has the potential to reach a well predicted overall steady state which partitions the cache space such that cache utilization is optimized.

PMCP is evaluated on the GEM5 simulator with the SPEC CPU2006 benchmark. The evaluations with a simple state-transition and performance model yield a maximum of 35.35% improvement in throughput and 5.5% improvement in fair-speedup when compared to the throughput of a processor with standard shared cache. The simplicity of the models allows PMCP to accommodate variations in the models of the applications that can run simultaneously on the system. However, certain empirical evaluations had to be included to guide PMCP through the state-space. An extension to PMCP is proposed with a performance model that evolves at run-time. In the upgraded version, the performance model is evaluated at run-time based on measurements using *Gradient Projection* with fixed boundaries on cache partition size for each thread and miss-rate of each thread.

The rest of the document is organized as follows. In Chapter 2, detailed discussions on related researches is presented. A brief overview of the construction of the Kalman filter that leads to the construction of EKF and the C-EKF are discussed in Chapter 3. In Chapter 4, PMCP's framework is presented and summarized in Chapter 5. Chapter 6 details the simulations carried out in MATLAB to validate PMCP. The GEM5 simulator is introduced in Chapter 7 and the evaluation of PMCP on GEM5 is discussed on Chapter 8. Chapter 9 presents the results of PMCP with the simple state-transition and performance models while chapter 10 expands on how the performance model can be improved using gradient projection method. The entire research work is concluded in Chapter 11 and the future direction of the research work is discussed.

2. RELATED WORK

Cache partitioning was first introduced by Stone et al. [12] in 1992 where he discusses the difference in the access patterns of instruction and data streams that later formed the inspiration for the split L1 data and instruction cache. Stone et al. also proposed a greedy algorithm to partition the cache in cases where multiple processes run on the processors. The greedy algorithm has been shown to be optimal if the utility curves are convex. But in cases where the utility curves are non-convex, the algorithm can have pathological effects.

Several cache partitioning schemes were proposed after Stone to improve throughput, fairness and QoS [1-12, 18] including course grain partitioning techniques like Way Partitioning [11] and software based techniques like virtual memory and page coloring [5, 6]. Way Partitioning is simple, but it supports a limited number of coarsely-sized partitions. Virtual memory and page coloring have the advantage of being software based techniques but the process of remapping is time consuming as it involves copying (recoloring) of physical pages.

Dynamic cache partitioning was first investigated by Suh et al. [18] where they propose the use of a variant of the greedy algorithm, similar to the one introduced by Stone et al. [12]. Suh's algorithm explores all non-convex points of the miss-rate curve of the applications to minimize the total number of misses. However, an application's miss-rate curve can have several non-convex points and, in a multicore processor with several threads, the number of non-convex points can increase significantly. To address this limitation Qureshi et al. [18], introduced Utility-based Cache Partitioning (UCP) which uses the *Lookahead algorithm,* similar to the Suh's greedy algorithm. However, unlike Suh's greedy algorithm, the Lookahead algorithm evaluates the increase in marginal utility for all possible number of allocable ways. So if the cache is an *N*-way cache with *n* cores sharing the cache, UCP evaluates the utility of all *N* partition sizes for each core in parallel and chooses the partitioning with the best utility. Vantage [3] uses UCP as its primary partitioning algorithm. Jigsaw [1] uses the *Peekahead Algorithm* similar to the *Lookahead Algorithm* of UCP that performs in linear time. However these algorithms and techniques require some prior knowledge of the miss-rate curves of the applications running on the processor.

Instead of relying on the miss-rate curve, PriSM [2] uses a hill-climbing approach to partition the cache and reach targeted performance metrics by evaluating eviction probabilities for all threads. Such a method precludes adaptability since the processor is expected to reach a targeted performance. In addition, PriSM uses the same method as UCP for monitoring the cache performance. Apart from relying on prior knowledge of miss-rate curves, UCP is limited by the way it retains past information. UCP halves the measurement from the previous TQ and adds it to the current TQ's measurements. SHARP [22] comes the closest to PMCP since it relies on a feedback controller. But, like PriSM, it relies on targets defined for performance which renders it less adaptable. SHARP also limits the tested workload suite to applications with convex miss-rate curves.

Like SHARP, PMCP also relies on mathematical models and empirical measurements to evaluate efficient partitioning strategies using a feedback controller. However, PMCP does not compromise on the historical state transitional trajectories nor does it rely on miss-rate curves. PMCP is designed using a C-EKF that can estimate the sizes of the partitions (next state) based on measured cache misses and accesses in order to optimize the cache utilization thereby improving the system performance. *C-EKF is a recursive estimator that relies on statistical models as well as empirical observations to estimate the next state of the system.* Although there are many state estimators, there are several advantages of using C-EKF. C-EKF is a very light weight estimator whose navigation through the state-space is relatively inexpensive. The estimated next state of the system is dependent only on the current state of the system. Therefore C-EKF's space and time complexity are significantly reduced.

A good statistical model can estimate the next state of the system independently and accurately. However designing such a model for a cache is a monumental task. However, a simpler model's estimations complemented by updates based on real-time measurements can certainly improve the estimate of the nest state of the system over a few iterations. PMCP with C-EKF aims to make such real-time estimates iteratively such that cache utilization is pareto-optimal in each TQ thereby improving the overall performance like throughput and fairness.

2.1. Comparing PMCP with other cache partitioning methods

Table 1 compares the characteristics of the various cache partitioning schemes discussed so far. Since cache memory is a significantly fast system, it is important that the partitioning scheme does not have a large number of complex computations. It is also critical that the partitioning method is scalable as threads can activate/deactivate dynamically. The computations in all the methods are scalable with the number of threads and are light weight computations with PMCP and SHARP being relatively more compute intensive. The main advantage of PMCP over all the other methods is in the way PMCP estimates cache partitioning for the shared cache without requiring prior knowledge of the miss-rate curves. Chapter 10 explains in detail how PMCP can estimate cache partitioning schemes while simultaneously training the performance model to make better estimates in real-time.

Table 1: Comparison of various cache partitioning techniques

	UCP	**PriSM**	**SHARP**	**PMCP**
Light-weight	~	~	~	~
Scalable with number of threads	✓	✓	✓	✓
Not dependent on Miss-rate curves	✗	✗	✗	✓
Not dependent on accuracy of system model	✓	✗	✗	✓
Utilizes historical information efficiently	~	✗	~	✓

A detailed description of C-EKF is discussed in the next chapter. Note that this chapter does not discuss how C-EKF is adapted in PMCP to optimize cache utilization. It simply outlines the theory behind the working of C-EKF and highlights the appropriate set of equations.

3. CONSTRAINED EXTENDED KALMAN FILTER

Constrained Extended Kalman Filter is a modified Kalman filter designed for non-linear systems where the system dynamics are given in terms of state-space models subjected to certain constraints. To understand C-EKF, it is important to understand the working of a Kalman filter and the extended Kalman filter first.

Kalman filter is a minimum mean-square error estimator for a state vector $\boldsymbol{X}$ operating recursively on streams of noisy input to produce statistically optimal estimates [16]. Kalman filter works under the assumptions that the underlying system is *a linear dynamical system* and all error terms and measurements have a Gaussian distribution. The Kalman filter model assumes that true state of the system at time step *k* is evolved from the state at time step *k-1* according to the *state transition model* (previously referred as to *system model*) which can be described as

$$\boldsymbol{X_k} = \boldsymbol{A_k}\boldsymbol{X_{k-1}} + \boldsymbol{B_k}\boldsymbol{U_k} + \boldsymbol{w_k} \tag{1a}$$

where

U:	Control vector
A:	State Transition Model
B:	Control-input Model
w:	Process noise assumed to be zero mean Gaussian distribution with covariance $\boldsymbol{Q_k}$.

Also, at time *k* an observation vector $\boldsymbol{Z_k}$ of the true state $\boldsymbol{X_k}$ is designed based on equation 1(b)

$$\boldsymbol{Z_k} = \boldsymbol{H_k X_k} + \boldsymbol{v_k} \tag{1b}$$

where

H: Observation model that models the true state-space to the observed space

v: Observation noise assumed to be zero mean Gaussian distribution with covariance $\boldsymbol{R_k}$.

3.1. Extended Kalman Filter (EKF)

The EKF allows the state transition model and the observation model to be differentiable non-linear functions. In our case, the state transition vector is a linear function and the observation vector is a differentiable nonlinear function, $\boldsymbol{h}$. We will discuss more on the properties of the observation vector used in Section 3. Therefore (1a) and (1b) are modified to (2a) and (2b).

$$\boldsymbol{X_k} = \boldsymbol{A_k X_{k-1}} + \boldsymbol{w_k} \tag{2a}$$

$$\boldsymbol{Z_k} = \boldsymbol{h}(\boldsymbol{X_k}) + \boldsymbol{v_k} \tag{2b}$$

Unlike Kalman filter, $\boldsymbol{h}$ cannot be used directly in the filter equations. Instead, the Jacobian (also known as the output sensitivity matrix, $\boldsymbol{H}$) of $\boldsymbol{h}$ is evaluated and used in the filter equations. This process linearizes the nonlinear function $\boldsymbol{h}$ around the current estimates allowing the use of Kalman filter equations in the state space around the current estimate.

The following steps summarize the various steps involved in the evaluation of the estimated state, $\hat{\boldsymbol{X}}_{\boldsymbol{k}}$, using EKF given initial state vector $\hat{\boldsymbol{X}}_{\boldsymbol{0}}$ with initial error covariance matrix $\boldsymbol{P_0}$.

1. Project state ahead with $\boldsymbol{w_k} = 0$;

$$\hat{\boldsymbol{X}}_{\boldsymbol{k}}^{-} = \boldsymbol{A}_{\boldsymbol{k}}\hat{\boldsymbol{X}}_{\boldsymbol{k-1}} \tag{3}$$

2. Project $\boldsymbol{P}$, the estimated error covariance matrix

$$\boldsymbol{P}_{\boldsymbol{k}}^{-} = \boldsymbol{A}_{\boldsymbol{k}}\boldsymbol{P}_{\boldsymbol{k-1}}\boldsymbol{A}_{\boldsymbol{k}}^{\boldsymbol{T}} + \boldsymbol{Q}_{\boldsymbol{k}} \tag{4}$$

3. Compute $\boldsymbol{H_k}$ at $\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{-}$.

$$\boldsymbol{H}_{\boldsymbol{k}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\Big|_{\boldsymbol{X}=\hat{\boldsymbol{X}}_{\boldsymbol{k}}^{-}} \tag{5}$$

4. Compute Kalman gain $\boldsymbol{K_k}$

$$\boldsymbol{K}_{\boldsymbol{k}} = \boldsymbol{P}_{\boldsymbol{k}}^{-}\boldsymbol{H}_{\boldsymbol{k}}^{\boldsymbol{T}}\left(\boldsymbol{H}_{\boldsymbol{k}}\boldsymbol{P}_{\boldsymbol{k}}^{-}\boldsymbol{H}_{\boldsymbol{k}}^{\boldsymbol{T}} + \boldsymbol{R}_{\boldsymbol{k}}\right)^{-\boldsymbol{1}} \tag{6}$$

5. Correct the state vector

$$\hat{\boldsymbol{X}}_{\boldsymbol{k}} = \hat{\boldsymbol{X}}_{\boldsymbol{k}}^{-} + \boldsymbol{K}_{\boldsymbol{k}}\left(\boldsymbol{Z}_{\boldsymbol{k}} - \boldsymbol{H}_{\boldsymbol{k}}\hat{\boldsymbol{X}}_{\boldsymbol{k}}^{-}\right) \tag{7}$$

6. Correct the error covariance matrix $\boldsymbol{P}$.

$$\boldsymbol{P}_{\boldsymbol{k}} = (\boldsymbol{1} - \boldsymbol{K}_{\boldsymbol{k}}\boldsymbol{H}_{\boldsymbol{k}})\boldsymbol{P}_{\boldsymbol{k}}^{-} \tag{8}$$

Equations (1) – (6) are evaluated iteratively till the output reaches steady state.

3.2. Constrained-Extended Kalman Filter

Although EKF is a powerful tool for state estimation, some information about the system cannot be incorporated in the filter design such as equality or inequality constraints. Since Cache Partitioning is constrained by a linear equality constraint (see section 3.4), the methods used to incorporate inequality constraints is not discussed in this section. In C-

EKF, an EKF estimates the state variables and projects them onto a constrained surface (equality constraints) which can be generalized as

$$\boldsymbol{DX} = d \tag{9}$$

Estimate Projection method [17] is one of the ways of incorporating constraints in the filter design to get the constrained estimates in the form of

$$\boldsymbol{\widetilde{X}_k = \widehat{X}_k - W^{-1}D^T(DW^{-1}D^T)^{-1}(D\widehat{X}_k - d)} \tag{10}$$

where

$\boldsymbol{\widetilde{X}_k}$:	Constrained estimate of the state variable $\boldsymbol{X}$
$\boldsymbol{\widehat{X}_k}$:	Unconstrained estimate of the state variable $\boldsymbol{X}$ as calculated by the EKF
$\boldsymbol{W}$:	Positive-definite weighting matrix

By setting $\boldsymbol{W}$ to $\boldsymbol{P_k^{-1}}$ we obtain the maximum probability estimates of the state subjected to constraints. The equations for C-EKF are modified from those of EKF to incorporate the constrained estimated into the filter design.

$$\boldsymbol{\widehat{X}_k^- = \widetilde{X}_{k-1}} \tag{11}$$

$$\widehat{\mathrm{X}}_{\mathrm{k}} = \widehat{\mathrm{X}}_{\mathrm{k}}^{-} + \mathrm{K}_{\mathrm{k}}\left(\mathrm{Z}_{\mathrm{k}}\text{-}\mathrm{H}_{\mathrm{k}}\widehat{\mathrm{X}}_{\mathrm{k}}^{-}\right) \tag{12}$$

$$\boldsymbol{\widetilde{X}_k = \widehat{X}_k - P_kD^T(DP_kD^T)^{-1}(D\widehat{X}_k - d)} \tag{13}$$

Equations (11) - (13) are the primary equation that form the framework to model PMCP. By adopting appropriate system and performance models in equations (2a) and (2b) and subsequently evaluating equations (11) – (13), PMCP aims to estimate the optimal partition sizes for each thread such that the cache utilization is optimized and overall system performance is improved.

4. ESTIMATING CACHE PARTITIONING USING C-EKF

This section describes the framework of PMCP (Figure 3). Employing C-EKF as the state estimator in PMCP requires correspondence of the filter setup with appropriate state-transition and performance models for the cache. Table 2 presents the various symbols that will be used to describe the state-transition and performance models.

Table 2: Symbols used to describe the cache and cache-performance models in PMCP

n	Number of threads
C_i	Size of cache partitioning for thread *i*
M_i	Miss-rate of thread *i*
m_i	Number of misses of thread *i*
a_i	Number of accesses of thread *i*

4.1. System (Last Level Shared Cache)

The *system* under consideration is the last level shared cache to be partitioned. A state of the system is defined by ***C***, a column matrix where the *ith* row represents the size of cache allocated to the *ith* thread. Similarly, measurements are defined by ***M***, a column matrix where the *ith* row represents the measured miss-rate of the *ith* thread. This section elaborates on the requirements in the system to implement PMCP.

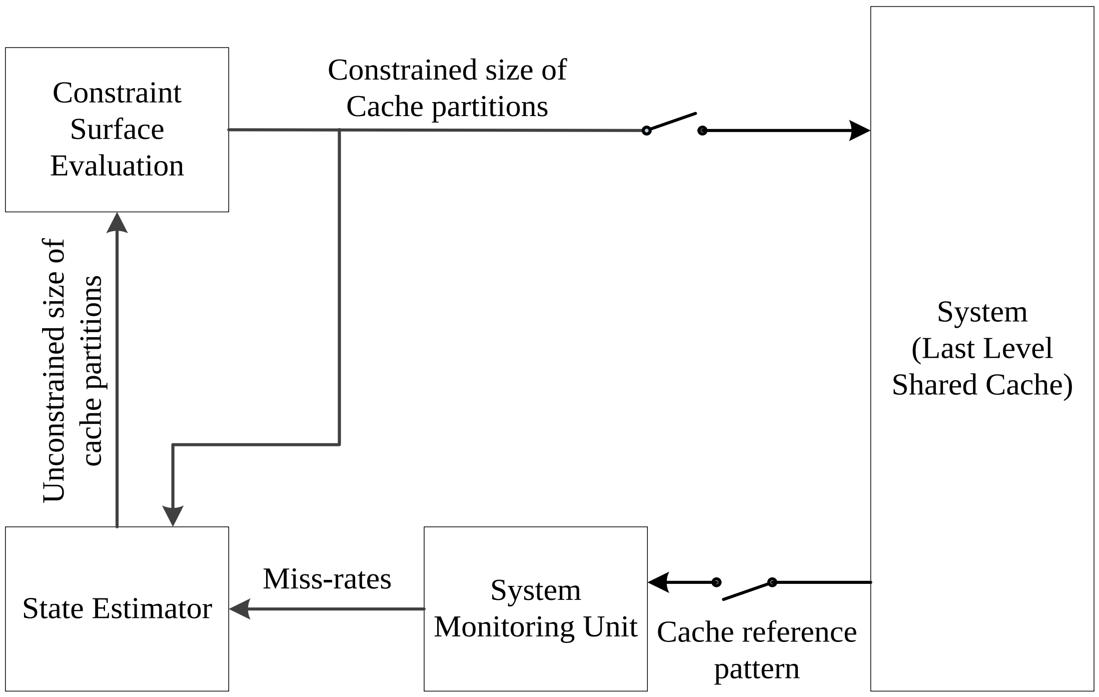

Constraint Surface Evaluation
Constrained size of Cache partitions
System (Last Level Shared Cache)
Unconstrained size of cache partitions
Miss-rates
State Estimator
System Monitoring Unit
Cache reference pattern

Figure 3: Framework of PMCP for estimating cache partitioning

4.1.1. Modifications in the cache

In order to capture the miss-rate of each thread, the cache is modified by adding an extra field called *tid* to each block in the cache. The number of bits required for *tid* equals $log_2 n$. This field stores information about the thread that owns the data in the block. For example, if *tid* equals '1' for a cache block, then the data mapped to this block is owned by thread 1. This field allows us to measure individual thread's hits and misses. Two registers per thread, referred as measurement registers, are reserved to store the number of hits and misses of each thread. Therefore the total number of measurement registers required is equal to *2n.*

4.1.2. Allocating space to the partitions

The fundamental philosophy behind PMCP is partitioning the cache and allocating the space to each partition. Each thread is then assigned to the partitions and have exclusive access only to the assigned partition. However, since the partition sizes can change in real-time a reallocation policy needs to be defined to ensure that the partitions remain defragmented. The reallocation policy is designed with the following properties

- After reallocation, the partitions are distinct and contiguous.
- After reallocation, the blocks in a partition are continuous per set.
- After reallocation, the first partition is always assigned to the thread tagged with *tid=1,* the second partition to the thread with *tid=2* and so on.
- After reallocation, there is at least one block in each partition.

Although such a reallocation policy makes it simpler to define distinct boundaries between partitions (see Figure 2: Organization of partitions in a cache partitioned using PMCP) thereby defragmenting the data, it can lead to certain issues with the way data is accessed by threads after a reallocation. The data access patterns of the threads are typically transient in real-time. Consequently the cache partition sizes for the threads may be transient as well. Figure 4a elaborates this concept in more detail. A cache block *B* which previously belonged to thread i+1 in TQ_j was reallocated to thread i in TQ_{j+1}. This can lead to redundant and/or inconsistent data in the cache memory. For example, in Figure 4b, *B* was reallocated from partition $i+1$ to partition i but not replaced in TQj_{+1} since thread i had no cache misses. If thread $i+1$ requested for *B* in TQj_{+1} it will experience a miss since thread $i+1$ can access partition $i+1$ only in TQ_{j+1}. Thread $i+1$ reads the value of *B* from the lower levels of memory, where it may not have the most recently updated value of *B*. In other words, not only are there multiple copies of the same data in the cache memory, it is possible that the data read by the thread is not the last updated value.

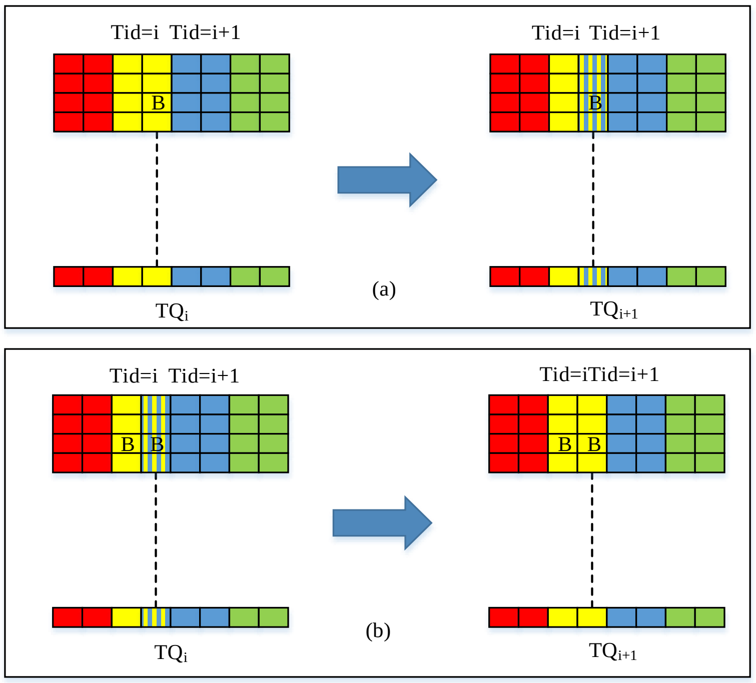


Figure 4: (a) displays the reallocation of the 4th block in each set from thread i to thread i+1. (b) Displays the reallocation of the 4th block back to thread i.

4.1.3. Modified cache replacement policy

The simplest way to address this issue would be to writeback all blocks that are reallocated followed by invalidating the same block in the cache. However, this process could be very time consuming. Instead, small modifications in the baseline Least Recently Used (LRU) replacement policy can achieve the same results without having to writeback all reallocated data every time the cache is repartitioned. The following rules were designed for the modified LRU replacement policy.

- All cpu-side requests (read and write) can access the entire cache.
- The cache can writeback data from any block in the entire cache.

- All memory side writes replace data from a block that belongs to requesting thread's partition only.
- In partition i, invalid blocks are used first.
- In partition i, if all block are valid, cache block that have $tid \neq i$ are replaced.
- In partition i, if all blocks are valid and have $tid = i$, LRU replacement policy is enforced on blocks belonging to partition i.

In the modified cache replacement policy, all threads are allowed to read from the entire cache. So even if a cache block is reallocated to a different partition, it can be read by all the threads. Only the writes from the memory side are restricted to the owning threads' partitions. There are two advantages of the modified replacement policy.

- It prevents mapping of redundant and inconsistent data in the cache memory
- It facilitates gradual defragmentation of data while significantly reducing the intermittent writebacks required to maintain data consistency.

4.2. System Monitoring Unit

PMCP requires periodic sampling of the miss-rates of each thread. The most straight forward way to realizing the miss-rates would be to monitor each thread's misses and accesses individually and derive the miss-rate of each thread. Miss rate of each thread is defined as

$$M_i' = m_i/a_i \tag{14a}$$

Since the bus connecting the shared cache to its neighboring units is shared, the access patterns of each thread is affected by the traffic on the bus that is generated by other threads as well. The probability that thread *i* accesses the cache is

$$\boldsymbol{p_i} = \boldsymbol{a_i} / \sum \boldsymbol{a_i} \tag{14b}$$

Therefore, from (14a) and (14b), the weighted miss-rate of thread *i* can be represented as

$$M_i = p_i * M'_i$$

$$M_i = m_i / \sum a_i \tag{14c}$$

4.3. State Estimator

In Figure 3: Framework of PMCP for estimating cache partitioning, the state estimator is essentially the EKF (equation (3) – (8)) part in the design of the C-EKF. To estimate cache partitioning, the EKF is adapted by choosing an appropriate *performance model* and *state-transition model*. As mentioned in Chapter 2, it is a rather difficult task to design these models. However, one of the advantages of using a C-EKF is that the user is allowed to use approximate models and compensate for the inaccuracy of the model by progressively updating the next state of the system based on measured performances and state-transition trajectories.

4.3.1. Choosing the Performance Model

The performance model is essentially a loss function that represents the cost (miss-rate) associated with the state of the system. In PMCP, it is used to evaluate the cost associated with the estimated next state of the system. The evaluated cost is compared with the observed cost to estimate the next state of the system iteratively. In PMCP, the loss function (performance model) describes the relation between cache sizes and miss-rates by employing the power function described by Chow [13, 14] and examined by Hartstein et al. [15]. Chow described the relationship between miss-rates and cache size as

$$\boldsymbol{M} = \boldsymbol{M_0}\boldsymbol{C}^{-\propto} \tag{15}$$

where $\boldsymbol{M}$ is the miss-rate (measurement vector, $\boldsymbol{Z}$ *in equation 2b*), $\boldsymbol{C}$ is the cache size (state vector, $\boldsymbol{X}$ *in equation 2a*) and α is approximated to 0.5 [15]. This is also called the √2 Rule. Therefore, from equations (5) and (15) $\boldsymbol{H_k}$ is constructed as an $n \times n$ diagonal matrix with

$$\boldsymbol{H_k} = \boldsymbol{diag}(\boldsymbol{M_0}(-\propto)\boldsymbol{C_k}^{-\propto}) \tag{16}$$

Note that the model chosen is a very simple model. The effects of the model on other aspects of the estimator will be discussed further in Section *4.5*.

4.3.2. Choosing the System Model

For the first part of the dissertation, the model chosen is the simplest model where the previous state of the system is projected to the next state of the system. In other words, equation (2a) is rewritten as

$$\boldsymbol{C_k} = \boldsymbol{C_{k-1}} + \boldsymbol{w_k} \tag{17}$$

4.4. Constraint Surface Evaluation

The output of the state estimator is not a practical partitioning scheme since it is unaware of the limitation on the state-space. If left unbounded, each partition can (theoretically) be infinitely large. However, since the cache space available in a system is limited, the output of the state estimator has to be truncated to a realistic partitioning by applying the constraint similar to equation (12). The constrained can be defined as– *The* *sum of all the partitions should be* ***less than or equal to*** *the total cache capacity*. Without loss of generality, this constraint can be redefined as – *The sum of all partitioning should be* ***equal to*** *the total cache capacity*. Equation (18) models such a constraint where C_i represents the cache partitioning allocated to the i^{th} thread and C_{total} is size of the cache which is usually fixed in a processor.

$$\sum C_i = C_{total}$$

$$\begin{bmatrix} 1 & 1 & \cdots 1 \end{bmatrix}_{1xn} \times \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{bmatrix}_{nx1} = \begin{bmatrix} \boldsymbol{C}_{total} \end{bmatrix}_{1x1}. \qquad (18)$$

Therefore from (9) and (18)

$$\mathrm{D} = [1\ 1\ 1\ \ldots\ 1]_{1\mathrm{xn}} \qquad (19a)$$

$$\mathrm{d} = \mathrm{C}_{\mathrm{total}} \qquad (19b)$$

The constrained estimates are evaluated similar to equation (13) and projected to the next iteration of the state estimator.

4.5. Tuning the Error Covariance Matrices $\boldsymbol{P_0}$*,* $\boldsymbol{Q}$ *and* $\boldsymbol{R}$

Q represents the expectation of the drift in the state variable and the inclination to follow the drift. ***R*** represents the prediction errors due to measurement error rather than parameter drifts. $\boldsymbol{P_0}$ is the initial error covariance. In a system, when the observations are complete, the error covariance matrices $\boldsymbol{P_0}$, ***Q*** and ***R*** covariance matrices can be estimated using sample covariance matrix. However, when the observation set is incomplete, deeper considerations are required. Statistical analyses of multivariate data usually involves exploratory studies about the interactions of the variables under consideration and is explicitly followed by statistical models involving the covariance matrices of the variables. Such estimations usually provide initial estimates that can be used to study the inter-variable interactions at run-time. In the absence of an efficient models for the error covariance matrices, the filter may not converge to the optimal states, the number of iterations required for the filter to converge may increase and in the worst case, the filter may even diverge. Furthermore, it is known that the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices for non-linear system and performance models may be dependent on the measurement vector and can evolve dynamically.

Several simulations were implemented in Matlab to empirically realize a good model to fit the definitions of these matrices. Note that the equations presented below need not represent the error covariance of every process that can potentially run on the system. However, our results will show that the system's performance was significantly improved nonetheless. This issue resurfaces again in Chapter 9 and is addressed in Chapter 10.

$$\boldsymbol{P_0} = \boldsymbol{I}$$

$$\boldsymbol{Q} = \boldsymbol{diag}(\boldsymbol{A}) * 1000000$$

$$\boldsymbol{R} = \boldsymbol{diag}((1./\boldsymbol{A})./\sum(1./\boldsymbol{A}))$$

4.6. Time Quantum

A Time Quantum (TQ) is defined as the time period after which PMCP monitors the cache to measure hits and misses. It can be perceived as a sampling period provided the system is monitored at regular intervals. However, based on the threads' dynamic variations in access patterns, the cache could be monitored at varying time intervals. Hence the time difference between two consecutive samples is referred as Time Quantum. The results presented in Chapter 9 were acquired by monitoring the cache at regular intervals which was chosen empirically to be 500 million clock cycles. It is important to note that a longer TQ can fail to isolate some of the major events but more time is available for the estimator to converge to the optimal partitioning. A shorter TQ captures events more closely but might be too short for the filter to converge to the steady state.

5. RECAPITULATING C-EKF FOR PMCP

In section 3, the Kalman Filtering algorithm is introduced which is based on linear dynamical systems described by equations (1a) and (1b). A variant of Kalman Filter called Extended Kalman Filter that linearizes a non-linear performance model about the current mean and covariance is documented next since cache performance (miss-rate) scales in a nonlinear fashion. Equations (2a) and (2b) describe such a system and the various steps involved in EKF is summarized in equations (3) – (8). EKF is a good state-estimator but sometimes it is difficult to account for all conditions and constraints in the state-transition and performance models. C-EKF is introduced next to account for any physical constraints that may exist in the system. Equations (11) – (13) apply such constraints of the type equation (9) on the estimates produced by equations (3) – (8).

In Chapter 4, the framework for implementing PMCP using various models and hardware modifications are identified. In order to utilize the models described in Sections 4.3.1 and 4.3.2 to perform PMCP, equations (15) – (19) are mapped onto equations (3) – (13) appropriately. The subsequent equations for the algorithm are as follows

State Transition and Observation Models:

$$\mathrm{C_k} = \mathrm{C_{k\text{-}1}} + \mathrm{w_k} \tag{20a}$$

$$\boldsymbol{M_k} = \boldsymbol{C_k^{-\propto}} + \boldsymbol{v_k} \tag{20b}$$

State Estimator :

$$\hat{\mathrm{C}}_{\mathrm{k}}^{-} = \tilde{\mathrm{C}}_{\mathrm{k\text{-}1}} \tag{20c}$$

$$P_k = P_{k-1} + Q_k \tag{20d}$$

$$\boldsymbol{H_k} = \boldsymbol{diag}((-\propto)\boldsymbol{C_k^{-}}^{\propto-\boldsymbol{1}}) \tag{20e}$$

$$K_k = P_k^- H_k^T \left(H_k P_k^- H_k^T + R_k\right)^{-1} \tag{20f}$$

$$\hat{C}_k = \hat{C}_k^- + K_k\left(M_k - H_k \hat{C}_k^-\right) \tag{20g}$$

$$P_k = (1 - K_k H_k) P_k^- \tag{20h}$$

Constraint Surface Evaluation :

$$\tilde{C}_k = \hat{C}_k - P_k D^T (D P_k D^T)^{-1} (D\hat{C}_k - d) \tag{20i}$$

At the start of each TQ, the cache is monitored, miss-rates evaluated and equation (20) is performed iteratively till the predicted state vector ***C*** reaches steady state. The steady state vector ***C*** is applied to the cache and is maintained until the start of the next time TQ. The following section elaborates on the various simulations that were performed in MATLAB and is intended to elaborate on the real-time analysis of PMCP.

6. SIMULATING PMCP IN MATLAB

Several simulations were performed in MATLAB to understand the process of C-EKF as it is adapted for PMCP in real-time. In all cases, the processor is assumed to be running four threads. The simulations are based on hypothetical situations and each thread is assigned equal partition at the start of the simulation. For example, if the system (last-level shared cache) is assumed to consist of 100 blocks, the initial state for the system is $\boldsymbol{C}=[25;25;25;25]$. The miss-rates of each thread is also assumed to be the same and randomly chosen as 0.2. The system is monitored and various scenarios are generated to test the validity of C-EKF. The C-EKF estimates the cache partition sizes for each thread over several iterations such that cache utilization is optimized. In other words, overall miss-rate of the system is minimized. The outputs recorded are the states predicted by PMCP in real-time. Although the simulation presents the estimated states in each iteration, only the steady state values are finally applied on the system.

The simulation results presented in this section have the following characteristics. The X-axis represents time and is measured in terms of the number of iterations (t). Each iterations is assumed to have identical calculations thereby taking the same amount of time for completion. The Y-axis represents the predicted cache partition sizes assigned to all the threads measured in terms of blocks. As mentioned before, in each case, the system is assumed to have the same initial condition with $\boldsymbol{C}=[25;25;25;25]$ and $M=[0.2;0.2;0.2;0.2]$.

Case 1:

The system is monitored at the end of TQ_0 (first TQ) i.e. at t=5. Figure 5 and Figure 6 shows the states predicted by PMCP during every iteration till it reaches steady state when ***M****=[0.8;0.2;0.2;0.2]* and ***M****=[0.5;0.2;0.2;0.2]* respectively at t=5. It is clear that more cache space is allocated to thread 1 since it has the highest miss-rate in both cases. It should also be noted that all the other threads are penalized equally even though they maintained their miss-rate of 0.2. This is because thread 1's miss-rate increased significantly which triggers C-EKF to allocate more space to thread 1. However, since the total cache space is limited to 100 blocks, allocating additional space to thread 1 required freeing some space allocated to the other threads. Therefore threads 2, 3 and 4 are penalized.

Another key observation is the increase in the space allocated to thread 1 in Figure 5 compared to the space allocated to it in Figure 6. In addition, the space allocated to threads 2, 3 and 4 in Figure 5 is less compared to the space allocated to the same threads in Figure 6. These are obvious conclusions since thread 1's miss-rate is higher in Figure 5. Note that C-EKF reached steady state at different times in the two cases.

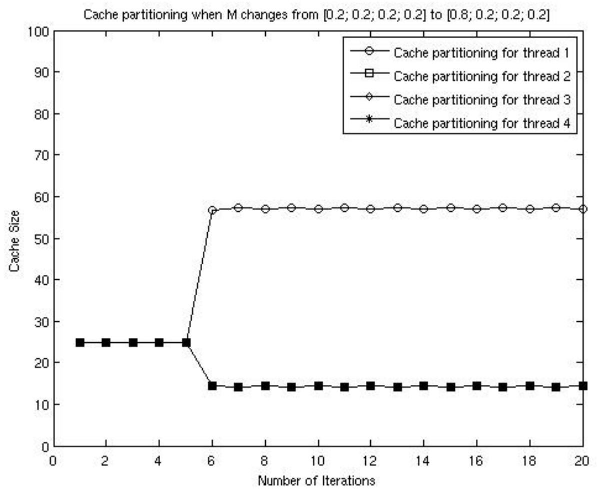


Figure 5: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.8.

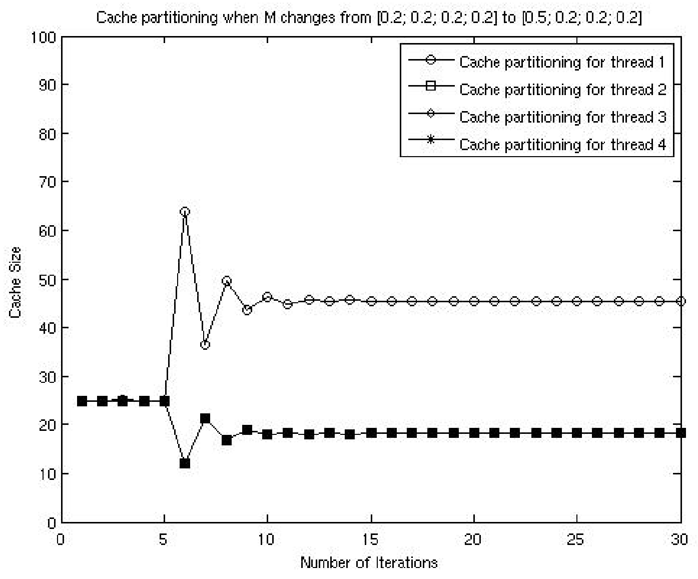


Figure 6: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.5.

Case 2:

In this case, TQ is assumed to be equal to 10. The system is monitored at the end of TQ_0 (first TQ) i.e. at t=5 and TQ_1 i.e. t=15. Figure 7 and Figure 8 show the states predicted by PMCP during every iteration until it reaches steady state for this case study. The measurement vector (miss-rate) $\boldsymbol{M}=[0.8;0.2;0.2;0.2]$ at the end of TQ_0 in both figures but $\boldsymbol{M}=[0.3;0.2;0.2;0.2]$ and $[0.2;0.2;0.2;0.2]$ in Figure 7 and Figure 8 respectively at the end of TQ_1. At the end of TQ_0, thread 1’s miss-rate increased prompting C-EKF to allocate more space. At the end of TQ_1, since the miss-rate for thread 1 reduces from 0.8 to 0.3 in Figure 7 thread 1’s partition reduced such that other threads can benefit from more space allocation. As expected thread 1’s partition size reduces and other thread’s partition sizes increase. However, since thread 1’s miss-rate is still higher than other threads, it is allocated more space in TQ_1. In Figure 8 all threads return to miss-rate of 0.2 at the start of TQ_1; so all threads are reassigned equal partition.

Figure 9 depicts the result of a test case similar to the test whose results are presented in Figure 8 except that at the end of TQ_0, thread 1’s miss rate changes to 0.5 and at the end of TQ_1 all threads’ miss-rates change to 0.5. Despite an increase in all the miss-rates, all threads experience the same penalty and have equal partitions since all threads have equal miss-rates.

Case 3

In this case, thread 1’s miss rate is recorded as 0.8 at the end of TQ_0 while all other threads record a miss-rate of 0.2. And at the end of TQ_1; miss-rates are recorded as ***M****=[0.3;0.5;0.8;0.2]*. Figure 10 presents the predicted partition sizes for the threads. It is clear that the partition sizes for the threads follow the trend that higher the miss-rate, more space is allocated to the partition.

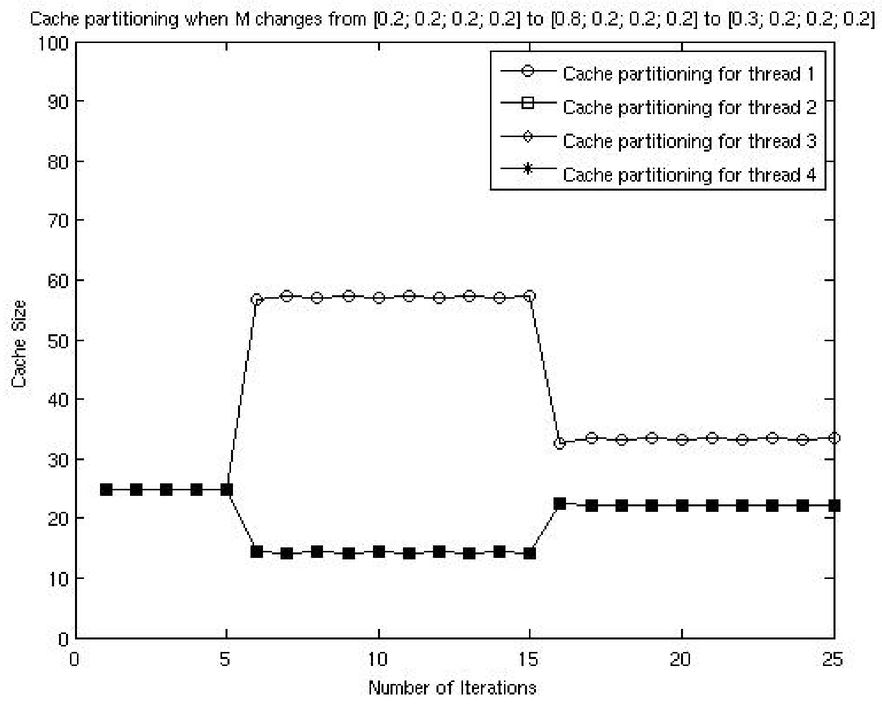


Figure 7: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.8 to 0.3.

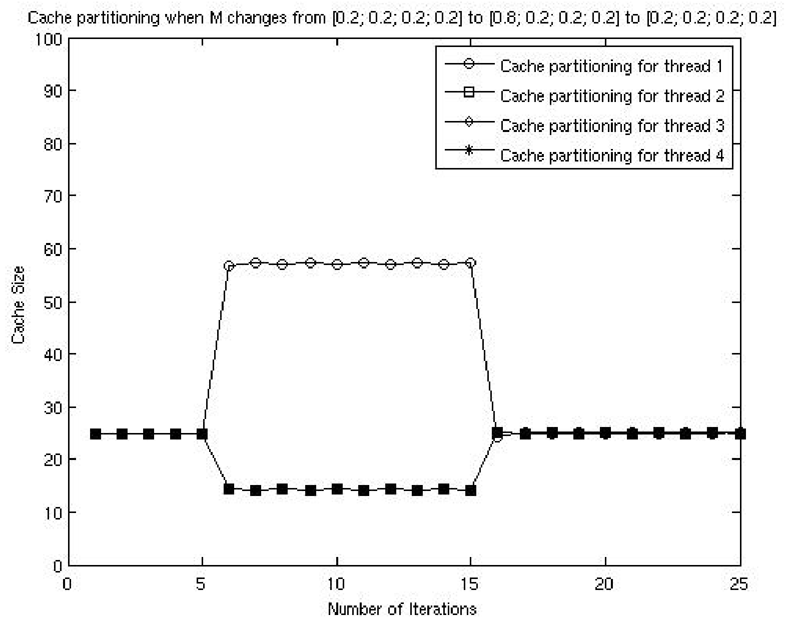


Figure 8: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.8 back to 0.2.

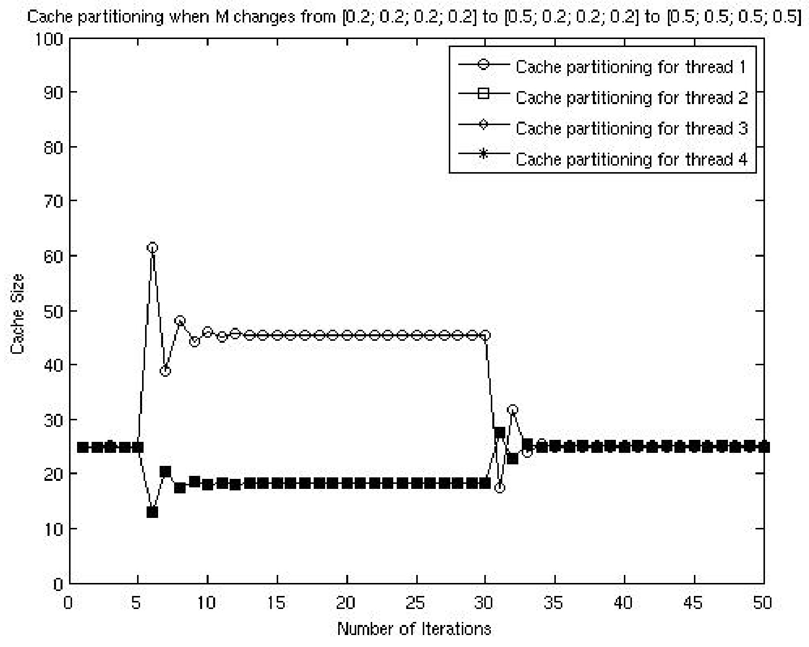


Figure 9: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.5. Then all threads' miss-rae change to 0.5

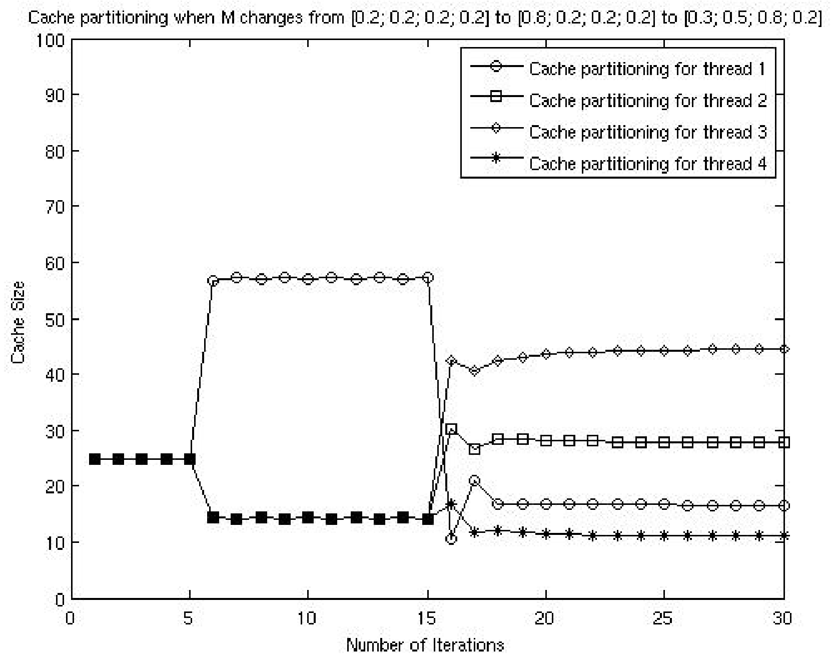


Figure 10: Simulating PMCP for 4 active threads on a processor with cache size of 100. Thread 1's miss-rate changes from 0.2 to 0.8. Then the miss-rate for the threads follow the order 0.3, 0.5, 0.8 and 0.2 respectively

7. GEM5 SIMULATOR

The GEM5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture. GEM5's commitment to modularity allows the users to focus on particular aspects of the simulator without having to understand all of the simulator's code. GEM5 provides significant flexibility to the user by providing four different CPU models for the simulated architecture. The user can configure the architectures using multiple models and switch between the modes at run-time.

- SimpleCPU - The SimpleCPU is a purely functional, in-order model that is suited for cases where a detailed model is not necessary. This can include warm-up periods, client systems that are driving a host, or testing to make sure a program works. There are two models under SimpleCPU namely
    - AtomicSimpleCPU - The AtomicSimpleCPU is the version of SimpleCPU that uses atomic memory accesses. It uses the latency estimates from the atomic accesses to estimate overall cache access time.
    - TimingSimpleCPU - The TimingSimpleCPU is the version of SimpleCPU that uses timing memory accesses. It stalls on cache accesses and waits for the memory system to respond prior to proceeding.
- InOrderCPU - The InOrder CPU model is designed to provide a generic framework to simulate in-order pipelines with the described ISA and pipeline. The generic

pipeline stages are provided but the user has the flexibility to add custom pipeline stages, vary issue width and scale the number of hardware threads without having to recreate the entire pipeline

- OutOfOrderCPU/O3CPU - The O3CPU model is pipelined out-of-order model that simulates dependencies between instructions, functional units, memory accesses, and pipeline stages. Parameterizable pipeline resources such as the load/store queue and reorder buffer allow O3 to simulate superscalar architectures and CPUs with multiple hardware threads. The O3 model is also “execute-in-execute”, meaning that instructions are only executed in the execute stage after all dependencies have been resolved. This model simulates the five stages which are fetch, decode, rename, issue/execute/writeback and commit.
- TraceCPU - The TraceCPU model was introduced in GEM5 recently and is still under development. The Trace CPU model plays back elastic traces, which are dependency and timing annotated traces generated by the Elastic Trace Probe attached to the O3 CPU model. The focus of the Trace CPU model is to achieve memory-system (cache-hierarchy, interconnects and main memory) performance exploration in a fast and reasonably accurate way instead of using the detailed but slow O3CPU model. Currently traces have been developed for single-threaded benchmarks and the model works with single-threaded simulations only.

Apart from CPU models, GEM5 also provides flexibility with the simulation mode with respect to the involvement of the Operating System (OS) in the simulation. Any of the above described CPU models can run in one of the following two modes.

- System-call Emulation (SE) mode - only the statically compiled binaries need to be specified and no operating system is required to run the binaries since GEM5 emulates most of the system-level services in this mode. GEM5 simulates the most common instructions from the executable in this mode but the system calls are trapped and passed to the host machines OS to be executed. Since thread scheduling is absent in the SE mode, the threads have to be statically mapped to the cores.
- Full System Emulation (FS) mode - In FS mode, a complete system with OS and devices is modeled. The simulated system executes both user level and kernel level instructions in the FS mode. The bare minimum environment for running the OS is simulated that can support basic functionalities like interrupts, exceptions, I/O devices and so on. Compared to SE mode, FS mode is slower but simulates a working system more accurately while allowing a larger set of workloads to be simulated.

GEM5 also simulates a variety of memory modules and interconnects as well. There are two memory system available for the user to choose from namely

- Classic Memory System - The classic memory system provides GEM5 a fast, flexible and easily configurable memory system at the cost of detail in the coherency interactions. Cache coherence is maintained using an abstract MOESI snooping protocol where state-transitions due to snoops occur instantaneously. Although there is an inflexibility in choosing the coherency protocol, the classic memory system allows the simulation to be fast-forwarded to the desired section of the simulation. It is also easy to configure and is much faster than the system described below.

- Ruby Memory System - In contrast to the classic model, the ruby memory system sacrifices simulation speed to provide GEM5 a flexible infrastructure capable of accurately simulating a wide variety of memory systems. In particular, Ruby supports a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) where one can define many different types of cache coherence protocols. Essentially SLICC defines the cache, memory, and dma controllers as individual per-memory-block state machines that together form the overall protocol. By defining the controller logic in a higher level language, SLICC allows different protocols to incorporate the same underlining state transition mechanisms with minimal programmer effort.

GEM5 also supports several Instruction Set Architectures (ISAs) and Operating Systems for both FS and SE modes namely Alpha, ARM, MIPS, Power, Sparc and x86. GEM5's modularity allows the required ISA to plug into a generic CPU model and the memory subsystem. It also supports several interconnection networks, devices and even allows the host machine to communicate with the simulated architecture over tcp/ip.

GEM5 is featured as an event-driven simulator which generates the whole system at run-time based on predefined object modules. These modules are typically developed in C++ and the object oriented design provides modularity and flexibility to the system design while significantly leveraging from inheritance. All major simulation components in GEM5 are inherited from a *SimObject* that describes the basic configuration, initialization, statistics and serialization. All major components of the system like the processor, cache, memory, interconnects are SimObjects. Each SimObject has two classes, one in python and one in C++. The C++ class describes the state and the remaining behavior of the

SimObject while the python class specifies the SimObject’s parameters and are used in script-based configuration. The python integration provides a powerful front-end interface that initializes and configures the system to be simulated and control for the simulation at run-time.

Another very useful feature in GEM5 that allows the user even more control over the simulation is the ability to define events in SimObjects and populate the event queue throughout the simulation. These events can be scheduled based on the number of *Ticks* or instructions simulated.

With all the features mentioned above a formal design of experiment is formulated to test PMCP on GEM5. Indeed some modifications were applied to a few standard modules of the simulator. The next chapter elaborates on the various modifications incorporated in the standard GEM5 simulator such that the PMCP framework defined in Chapter 4 could be simulated.

8. EVALUATING PMCP ON GEM5

PMCP is evaluated on the GEM5 simulator using a processor that consists of a four core CMP processor modeled with the x86-64 ISA. Each core has private 64KB split L1 data and instruction caches and a shared 2MB L2 cache. Table 2 shows the details of the processor.

Table 3: Main characteristics of the simulated processor

Cores	4 cores, x86-64 ISA, in-order ideal Instructions per cycle =1 at 2GHz
L1 Cache	64KB, 4-way Set Associative, split L1/D1, 1 cycle latency
L2 cache	2MB, 32-way Set Associative, 4 cycles L1-L2 latency,
MCU	4 memory controllers, 200 cycles zero-load latency, 32GB/s peak BW

PMCP is performed on the shared L2 cache and the number of ways is divided between the competing threads. As mentioned in Section 4.1.1, the L2 cache is equipped with custom registers that update with every miss and hit. The SMU (see Figure 3) samples these registers throughout the simulation to gather the number of hits and misses each thread experiences in the L2 cache.

8.1 Experimental Setup

GEM5 simulates PMCP in the FS mode to simulate a real system. A similar run procedure described in [3] is adopted to the simulation on GEM5. The simulation starts with the assumption that all the threads have same priority and equal partitioning. The cache is warmed by ensuring that at-least one thread has executed 20 billion instructions before PMCP starts. During warmup the simulated processing cores run in AtomicSimpleCPU mode where the latency due to memory is not included in the simulation. However, the entire memory hierarchy is still updated with the data being transferred between the main memory and the processor. This makes the warm-up process significantly faster. GEM5 then switches to O3CPU mode to simulate an out-of-order CPU while considering all caches and memory latencies. PMCP is activated as well and is scheduled to run every 50 million clock cycles (TQ). At the start of each TQ, PMCP monitors the custom performance registers and evaluates the miss-rates (equation 14c) of all the threads. Using the measured miss-rates, the State Estimator with the constraint execute iteratively until the cache partitions converge to a steady state value. During the transient states, C-EKF is disconnected from the system. Only when the steady state value is reached PMCP is connected back to the system and the steady state values are applied to the system.

An interesting observation made during experimentation was that the steady state values of the cache partitions make significant transitions during the first few TQ. This period is named as *"High level Transient Period" (HLTP)* (see Figure 11) and is defined as the number of TQ required such that the steady state values during five consecutive TQ differ by more than 1%. During HLTP it is impractical to measure throughput and fairness since the system would be making several writebacks to maintain data consistency. Once the steady state values stabilize over several TQs, performance measures like throughput and fairness are measured. This period is named *"High Level Steady-State Period" (HLSP)*. The simulator continues to simulate the processor in HLSP until all threads have executed at least 10 million instructions. Note that PMCP spends significantly more time in HLSP than HLTP. So the overhead due to writebacks during HLTP is significantly small and can be ignored. Therefore all measurements are made in HLSP. Figure 11 shows the top level time-profile of such an experiment.

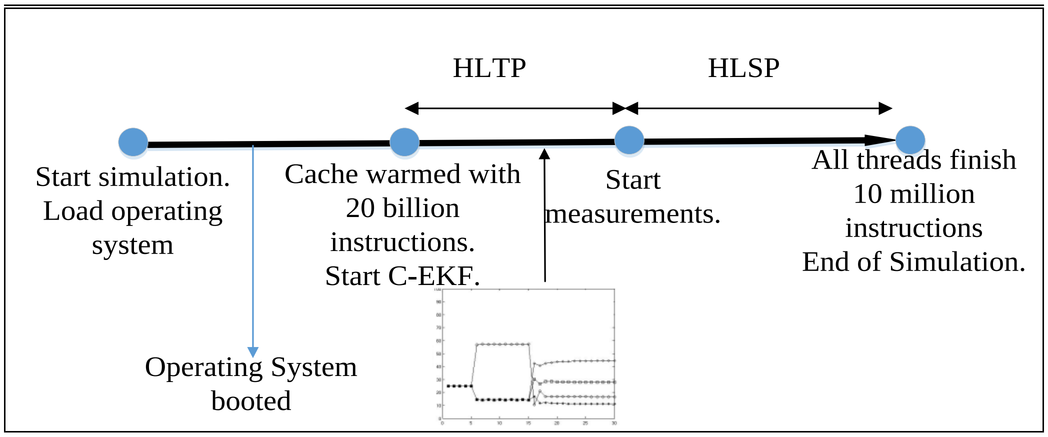

HLTP
HLSP
Start simulation. Load operating system
Cache warmed with 20 billion instructions. Start C-EKF.
Start measurements.
All threads finish 10 million instructions End of Simulation.
Operating System booted

Figure 11: Top level time profile of the simulation

8.2 Performance metrics

The simulation continues until all active threads execute 10 million instructions in HLSP. The performance metric chosen to represent the performance of the system is inspired by the standard metrics used by UCP [8], Vantage [3], PriSM [2] and Jigsaw [1].

They present the performance of the system as the aggregate IPC for throughput and fair-speedup for fairness.

8.3 Workloads

SPEC CPU2006 benchmark suite was used to evaluate PMCP. Sanchez et al. [3] divided the 29 benchmarks of the suite into 4 categories namely intensive *(n)*, cache-friendly *(f)*, cache-fitting *(t)* and thrashing/streaming *(s)*. Table 4 presents the description of these 4 categories.

Table 4: classification of the spec cpu2006 benchmark [3]

n	Applications with very low L2 misses per kilo instructions
f	Applications that gradually benefit from increase in cache size
t	Applications whose misses reduce abruptly with increase in cache size
s	Applications that do not show any benefit when increasing the cache size.

The same classification of the benchmarks is assumed for designing the workload suite to test PMCP on GEM5. The workload for the test cases are built by choosing one application from each category as the representative application. Each core runs a single threaded application choosing one of the 4 representative applications. Multiple cores are allowed to execute the same application as long as the input set varies. Since each core executes a single-threaded application and none of the threads share the same data set, all threads ae independent of each other.

8.4 Modifications in GEM5

Several changes are made in the GEM5 software to accommodate PMCP. The modular design aids with adding new features or modifying existing functionalities in the simulator. Since the memory hierarchy is of interest in PMCP, as long as the fundamental interaction of the memory hierarchy with the processing core and its pipeline is maintained, there is no need to change the simulation methodology of the processors' pipeline. Hence most of the effort was directed towards introducing the following modifications and modules to the simulator.

- Building the C-EKF module
- Modifications to the L2 cache design
- Modifications to the L2 cache's replacement policy
- Modifications to the Data packets received and sent from L2 cache
- Introducing measurement registers to evaluate miss-rates for each thread
- Defining and scheduling various events

.

8.1.1. C-EKF module

The C-EKF module is built as a class named *CEKalman*. The equations described in Chapter 5 are implemented in this module. These calculations are executed iteratively until C-EKF converges to the steady state values. Typically it takes less than 10 iterations for PMCP to reach the steady state values. Without loss of generality, each iteration of C-EKF is scheduled to execute every simulated clock cycle until the steady state values are reached since the computations are relatively simple and the system is monitored every 50 million clock cycles. During the transient states, CEKalman module is disconnected from the cache to avoid the application of the transient states and the cache continues to remain in the steady state from the previous TQ. Only when CEKalman converges to a steady state, the cache is updated with the new partitioning sizes.

All the variables are declared as float variables except for the state variable $\tilde{C}_k$ which is the output of C-EKF and is applied to the L2-cache. $\tilde{C}_k$ is declared as in integer since $\tilde{C}_k \in \boldsymbol{I}$. The following pseudo code describes the C-EKF module.

```
$C_{old}4 \leftarrow C_{old}3$
$C_{old}3 \leftarrow C_{old}2$;
$C_{old}2 \leftarrow C_{old}1$;
$C_{old}1 \leftarrow C_{old}0$;
Read current state ($C_{old}0$) of the system;
Read number of misses, hits for each thread;
Calculate miss-rate ($\boldsymbol{Z}$) for each thread;
Calculate Q and R matrices;
If ($C_{old}1$ - $C_{old}0$) < 0.01 and ($C_{old}2$ - $C_{old}1$) < 0.01 and ($C_{old}3$ - $C_{old}2$) < 0.01 and($C_{old}4$ - $C_{old}3$) < 0.01 then
    Calculate $\tilde{C}_k$;
    Schedule to run in next simulated clock cycle;
Else
    Update Cache partition sizes on the system
    Unscheduled any remaining iterations
```

8.1.2. Modifications to the L2 cache design

In GEM5 all the cache memory modules inherit their functionalities from the Cache class (Cache inherits from MemObject which inherits from SimObject) and are created/configured into the system at run-time. Therefore the instances of L1 d/i cache, L2, L3 and so on; all inherit their functionalities from the same module. The code for the generic cache module is modified to isolate the functionalities specific to the L2 cache such that PMCP can be applied only to it This includes the modified LRU replacement policy and some variables that are directly impacted by the changes included variables that gathered statistics for the L2 cache, variables to measure misses and hits and synchronization variables. Details on these variables are discussed later in the chapter. From the simulated hardware point of view, the main modification to the L2 cache is the addition of the threaded variable to each block in the cache. Each block in the L2 cache keeps track of the thread whose data is written from the main memory into it.

Provisions are also applied to make sure that the modified replacement policy is implemented correctly in the L2 cache specifically. Details on how this is achieved is discussed in the next section.

8.1.3. Modifications to the the LRU replacement policy

There are four kinds of bus requests that are placed to the cache memory. Read/write requests from a higher level memory/processing core (named “cpu-side” in GEM5) and read/write requests from a lower level memory/ devices (named “memory-side” in GEM5). The cache memory is designed as a stack in GEM5 with the least recently used block at the head of the stack. When a block is requested by a bus and is a hit, the data in the block is passed to the corresponding bus and the block is moved to the bottom of the stack. In case of a miss, the block at the top of the stack is replaced with the new block and moved to the bottom of the stack. Indeed the data in the block is written back to the lower level of the memory before being replacement. Essentially the most recently used block is at the bottom of the stack and the least recently used is at the top. In case of a set associative cache, each set is implemented as a stack. When a set is referenced, the corresponding stack is checked for a hit or a miss.

To implement the modified cache replacement policy, an exclusive “Head” and “Tail” pointer is introduced for each thread. The pointers are positioned in the stack such that

$$head_i = \sum_{j=0}^{i} partition\ size_{j-1}$$

$$tail_i = \sum_{j=0}^{i} partition\ size_j - 1$$

Where *i = 0, 1, 2, 3, …, n-1*. Note that $head_0$ always points at block 0.

When a request is placed from the cpu-side the cache performs the following steps

- Entire cache is searched for the requested data irrespective of which thread requested the data.
- If the cache experiences a hit, the data from the block is read and the block is moved to the bottom of the partition it belongs to irrespective of its *threadid*.
- If a miss is experienced, the incoming request’s *threadid* is checked first (say *threadid* = *i*).
- The part of the stack between $head_i$ and $tail_i$ is searched for any block where $threadid \neq i$.
    - If a block with $threadid \neq i$ is found, the cache writebacks the block and the new data is read into the partition.

- The block is moved to the $tail_i$ position instead of the bottom of the entire stack.
- If all blocks have $threadid = i$ between $head_i$ and $tail_i$, the standard LRU replacement policy is enforced. But the new block is moved to $tail_i$ position instead of the bottom of the stack.

To implement the replacement policy mentioned above, the bus requests have to present to the cache the origin of the requests to be compared with the *threadid* of the blocks. The design of the “request packets” had to be modified to include this information.

8.1.4. Modifying the request packets

A new variable called *tid* is introduced in the *Request* packets class that holds information about the origin of the request. Currently the *coreid* is stored as *tid* in every packet based on its origin. Although the modified replacement policy requires the *threadid*, it is acceptable to store *coreid* as *threadid* since all cores run independent single threaded benchmarks. The requests generated from any other device or OS have $tid = -1$. Therefore every request packet is equipped with the information about its origin. This information is cross-checked in the L2 cache to implement the modified cache replacement policy.

8.1.5. Measuring misses and hits

New variables had to be introduced to measure the hits, misses and access for L2 cache during the simulation. These variables are different from the statistical variables that exist in GEM5. Redundant variables had to be introduced because the statistical variables of GEM5 cannot be referenced at run-time. In fact, the creators of GEM5 call it “magic” when they describe these variables. These variables measure the hits/misses/accesses of each thread by monitoring the blocks with the corresponding *threadids*. This information is used by PMCP to generate miss-rates to be used by the C-EKF module. The statistical variables help in recording the overall trend in the system for data analysis.

8.1.6. Scheduling events

An acumen of the events that need to be scheduled in GEM5 in order to implement the experimental setup is presented in Table 5. These events are system wide events and are defined as part of the executable when running an experiment in GEM5.

Table 5: Description and schedule of the various top level events as referred in the simulation

Event type	Description	Scheduled to end
Fast Forward	Runs the experiment in AtomicSimpleCPU mode	Until OS boots + all region of interests reached
Warm-up	Runs until the each thread completes defined number of instructions	At least 20 billion instructions by each thread after Fast-forwarding
HLTP	Switches to O3CPU mode	Dynamically realized
HLSP	Remains in O3CPU mode	At least 10 million instructions by each thread

Apart from these events that guide the simulation in GEM5, some more events are created to collecting data at appropriate intervals, measure misses/hits, start PMCP every

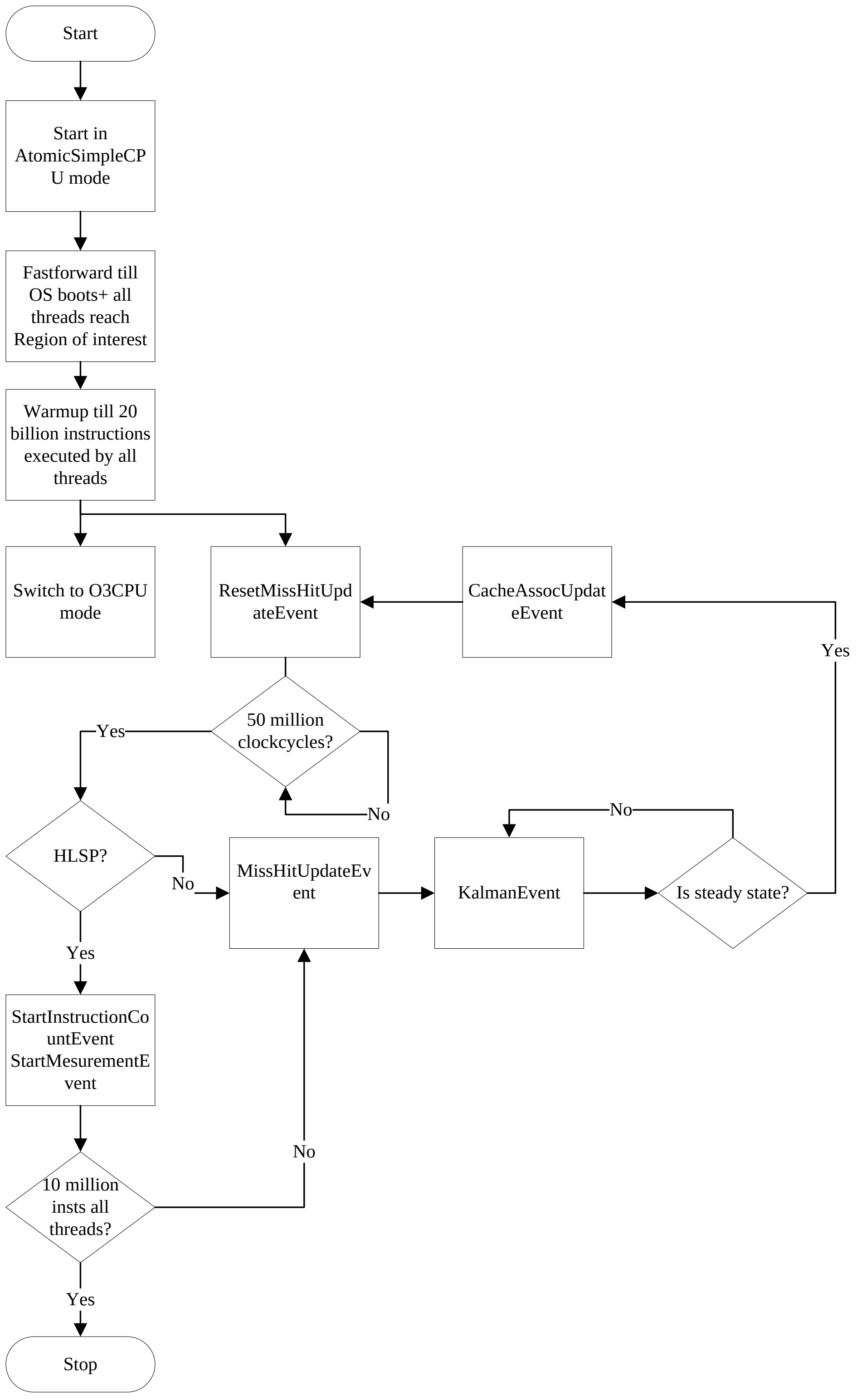
50 million simulated clock cycles, schedule the next iteration of C-EKF and update the partition sizes in the cache. This section documents all the events as they are defined in the various modules to achieve the desired design of experiment. All the events that appear in the various classes in GEM5 to implement PMCP are listed in Table 6.

Table 6: Location, description and schedule of the user defined events used to implement the design of experiment

Event variable	Class	Description
KalmanEvent	CEKalman	Run one iteration of C-EKF
StartInstructionCountEvent	BaseCPU	Starts the 10 million instruction count of HLSP
StartMesuringEvent	BaseCPU	Schedules a statistics dump every 50million clock cycles
StartKFEvent	BaseCPU	Starts StartInstructionCountEvent and StartMesuringEvent
MissHitUpdateEvent	Cache	Updates the ***Z*** vector with the number of misses/hits at the end of the current TQ
CacheAssocUpdateEvent	Cache	Updates the number of blocks for each partition in the L2 cache once steady state is achieved
ResetMissHitUpdateEvent	Cache	Resets the misses/hits counters

InitSetupEvent	Cache	Initializes the system with equal partitions for each thread and resets the misses/hits counters

Figure 12 shows the flowchart of all the events as they occur throughout the experiment including the setup events as well as the individual events defined in the classes.


Start
Start in AtomicSimpleCPU mode
Fastforward till OS boots+ all threads reach Region of interest
Warmup till 20 billion instructions executed by all threads
Switch to O3CPU mode
ResetMissHitUpdateEvent
CacheAssocUpdateEvent
Yes
50 million clockcycles?
Yes
No
No
HLSP?
No
MissHitUpdateEvent
KalmanEvent
Is steady state?
Yes
StartInstructionCountEvent StartMesurementEvent
No
10 million insts all threads?
Yes
Stop

.

Figure 12: Schedule of all events both user defined and system-level such that the design of experiment s implemented efficiently.

9. RESULTS

The performance of a processor where the L2 cache is partitioned using PMCP is compared with the performance of a processor with the baseline shared L2 cache and the standard LRU replacement policy. The results are presented as clustered bar graphs where each cluster is the performance of a specific workload in both the processors. The left bar in the cluster always represents the performance of the processor with the standard shared L2 cache and the right bar represents the performance of the processor with the partitioned cache with PMCP. The workloads are named based on their classifications (see Table 4). For example, workload *fnts* is a workload where an *f* type benchmark runs on core 1, *n* type on core 2, *t* type on core 3 and *s* type on core 4.

9.1 Performace on throughput

The most common way to represent the aggregate throughput is the sum of individual IPCs.

$$IPC_{total} = \sum IPC_i \text{ where } i = 0, 1, 2, 3, \ldots n\text{-}1$$

Figure 13 compares the aggregate throughput for the simulated workloads. Each workload's throughput is represented as a stacked bar where each stack in the bar is the IPC for a benchmark in the workload. From bottom to top are the IPCs of the benchmarks on core 0 to core 3 respectively. The total height of the stacked bar is the aggregate throughput of the processor. The throughput in most cases improved in case of the partitioned L2 cache with a maximum of 35.35% in the case of the *fnts* workload.

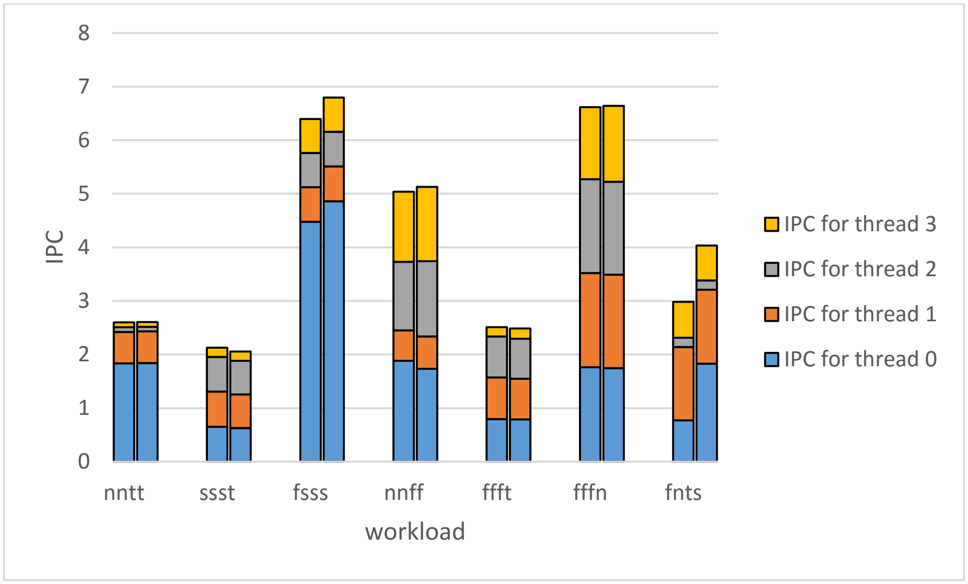


Figure 13: IPC for various workloads simulated. The left bar in each cluster represents the IPC for a processor where the with standard LRU replacement in the L2 cache. The right bar in each cluster represents the IPC for a processor where the L2 cache is partitioned using PMCP. The label of each cluster indicates the benchmarks for the cluster.

IPC for the system with partitioned cache is higher in cases where at least one benchmark is f type and the other applications have a small footprint in the L2 cache. For example, workloads *fsss*, *nnff*, *fffn* and *fnts*. In each case, IPC_{total} improved primarily because of improvement in the IPC for the core with the *f* type benchmark. It is also interesting to observe the effect of the *s* type benchmarks. Very little to no improvement was observed in IPC for the core with the *s* type benchmarks through PMCP. In fact the presence of the *s* type benchmark in the workload limits the maximum achievable improvement through PMCP. *S* type benchmarks are streaming or thrashing type benchmarks. Such benchmarks require significantly more bandwidth but do not have a large footprint in the cache. Miss-rate for these benchmarks were always 0.5 (since there's always one hit and one miss for every new block). This makes PMCP assign more space for the core with the *s* type benchmark that its footprint. Figure 14 shows the cache partition sizes assigned to the cores when the *fnts* workload is simulated on GEM5 with the *s* type benchmark running on core 3. Indeed, a significant amount of space is allocated to core 3. Although the space is not utilized efficiently, it is the smallest partition among all other partitions. However, Figure 15 shows that *s* type's miss-rate was the highest and should be assigned the largest partition. This is because core 3 had the smallest number of accesses made compared to other benchmarks. The number of accesses affect the Q and R matrices (see Section 4.5). Indeed a core with low number of accesses can have a high miss-rate. Therefore it is critical to account for not only the miss-rates but also the total number of accesses for a benchmark. PMCP accounts for both when making its estimates.

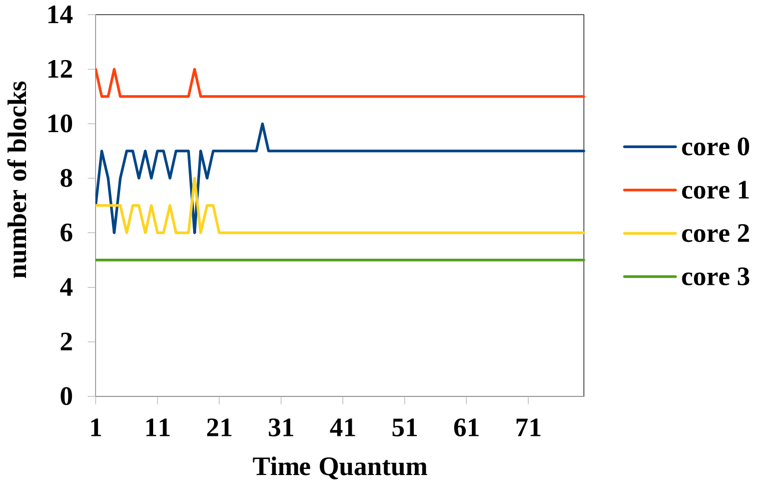


Figure 14: Steady state partition sizes in each TQ for the fnts workload. Both HLTP and HLSP are included in the figure.

Figure 14 also shows HLTP and HLSP as they occur on the timeline. Note that PMCP is executed during HLSP too. However, the variations in the estimated partition sizes are so small that they are rounded off to the nearest integer value. The spike in core 0's partition size during HLSP (TQ = 31) is because of the variation in miss-rate observed during the same TQ (see Figure 15). However since the miss-rate dropped in the consecutive TQs, the partition size dropped and is rounded off to the nearest integer.

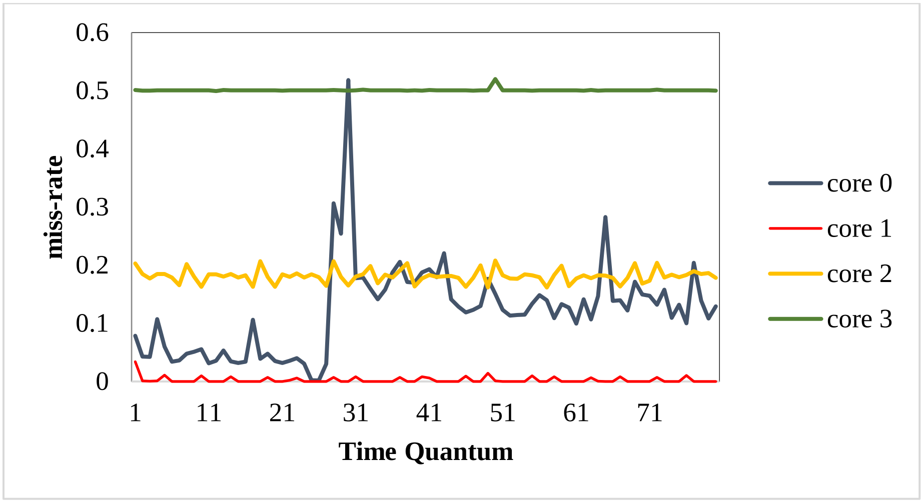


Figure 15: miss-rate measured at the start of each TQ for the fnts workload.

9.2 Evaluation of the fair-speedup metric

The baseline sharing mechanism for the shared bus and space in the L2 cache causes each thread in a multithreaded workload to observe a slowdown compared to when only a single threaded version of the individual workloads run on the processors. A widely accepted metric to quantify the slowdown is by comparing the performance of the entire workload running on a multithreaded processor to the single threaded individual workloads involved with exclusive access to the entire processor. Fair-speedup (also known as H-mean) is a way to quantify this slowdown when the same number of instructions can be executed for each individual workloads. Section 8.1 describes the experimental setup and dictates that the measurements are collected each time a thread completes 10 million instructions in HLSP. This ensures that the run time for each thread to complete 10 million instructions is recorded and utilized to evaluate $IPCmt_i$. IPC for the exact same 10 million instructions is collected by running single threaded benchmarks individually on the same simulated processor ($IPCst_i$) individually. *IPCmt* and *IPCst* is measured for all *n* threads running on the processor.

$$fair - speedup = \frac{n}{\sum_n {IPCst_i}/{IPCmt_i}}$$

Figure 16 compares the fair-speedup between a system where the L2 cache is partitioned by PMCP and a system with baseline shared cache. Each cluster represents a distinct test case. The organization of the figure follow the same rules as described in Section 9. Among all the workloads, a processor with PMCP on L2 cache was fairer by a maximum of 5.5% and an average of 2.3% better than a processor with standard shared L2 cache.

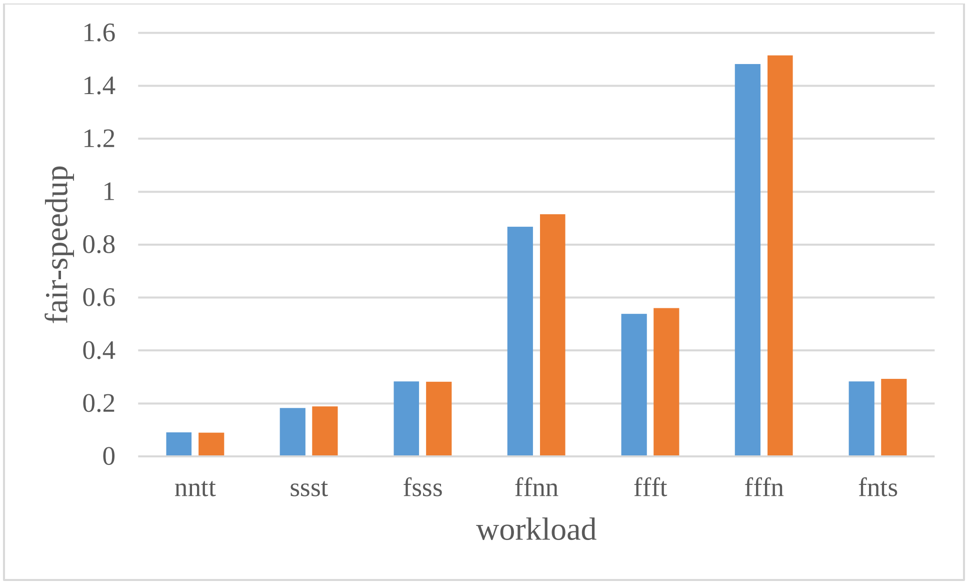


Figure 16: Fair-speedup for various workloads simulated. The left bar in each cluster represents the IPC for a processor where the with standard LRU replacement in the L2 cache. The right bar in each cluster represents the IPC for a processor where the L2 cache is partitioned.

10. IMPROVING PMCP BY EVALUATING PERFORMANCE MODEL DYNAMICALLY

Currently, the performance model used in PMCP is based on the √2 Rule (equation 15). Although it is a good model it certainly generalizes the performance of all the processes running on the system. It's been documented [15] that $0.3 < \alpha < 0.7$ and $\boldsymbol{M_0}$ can be different for each process running on the cores. In addition, equation (15) does not address processes with discontinuous miss-rate curves directly. Despite these limitations, in its current form PMCP is capable of predicting cache partitioning schemes such that throughput and fair-speedup are improved in most cases. However, it requires certain empirical evaluations of the error covariance matrices of the processes running on the cores. Empirically designed error covariance models allow PMCP to customize to the active threads data access patterns; but it limits PMCP's performance in cases where no prior knowledge is available for the user to design models for the error covariance matrices. A more accurate performance model need not rely on models for the error vector ($\boldsymbol{w_k}$ in equation 17 and $\boldsymbol{v_k}$ in equation 18). They can be treated as Gaussian error. But with limited information about the different processes running on the system, designing an appropriate system model is not trivial.

Gradient Projection (GP) is an optimization problem that has the potential to determine miss-rate curves with respect to cache sizes (performance model) dynamically with limited knowledge of the data access patterns of the active threads. GP starts with an initial miss-rate curve and updates the curve iteratively based on measurements from the cache. To do this GP redefines the performance model by including a control vector *u(k)* (equation 21) which is determined in real-time.

$$\boldsymbol{M(k+1) = M(k) + u(k)} \tag{21}$$

Where *M(k)* represents the miss-rate for the thread under consideration when cache size equals *k*. In other words, $0 < k \leq C_t - 1$ and $k \in I$. C_t is the total cache capacity.

GP estimates $u(0), u(1), \ldots, u(C_{t-1})$ that is used to update $M(1), M(2) \ldots M(C_t)$ iteratively while optimizing a relation, or a rule (called a *functional*) based on measured miss-rates while maintaining the boundary conditions of the variables *M* and *u*.

10.1 Defining the functional

A functional *J* is defined as a rule of correspondence that assigns to each function *f* in a certain class **Ω** a unique real number. **Ω** is called the *domain* of the functional and the set of real numbers associated with the functions is called the *range* of the functional. Note that **Ω** (the domain of the functional *f*) is a class of functions. In other words, a functional *J* is a "function of functions". By relating functions to real numbers the relationship between two functions can be quantified and evaluated.

PMCP can include GP in designing the performance model by selecting a functional ***J*** (equation 22) in the form of a mean square error function that compares the measured miss-rates and the miss-rates evaluated in the previous TQ for each thread.

$$J = \sqrt{\frac{\sum_{k=0}^{C_t}\left(M_{measured} - M(k)\right)^2}{C_t}} \tag{22}$$

where $M_{measured}$ is the measured miss-rate in the current TQ.

10.2 Setting up GP

To get the approximate miss-rate curves of all the threads in the system it is essential to understand how GP generates miss-rate curves for a single thread first. Assume a single core processor with a single thread running on the system. However, the user is unaware of the last-level shared cache's data access patterns of the thread or the miss-rate curve. In such cases, the user is limited to certain general information about the miss-rate curves. For all practical purposes, the cache can be assumed to be limited in capacity. Therefore the miss-rate curve is evaluated by scaling the cache size c, such that

$c \epsilon I$ and

$$0 < c \leq C_t \tag{23}$$

where

C_t = cache size

I = set of all integers

Another information available to the users is the limits on the miss-rate M of the thread i.e.

$M \epsilon R$ and

$$0 \leq M(k) \leq 1 \tag{24}$$

Where R is the set of all real numbers.

Note that miss-rate is always assumed to be 1 when $\lim_{c=0} c$. Using GP, over several TQs, PMCP has the capability of evaluating very accurate miss-rate curves. As mentioned in equation (21),

$$M(k) = M(k-1) + u(k-1)$$

$$M(k) = M(k-2) + u(k-2) + u(k-1)$$

$$M(k) = M(k-3) + u(k-3) + u(k-2) + u(k-1)$$

.

.

.

$$M(k) = M(0) + \sum_{l=0}^{k-1} u(k-l)$$

Without loss of generality, *M(0)* can be assumed to be always 1. Therefore based on the above equation

$$M(k) = 1 + \sum_{l=0}^{k-1} u(k-l) \tag{25}$$

Since $0 \le M \le 1$, the limits on *u(k)* can be defined as

$$-1 \le u(k) \le 1 \tag{26}$$

Equation (24) can be modified, by substituting equation (25) in equation (24), as

$$0 \le 1 + \sum_{l=0}^{k-1} u(k-l) \le 1 \tag{27}$$

In addition *J* can be modified by substituting equation (25) in equation (22) as

$$J = \sqrt{\frac{\sum_{k=0}^{C_t} \left(M_{measured} - \left(1 + \sum_{l=0}^{k-1} u(k-l)\right)\right)^2}{C_t}} \tag{28}$$

Equations (26) and (27) define the limits on the control vector *u(k)*. They can be summarized as

$$1 + \sum_{l=0}^{k-1} u(k-l) \ge 0$$

$$-\sum_{l=0}^{k-1} u(k-l) \ge 0$$

$$u(k) + 1 \ge 0$$

$$-u(k) + 1 \ge 0$$

GP requires all the coefficients to be normalized. Therefore the above constraints are rewritten as

$$\frac{1}{\sqrt{k}}\left(1+\sum_{l=0}^{k-1} u(k-l)\right) \geq 0$$

$$-\frac{1}{\sqrt{k}}\sum_{l=0}^{k-1} u(k-l) \geq 0$$

$$u(k)+1 \geq 0$$

$$-u(k)+1 \geq 0$$

Since *0<k≤C*$_t$, there are *4C*$_t$ constraint equations. These constraints can the combined together in a vector and represented as

$$\begin{bmatrix} L(1/\sqrt{k})_{C_tXC_t} \\ L(-1/\sqrt{k})_{C_tXC_t} \\ I_{C_tXC_t} \\ -I_{C_tXC_t} \end{bmatrix}_{4C_tXC_t} * \begin{bmatrix} u(0) \\ u(1) \\ \vdots \\ u(C_t-1) \end{bmatrix}_{C_tX1} + \begin{bmatrix} 1/\sqrt{k}_{C_tX1} \\ 0_{C_tX1} \\ 1_{C_tX1} \\ 1_{C_tX1} \end{bmatrix}_{4C_tX1} \geq [0]_{4C_tX1} \qquad (29)$$

Where *L* is a lower matrix and *I* is the Identity matrix. Therefore the problem of evaluating the miss-rate curve can be formulated as – Find the control values that satisfy equation (29) and minimize the function represented by equation (28). Typically, *J* (equation (28)) is minimized using the Gradient Projection Method until the norm of the difference of two consecutive control vectors is less than a predefined value δ. In other words, end the iterative process at the *ith* iteration if

$$\left\|u^i-u^{i-1}\right\| \leq \delta$$

Note that the entire process of GP is evaluating the values of *u(k)*. The functional to be minimized and the constraints are represented in terms of *u(k)*. The values of *u(k)* is then used to evaluate *M(k)* using equation (25). It should also be noted that the exact function of *M* is never evaluated. GP simply evaluates the values of M for a set of discrete points $0<k\leq C_t$.

In processors where multiple threads run simultaneously, GP can be scaled easily by evaluating the miss-rate curves for all threads as if the threads are mutually exclusive. Indeed this can be used in conjunction with PMCP to evaluate optimal cache-partitioning strategies dynamically. Note that, if GP takes *t* number of iterations to converge to the miss-rate curve, the computation complexity of the GP would be $O(ntC_t)$ where n is the number of threads. Typically *t* is a very small number (less than 10) and C_t, is constant in a processor. Therefore, in a real system, the number of threads running on the processor are the dominant factor in the scaling of the computations of GP.

11. CONCLUSION AND FUTURE WORK

As shown in this document, PMCP has the potential to improve the performance of the system by partitioning the last level shared cache such that cache utilization is optimized. PMCP is a light-weight and scalable technique that uses both statistical models and measured observations to update the size of cache partitions and the weighing matrices (error covariance matrices) dynamically. By iteratively updating the two factors, PMCP makes predictions dynamically without needing information in advance about the data access patterns of the combination of workloads before the.

Experiments were designed to test PMCP on GEM5, an event-driven multiprocessor simulator. The simulation results demonstrate that PMCP can partition the cache efficiently and improve the throughput of the processor by as much as 35% and fairness by 5.5%.

Although the results were satisfactory, a closer examination of the partition sizes revealed that the partitions could be allocated in a better way to further improve the overall processor's performance. The suboptimal partitioning is the consequence of the error covariance matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$'s empirical modelling that need not cater to every type of workload. If the performance model is realized dynamically, PMCP's dependence on the empirically modeled error covariance matrices can be avoided and the covariance matrices can be treated as Gaussian noise. A Gradient Projection method has been proposed to evaluate the performance model dynamically. Note that the performance model only estimates the distinct miss-rate values for the benchmarks when the cache size is scaled. It does not evaluate the mathematical model for the performance of the system.

Theoretical the GP method of evaluating performance model holds a lot of promise. A study of the timing effects of such an update in PMCP is necessary along with the study of the physical implications and requirements of PMCP on the processor. However, in its current state, this research should have presented enough evidence for an alternative *Control Theory* based solution to optimizing cache utilization dynamically and, perhaps, extend the framework towards other resources' utilization.

REFERENCES

[1] N. Beckmann and D. Sanchez, "Jigsaw: scalable software-defined caches", in Proceedings of the 22nd international conference on Parallel architectures and compilation techniques, October 07-07, 2013, Edinburgh, Scotland, UK

[2] R. Manikantan, Rajan, K.; Govindarajan, R., "Probabilistic Shared Cache Management (PriSM)," in Computer Architecture (ISCA), 2012 39th Annual International Symposium on , vol., no., pp.428-439, 9-13 June 2012

[3] D. Sanchez , C. Kozyrakis, “Vantage: scalable and efficient fine-grain cache partitioning”, in Proceedings of the 38th annual international symposium on Computer architecture, June 04-08, 2011, San Jose, California, USA

[4] C. Wu and M. Martonosi. "A Comparison of Capacity Management Schemes for Shared CMP Caches." In Proc. of the 7th Workshop on Duplicating, Deconstructing, and Debunking, 2008.

[5] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium in, vol., no., pp.367-378, 16-20 Feb. 2008

[6] D. Tam, R. Azimi, L. Soares, and M. Stumm. "Managing shared L2 caches on multicore systems in software." In WIOSCA'07, Jun. 2007.

[7] M. Woodside, , T. Zheng, and M. Litoiu, "Service System Resource Management Based on a Tracked Layered Performance Model", in Proceedings of the 2006 IEEE International Conference on Autonomic Computing. 2006, IEEE Computer Society. p. 175-184.

[8] M.K. Qureshi and Y.N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches", in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. 2006, IEEE Computer Society. p. 423-432.

[9] S. Kim, D. Chandra and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", in Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. 2004, IEEE Computer Society. p. 111-122.

[10] G.E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning", in Proceedings of the 15th international conference on Supercomputing. 2001, ACM: Sorrento, Italy. p. 1-12.

[11] D.Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches" in Design Automation Conference, 2000. Proceedings 2000. 2000.

[12] H.S. Stone, J. Turek, and J.L. Wolf, "Optimal partitioning of cache memory." Computers, IEEE Transactions on, 1992. 41(9): p. 1054-1068.

[13]C.K. Chow, "On Optimization of Storage Hierarchies" IBM Journal of Research and Development, 1974. 18(3): p. 194-203.

[14] C.K. Chow, "Determination of Cache's Capacity and its Matching Storage Hierarchy" in Computers, IEEE Transactions on, 1976. C-25(2): p. 157-164.

[15] A. Hartstein, V. Srinivasan, T.R. Puzak, and P.G. Emma, "Cache miss behavior: is it $\sqrt{2}$", in Proceedings of the 3rd conference on Computing frontiers. 2006, ACM: Ischia, Italy. p. 313-320.

[16] R.E. Kalman, "A New Approach to Linear Filtering and Prediction Problems" Journal of Basic Engineering, 1960. 82(1): p. 35-45.

[17] D. Simon, "Kalman filtering with state constraints: a survey of linear and nonlinear algorithms" Control Theory & Applications, IET, 2010. 4(8): p. 1303-1318.

[18] G.E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory" J. Supercomput., 2004. 28(1): p. 7-26.

[19] F. Guo, H. Kannan, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis. From Chaos to QoS: Case Studies in CMP Resource Management. ACM SIGARCH Computer Architecture News, 35(1), 2007.

[20] F. Guo, H. Kannan, L. Zhao, R. Illikkal , R. Iyer, D. Newell , Y. Solihin , C. Kozyrakis, “From chaos to QoS: case studies in CMP resource management”, ACM SIGARCH Computer Architecture News, v.35 n.1, March 2007

[21] L. R. Hsu, S. K. Reinhardt, R. Iyer, S. Makineni, “Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource”, Proceedings of the 15th international conference on Parallel architectures and compilation techniques, September 16-20, 2006, Seattle, Washington, USA

[22] S. Srikantaiah, M. Kandemir, Q. Wang, "SHARP control: Controlled shared cache management in chip multiprocessors," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on , vol., no., pp.517-528, 12-16 Dec. 2009

[23] M. Neisser, S. Wurm, "ITRS lithography roadmap: 2015 challenges." in Advanced Optical Technologies, 4(4), pp. 235-240. Retrieved 22 Nov. 2016

[24] J. Wu, Y. L. Shen, K. Reinhardt, H. Szu, B. Dong, "A nanotechnology enhancement to Moore's law." in Applied Computational Intelligence and Soft Computing, Frb. 2013