Modeling Data Center Co-Tenancy Performance Interference

Wei Kuang
Michigan Technological University, wkuang@mtu.edu

Modeling Data Center Co-Tenancy Performance Interference

By

Wei Kuang

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Computer Science

Michigan Technological University

2018

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

DISSERTATION CO-ADVISOR:	*Dr. Zhenlin Wang*
DISSERTATION CO-ADVISOR:	*Dr. Laura E. Brown*
COMMITTEE MEMBER:	*Dr. Steven M. Carr*
COMMITTEE MEMBER:	*Dr. Min Wang*
COMMITTEE MEMBER:	*Dr. Timothy Havens*
DEPARTMENT CHAIR:	*Dr. Min Song*

Contents

List of Figures

List of Tables

Preface

Author contribution statement: A partial content in Chapter 1 and the Chapter 3 in this dissertation uses contents from a published conference paper [61] by the author. The author conducted all the experiments and both advisors, Dr. Zhenlin Wang and Dr. Laura Brown provided advices on the work and they are fully aware that the work is reorganized and written in this dissertation. For a detailed acknowledgement of the permission granted by the holder of the conference paper copyright, please see Appendix.

Abstract

A multi-core machine allows executing several applications simultaneously. Those jobs are scheduled on different cores and compete for shared resources such as the last level cache and memory bandwidth. Such competitions might cause performance degradation. Data centers often utilize virtualization to provide a certain level of performance isolation. However, some of the shared resources cannot be divided, even in a virtualized system, to ensure complete isolation. If the performance degradation of co-tenancy is not known to the cloud administrator, a data center often has to dedicate a whole machine for a latency-sensitive application to guarantee its quality of service. Co-run scheduling attempts to make good utilization of resources by scheduling compatible jobs into one machine while maintaining their service level agreements. An ideal co-run scheduling scheme requires accurate contention modeling. Recent studies for co-run modeling and scheduling have made steady progress to predict performance for two co-run applications sharing a specific system. This thesis advances co-tenancy modeling in three aspects. First, with an accurate co-run modeling for one system at hand, we propose a regression model to transfer the knowledge and create a model for a new system with different hardware configuration. Second, by examining those programs that yield high prediction errors, we further leverage clustering techniques to create a model for each group of applications that show similar behavior. Clustering helps improve the prediction accuracy of those pathological cases. Third, existing research is typically focused on modeling two application co-run cases. We extend a two-core model to a three- and four-core model by introducing a light-weight micro-kernel that emulates a complicated benchmark through program instrumentation. Our experimental evaluation shows that our cross-architecture model achieves an average prediction error less than 2% for pairwise co-runs across the SPECCPU2006 benchmark suite. For more than two application co-tenancy modeling, we show that our model is more scalable and can achieve an average prediction error of 2-3%.

Chapter 1

Introduction

Today's computation has been continuously moved into the Cloud. Data centers often employ thousands of heterogeneous modern machines, providing their customers with elastic, scalable, and affordable computing and storage services. A modern machine is typically multi-core or many-core and hence able to host multiple applications or virtual machines (VMs). Co-locating applications or VMs can help improving server resource utilization and reduce operating costs. Certain resources, such as disk and main memory, can be partitioned across VMs on the same machine. However, some shared resources, such as the last level cache and memory bandwidth, are often implicitly shared and cannot be easily partitioned. Interference due to shared resources contentions can lead to performance degradation, a typical problem resulting from co-tenancy.

If the performance impact of co-tenancy is not predictable for latency-sensitive applications where response time or throughput is critical, data centers often have to dedicate whole machines to guarantee QoS even when this would result in under-utilization of the machine. Recent studies in co-tenancy and data center scheduling have made steady progress toward scheduling co-run applications to guarantee the performance [25, 26]. However, there still lacks an accurate model that can predict co-run performance across machines of different architectures.

Modeling and predicting co-tenancy interference is critical for data center job scheduling and guarantees QoS. Recent studies rely on two metrics, *sensitivity* and *pressure*, to quantify co-tenancy interferences [70]. Sensitivity measures how an application's performance is affected by co-run applications. Pressure measures how much an application impacts the performance of co-run applications. This dissertation shows that sensitivity and pressure are both application- and architecture-dependent. We propose a regression model that predicts an application's sensitivity and pressure across architectures with high accuracy. Co-run performance degradation can be predicted accurately by using the models of sensitivity and pressure for the co-run applications. High-accuracy co-run performance modeling enables a data center scheduler to guarantee the QoS of the applications. Yet we find some programs' performance degradation are difficult to predict using this approach. Further study shows that cross-architecture patterns are different across different group of benchmarks. Clustering is used to categorize programs and separate models are constructed for each group to address the problem. Moreover, as there can be more than two cores sharing the last level cache and memory bandwidth. Thus, co-tenancy is not limited to just the pairwise case. It also includes the case for three cores and four cores. We propose a new micro-kernel design that accurately resembles actual applications in a co-run setting and can be used to model the performance slowdown in a contended environment.

1.1 Predicting contention for different architectures with regressions

Models that use metrics such as sensitivity and pressure can predict pairwise co-run performance accurately. However, an application's sensitivity and pressure will change as the architecture configuration changes. Therefore, such metrics must be measured on a per architecture basis. To reduce the cost of profiling, we can model an application's sensitivity and pressure, both of which are architecture dependent, by employing machine learning techniques such as regression modeling.

1.1.1 Modeling sensitivity and pressure with regression

The miss ratio curve is widely used in application characterization and serves as a heuristic for resource allocation, where the x-axis is the cache size assigned to the application and the y-axis is its cache miss ratio. Program performance degradation with respect to different levels of co-runner pressure can also be depicted as a function similar to a miss ratio curve. Performance degradation is actually the "sensitivity" of a program towards various levels of contention, where the x-axis is the co-runner's pressure score and the y-axis is the performance degradation scaled by solo execution time. Different applications react differently towards pressure, therefore the shape of the sensitivity curve can be varied. We have tried several classes of regression functions to fit the curve, including a linear model, quadratic model and a logistic model. We find that a logistic model of the form $\frac{c}{1+e^{-b(x-a)}}$ fits best across all benchmark programs with different shapes. Moreover, the parameters of the logistic function are easy to interpret. As for each co-running program, the performance degradation will reach a point where cache is saturated and will no longer decrease, and one of the parameter reveals this upper bound. On the contrary, the program will perform similarly as the solo execution cases without too much slowdown when co-runner's pressure is at a 'benign' range and will be sharply increased after a certain point. The second parameter depicts such inflection point. At last, the third parameter shows how fast is the slowdown towards pressure after the knee point. We use a synthetic program, which is a memory bubble that can inflate or deflate to imitate the last level cache contention, to profile each program's slowdown with the bubble pressure ranging from 0 to the size of last level cache. We then fit those sample points into a logistic function. The fitting accuracy across all the programs is over 99%.

1.1.2 Cross-architecture performance modeling with polynomial curve fitting

In order to learn and model how a program's sensitivity changes between different architectures, a training set of applications are profiled on both machines. To reduce the profiling cost, only representative applications are selected for profiling. We divide the SPEC CPU2006 benchmarks into training and testing sets, and profile all training programs on each machine. We use a logistic function to represent the sensitivity curve of each program, from which three parameters are extracted. In order to capture the transition pattern of those logistic parameters from one machine to the other machine, we fit the mapping of each parameter across two different architectures into a quadratic function to create the cross-architecture model. We then profile the testing program set on one architecture to generate logistic functions as their sensitivity curves. Using the logistic function parameters as the input, we predict the parameters of the logistic functions for each testing program on the other architecture to make final performance degradation prediction. The average prediction error is within 2%.

1.2 Improving prediction accuracy using clustering

The framework's performance depends significantly on how closely the bubble and reporters can resemble the actual programs. However, some programs are benign to each other and some programs are aggressive towards resources. A single micro kernel therefore cannot resemble every case. This results in modeling errors. Even though average prediction error is within 2%, certain programs have prediction errors around 10%. We examine the prediction procedure and find that programs can be clustered such that programs within a cluster share similar cross-architecture transition pattern while different clusters have different transition pattern. Therefore, we divided the programs into clusters based on their cache access behavior and construct a model for each cluster. With three clusters, the average prediction accuracy is improved and programs with high prediction errors are now have much more accurate prediction result.

1.3 Predicting performance for more than two core co-run scenarios

Co-run scenarios are not limited to only two cores. We move one step forward, aiming to solve the co-run prediction problem for more than two cores. We find there is a scalability limitation in our current approaches and propose using program-dependent bubbles to tackle this limitation. The merit of using sensitivity and pressure is to decouple the pairwise co-run performance prediction into a linear-solvable problem. However the profiling cost increases as the number of participating cores increases. Moreover, the combinations of co-run peers expand exponentially, making the approach infeasible in practice. The purpose of using a bubble is to provide a way to resemble the actual program, and the bubble comes with various pressure scores. Therefore, we need an additional step to measuring which one best mimics the actual application. However, we can eliminate the measuring process if we can design a program-specific bubble that perfectly resembles the actual application. Moreover, we can make predictions out of bubble-only co-run instead of program-bubble co-run by observing the hardware statistics of the bubble when it co-runs with other bubbles. We use Intel^{TM}'s Pin tool to analyze memory-related behavior of each program, identify hot code segments in the source code, and use that information to direct new bubble design. The new bubble has the fine-grain property such that it behaves almost exactly as the actual program does whether co-runners are present or not. With the newly designed bubbles, we can predict multi-core (3 or 4) co-run degradation still within 2% error and eliminate the process of measuring program bubble score, which greatly reduces profiling time.

1.4 Dissertation organization

The rest of this dissertation is organized as follows. In Chapter 2, we briefly cover relevant background knowledge and discuss related work. In Chapter 3, our cross-architecture co-tenancy contention performance modeling is presented. We will describe our implementation and experimental results. In Chapter 4, we investigate the lower prediction accuracy of some programs and show that clustering programs into different groups according their cache access behavior can significantly improve overall performance. In Chapter 5, we expand the framework into a more than two core co-run scenario. We also analyze the access behavior of each SPEC CPU2006 benchmark program by using the Intel's Pin tool. By identifying the hot code segments, we create program-dependent bubbles, which duplicate the access behavior of an actual program. The program-dependent bubbles are then compared with the bubbles we use in Chapter 3 and Chapter 4. We demonstrate that the newly designed bubbles show a very similar behavior as the actual programs do and adapt well to the changes of peer runners so that the prediction error is still within 2%. We conclude this dissertation in Chapter 6 by summarizing contributions, discussing the limitations of our approach and possible future work.

1.5 Summary of contributions

We make following contributions in this dissertation.

- A cross-architecture contention model that enables performance prediction on both source machine and target machine with different hardware configurations.
- A categorization of applications according to memory subsystem sensitivity and pressure. Category-specific modeling improves the prediction accuracy of cross-architecture co-tenancy performance degradation.
- A contention prediction model for more than two core scenarios. We propose a new micro-kernel design method that accurately imitates actual applications in terms of memory subsystem accessing behavior.

Chapter 2

Background and Related Work

2.1 Modeling contentions for shared resources

Modern architectures contain multiple processors on a single die so that several tasks can run simultaneously on the chip. To provide a certain level of isolation, each core has a dedicated L1 and L2 cache. However, applications usually require larger memory space, and a shared last level cache and main memory hierarchy can guarantee such requirements. Contention exists as long as applications share some resources. A wise job scheduling scheme makes good utilization of resources while at the same time prevents significant performance degradation due to contentions. The studies focus on resource allocation for contention environment has been a hot topic in the area. On one hand, a significant amount of prior work focuses on performance modeling, such as cache miss ratio modeling, memory bandwidth consumption modeling, and overall performance slowdown modeling. On the other hand, a great portion of prior work focuses on how to schedule co-run applications based on on-line or off-line heuristics so that system utilization and QoS are both satisfied whereas quite a few studies focus on pinpointing the pathological portion of the code and make appropriate modifications to alleviate contention. Related works will be given in the following section.

2.1.1 Scheduling or modeling various contentions with off-line or on-line profiling

For a Von Neumann architecture, data is first loaded into memory and fetched by the CPU for computation. On-chip cache can store frequently accessed data for such that loading cached data is much faster than directly fetching it from main memory. Therefore the performance of an application is strongly related to its cache hit/miss behavior. Early studies try to model how an application's cache miss ratio is affected by the cache size allocated to it. Mattson et al. [71] and Kim et al. [60] propose similar algorithms to calculate the cache miss ratio for a given cache size. The algorithm scans through the memory trace of a program and maintains a LRU stack, where the most recently used data is at top of stack and the least recently used data is at bottom. When a memory location is accessed, one can first calculate the distance between the accessed location and the top of stack, then move it to the top to ensure a LRU replacement policy. This distance can determine whether or not current access will be a hit, as data located at the bottom section of the stack might be evicted due to the limited cache size. In general, if the cache size is smaller than the reuse distance, the data access will be a miss, vice versa if the cache size is larger than the reuse distance for the hit case. Throughout the execution trace, one can collect such reuse distance information and build a histogram. And calculate the miss ratio to a specific cache size C by adding up all memory accesses whose stack distance is larger than C.

Recording a memory trace requires huge space, Bennett and Kruskal [6] find storing the last access of each memory access is enough to recover the reuse distance histogram, as all windows between a consecutive pair of last accesses have the same footprint and can be counted in a single step. Olken [79] improve the memory trace algorithm efficiency by organizing a simplified trace as a tree where each node stores the timing information of the data. However, as one tree node represents a unique memory access, the resulting tree can become huge in size. Approximate algorithm is introduced later to trim the tree. Zhong et al. [129] propose a scale tree, where each node represents a collection of datum. Each tree node has a time range attribute, and a size attribute, where the size indicates number of datum last accessed during the time range. A reuse distance histogram can be constructed from the scale tree where different reuse distance are grouped into bins so that the computational complexity can be reduced. Xiang et al. [113] improve the trace analysis algorithm by dividing trace into c sub-intervals, where each interval has k distinct data. The complexity is $O(ck \log m)$, where m is the total data size of the program. In later study, Xiang et al. [114] propose a linear algorithm for analyzing a length n trace without considering the data size m. The algorithm measures the distribution of time distance of all data reuse, the first and last access of each distinct data by scanning through the trace once with a hash table implementation. The algorithm results in $O(n)$ time complexity and $O(m)$ space complexity with accurate average footprint for all execution windows of the trace. Using the footprint and reuse histogram, one can predict the miss ratio for difference cache size. However, as a program can accept input of different size, the miss ratio curves (MRC) obtained through profiling using a specific input can not direct cache allocation for the same program using a different size input . Zhong et al. [128] propose a way to predict the MRC for various input size using two profiling runs. The histogram of two different runs are divided into the same number of bins, where each bin contains same proportion of total access. This is based on presumption that the proportion of each class of reference remains the same for different inputs, which is experimentally hold for most programs tested. Then use a linear model and solve the parameters with data collected from two runs. After that, one can recover an accurate the histogram of references for any input size and further create MRC for a specific input. The model predicts the MRC for different input sizes within 1-2% error compared with the actual MRC collected from hardware performance monitoring units.

Berg and Hagersten [7, 8], Eklov and Hagersten [33], Eklov et al. [32] propose StatCache, which is another approximated algorithm to derive the MRC based on reuse distance. The original reuse distance is to count number of distinct memory accesses in-between two same memory references. Statcache constructs a probabilistic model of memory references and using this model to derive the reuse distance. Using a sample of 0.01% of memory references, experimental results show that the shape of reuse statistic of the sampled set is very similar to the reuse statistic that uses every single memory access. In Shen [93, 92, 53]'s work, a similar probabilistic model is used to convert reuse time to an estimation of distinct memory references in-between two accesses of the same datum. The approximation accuracy is over 99% for cache block reuse and over 94% for element reuse, suggesting a strong connection between reuse time and reuse distance.

Reuse distance, footprint and miss ratio are related but the relationship between them is unclear. Ding et al. [115] revisit five metrics: footprint, the amount of data accessed during a time window; volume fill time, the average time a program access a given volume of data; inter-miss time, the average time between two cache misses; miss ratio, the fraction of references that cause a cache miss; and reuse distance, number of distinct memory access between current and previous access to the same datum. They demonstrate that those metrics can be derived from one another and verify the transformation through exhaustive testing. StatStack [34] employs the footprint theory and constructs MRC in linear time by converting reuse time to reuse distance. However, the space complexity of StatStack is $O(M)$, where M is the distinct elements in the memory trace. For storage workloads, which can last for days or weeks, the space requirement is huge. Counter Stack [110] and SHARDS [106] are proposed to reduce the space complexity. Counter Stack uses a probabilistic model to approximate MRC with guaranteed accuracy while at the same time uses sub-linear space. SHARDS uses a splay tree to track distinct data, and both algorithms achieve low level space requirement. Hu et al. [46] propose an average eviction time (AET) model which can quickly construct MRC with very low cost. Moreover, the algorithm can characterize shared cache behavior using footprint theory by modeling each application in a co-tenancy group.

Above researches build a cache model based on application's memory access traces. It assumes that only one application is running with a specific cache configuration, which is true for early architectures that only have single CPU core on them. On one hand, different programs can be executed in a sequential order, one can predict the cache misses for each program based on its own reuse distance analysis. On the other hand, programs can be executed in an interleave pattern, where all participating applications contribute to a memory access trace, single program trace analysis will fail to make accurate miss ratio prediction due to the interference.

Research on how a cache is affected by such interleave patterns has been done as early as 1980s [102]. Thiebaut and Stone [102] propose a probability model that can estimate the miss ratio if two programs A and B running on a core in an interleave pattern. Their model calculates the average number of occupations in cache $N\bar{X}$ for A, the average number of A's occupations evicted when B takes over control $N\bar{Z}$ and the average reload transient using A's footprint FP_A minus the difference between $N\bar{X}$ and $N\bar{Z}$. Their model also suggests that for a smaller cache, smaller associativity tends to have a lower cache miss rate since the entries are more evenly distributed to each congruent class and are less likely to overflow the cache. When cache size is significantly larger than the footprint, the reload transient of a higher associativity cache is smaller than the reload transient of a lower associativity cache, as rows that receive several lines because of a larger associativity are truncated in cache with lower associativity. Thus the footprint is smaller for a cache with a smaller associativity. Modern architecture now provides hardware support such that the shared cache can be partitioned. Programs given sole access to a cache of optimized size can run with a few or even without performance degradation. However, the profiling and simulation process required to construct the MRC is time consuming. Even with on-line instrumentation tools, it still significantly slows the execution. An on-line method is proposed by Tam [99] to quickly construct the MRC for a given program with only several hundreds milliseconds. The approach records only L1 data cache misses, which will become data accesses to L2 cache. With only a short burst of time during execution, the PMU records a memory trace and the reuse histogram is built from this trace. The MRC generated from the trace using the approaches described byMattson et al. [71] can be used to direct cache partitioning for pair-wise co-run program groups. To be specific, suppose the MRC for program A and B are MRC_A and MRC_B, total shared cache size is C, the solution is to find the cache size x, and $C - x$, such that $MRC(x)_A + MRC(C - x)_B$ are minimized. As is mentioned above, one cannot use single program reuse distance analysis to predict the situation in a contention scenario. For example, Xiang et al. [115] show experimentally that the estimation of miss ratio of co-run pair cannot be done through simple summation of individual miss ratio. However, the footprint is linear composable, and the group miss ratio can be derived from the group footprint, which can direct the co-scheduling of jobs on a multi-core architecture. Wang et al. [109] adopt higher order of locality theory [115] and explore symbiosis scheduling through on-line sampling. They propose an on-line based approach to quickly construct the group MRC. Their approach has a trade-off between sampling frequency and model accuracy. Adaptive burst footprint (ABF) sampling is employed in this paper. By setting up the threshold h, the sampling frequency is calculated each time such that it is long enough to measure the miss ratio greater than or equal to h. Moreover, by setting up the ratio between sampling and non-sampling interval, the overall cost of sampling is bounded, usually below 1% of the overall execution time of actual program. With accurate miss ratio prediction based on locality analysis, the MRC is a quantitative measurement for cache contention and can be used as a heuristic to direct job scheduling. However, the relationship between miss ratio and performance degradation is unclear. In [82], two metrics are given to depict the effect of cache on execution time. In fact, Sun et al. [98] show that a significant portion of the performance slowdown comes from the events unrelated to CPU and cache. Rather the memory controller, memory bus and DRAM modules cause significant slowdown, as contention on memory bandwidth can affect performance by increasing memory access latency and decreasing memory bandwidth [67]. This suggests that a memory bandwidth consumption model might be more accurate in directing co-tenancy scheduling over cache miss ratio model. Tools such as Cache Pirating [35] and Stressmark [116] can plot similar performance metrics such as CPI as a function of allocated cache size, and can be used to quantify the relationship between performance and memory bandwidth consumption in a contention scenario.

Yan [20] propose a profiling based approach to predict inter-thread cache contention on a CMP architecture. The MRC, with respect to different cache size, can be obtained through stack distance profiling. For a given cache size, one can estimate the cache miss rate. The authors present three methods to model cache miss prediction based on following: frequency of access (FOA) model, stack distance competition (SDC) model and a probabilistic model. The FOA model suggests that as programs co-run on a multi-core system can run at different pace in terms of cache access speed. Thus, one can estimate a program's effective cache size by calculating the proportion of its own access speed over the overall access speed. This can be problematic since programs co-run together can have different stack position shapes and reuse frequencies. The SDC model merges individual stack distance profiling into a combined one and calculates the winner at each stack position. After profiling, one can calculate the effective cache size based on this newly created, combined stack position information. The probabilistic model considers ones step further. Previous methods build on the assumption that an access will always be a hit as long as the reuse distance is less than given cache size. However, even the access of a most recent use can become a miss if there are enough misses introduced by other process. Thus the probabilistic model considers the probability an access will turn from a hit into a miss. They test this model on a CMP machine with 14 pairs of co-run groups. Experimental result show that the model prediction error is within 3.8%.

Xu, Chen, Dick, and Mao [116] propose the CAMP framework, which uses reuse distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each program to predict the effective cache size and instruction throughput estimation (IPC) when running concurrently and sharing cache with other programs. The merit of this method is that it requires no off-line profiling, operating system modification and additional hardware support. CAMP calculates the effective cache size of programs running in a contention memory subsystem at a steady state, and treats non-repeating phases separately. To be specific, given a carefully designed stress-mark (micro-kernel benchmark) which can be tuned to have a specified effective cache size, CAMP co-runs the stress-mark with some program P several times with different effective cache sizes. CAMP constructs the probability of effective cache sizes of program P using various performance statistics such as miss per access and further derives reuse distance.With the performance degradation curves for various cache size and effective cache size of the program in a specific co-run groups, one can predict the program's performance degradation. A similar approach is given byMars et al. [70]. On-line profiling of the reuse distance can reveal how much proportion of cache each contention peer can occupy. CAMP is tested on a two-core CMP machine, with 55 different combination of SPEC CPU2000 benchmark programs. The average prediction accuracy is 1.57%. Performance degradation results not only from cache contention but also congestion at off-chip memory hierarchy. Quite a few works focus on mitigating contentions by memory request scheduling [4, 58, 59] or using memory channel partitioning [51] or interleaving jobs to alleviate interference [55]. Only a few researches focus on actual performance modeling. Riseman et al. [85] observe that the number of instructions that can be issued per cycle is approximately the square root of number of instructions in the window. Michaud et al. [73] also come up with power-law relationship, which is essentially a square-root conclusion. This conclusion is later used for modeling basic/sustained CPI [54, 39, 40], followed by building up the model for miss-event penalty, including branch mis-prediction, instruction-cache miss and data-cache miss. Van Craeynest et al. [105] use similar stack idea but predict program performance changes when migrating it from a small (big) core to a big (small) core. The performance (CPI) is divided into two parts, base component and memory component. Base component is used to model instruction level parallelism (ILP) changes when migrating and memory component is used to model memory level parallelism (MLP) changes when migrating.

Eklov et al. [36] propose the Bandwidth Bandit to characterize how an application's performance is effected when memory bandwidth is decreased. The bandit engine co-runs with a target application and monitors the target's CPI while keeps increasing memory bandwidth consumption. The bandit engine is essentially a carefully designed micro-kernel that can inflate or shrink in terms of its memory bandwidth usage, we will use similar concept such as bubble and pirate interchangeably in this dissertation to refer such probing kernel. The bandit manages to only stress memory bandwidth without touching other shared resources. By running the bandit kernel at a specific rate in a controlled manner, the target application's performance changes due to contention for off-chip memory resources alone can be quantified. To obtain the memory access latency for 3 different types, which are page-hit, page-empty and page-miss, another carefully designed micro-kernel is run on the target architecture, this program traverses a link-list, where each access is dependent on previous access. Therefore, manipulating the layout of such link-list ensures the latency of a desired event can be recorded.

Subramanian et al. [97] focus on modeling performance degradation of memory bound programs in a contention environment. They observe that the slowdown of a memory bound application is linear proportional to the memory request service rate. Therefore the slowdown can be estimated by using the solo run memory request service rate divided by application's co-run memory request service rate. The application's co-run memory request service rate can be directly observed on the fly when it runs in an contention environment. The application's solo run memory request service rate can be estimated by assigning the program with the highest priority in accessing memory. For non-memory bound programs, which spend significant amounts of time performing computation, a parameter α is introduced which is calculated by ratio between the number of cycles stalled because of memory request and total number of cycles elapsed. These statistics can be obtained through PMU. Subramanian et al. [96] propose application slowdown model (ASM) in a follow up study. Prior studies assume an application's sole execution information is known and create a model to predict the slowdown in a contention scenario. In this study, programs are already running in a contended environment, and one must instead predict its solo execution time to calculate the slowdown, so that the progress information can be used for a fairness-aware scheduling scheme. They find that program performance is strongly related to its access rate to the shared cache. Therefore, slowdown prediction is reduced to estimating a program's access rate to the memory subsystem. Estimation of solo execution information is a two step process. The first step is to minimize memory bandwidth contention by assigning the request from a program of interest with highest priority at memory controller periodically. The second step is to quantify shared cache contention by using an auxiliary tag store to estimate cache misses due to contention. The auxiliary tag directory (ATD), first proposed by Qureshi and Patt [84], is a way to quantify utilization of shared cache and direct scheduling according to that metric. Du Bois et al. [28] propose PTCA, which make use such structure and estimate cache hit and miss for a solo execution using a co-run profiling result. Similar work also found in Ebrahimi et al. [30].

To predict the performance of memory bound applications, the application access pattern should be considered. However, as memory accesses issued from CPU cores are filtered by the private L1 cache and shared last level cache, it is extremely difficult to capture accurate off-chip memory access behavior. Instead, DRAM commands generated by the memory controller can be examined to derive the memory access pattern [22]. Moreover, as the minimum time delay depends on different cases of DRAM command pairs, the bank busy time is modeled as weighted summation of those different cases of minimum time delays, and the weight is determined by the frequency of occurrence for such pairs during program execution. Gulur et al. [44] propose ANATOMY, which uses a three stage queue model to evaluate memory performance, including the command bus, memory bank and data bus. The average memory request latency can be calculated as the summation of queuing delay and service time for all three stages. Wang et al. [107] consider multiple factors in predicting the memory bandwidth consumption of programs in a contention environment. They propose two models: Dramon-T and Dramon-R. Dramon-T is a memory trace based implementation for bandwidth usage prediction. To estimate bandwidth consumption, Dramon-T considers both program memory request rate and memory request service rate. Memory request rate prediction is reduced to program request issue rate prediction, which is limited by the program behavior, and DRAM service rate, which is limited by memory contentions. Memory request rate is reduced to calculating the reciprocal of average memory request latency, which is further reduced to estimation of memory request hit, miss, conflict ratios as these 3 cases will have different latency. Then a probabilistic model is used to predict if an arbitrary request is a hit, miss or conflict. Dramon-R is on-line bandwidth prediction model using PMU readings as input. Thus the probabilistic model can be replaced by hardware statistics. Both model achieve high prediction accuracy over 95% on portable benchmark proposed by McVoy and Staelin [72].

Sandberg et al. [87] make an interesting analogy of the cache behavior of two contended applications. As it can be viewed as two flow of liquid filling a glass. The liquids which overflow the glass correspond to the data evicted by the cache due to the replacement policy, and the concentration of each liquid inside the glass is proportional to its inflow rate. By this analogy, one may predict performance using each application's fetch rate and data reuse pattern and knowledge of how these factors change due to contention. In a steady state, the amount of program data evicted by the cache should equal the amount of program data fetched by the cache, which in other words, the replacement rate is equal to the fetch rate. If the replacement is random, the probability of replacement is proportional to the amount of cache allocated to the application. Cache pirating can generate the function needed. To be specific, an application's miss rate, fetch rate, hit rate, miss ratio, etc. as a function of cache size can be obtained through cache pirating. Many performance modeling consider average slowdown of a program when it co-run with other applications. However, programs are not always perfectly aligned and they can have phases that exhibit significantly different behavior, using average can be misleading and result in high prediction errors. In Sandberg et al. [88] follow up work, phase detection [29, 89] is employed to generate contention prediction model at a refined granularity. Programs in execution are divided into slices and the prediction model is applied to every slice. To reduce overhead, a dynamic window combines all slices within a phase together and apply the prediction model only once. To further speed up the process, the predicted slowdown of two specific phases from two applications can be cached and reused whenever the same pattern occurs again. This fine-grained prediction model results in average prediction error of 0.41% and maximum prediction error of 1.8%.

Mars et al. [70] propose "bubble-up" as an approach for tackling the contention prediction problem. Their approach relies on thorough off-line profiling of programs' reaction toward pressures. For a two core co-run scenario, for optimal scheduling, an oracle scheduler must obtain the performance degradation of all possible pairs-wise slowdowns, which results in a $O(n^2)$ time complexity. The framework decouples the pairwise performance prediction problem into measurement of sensitivity and pressure of each individual programs, thus lowering the complexity to $O(n)$ as oppose to $O(n^2)$ with brute force. By combining one program's sensitivity toward pressure and a co-run program's pressure score, the performance degradation of co-run group can be identified. Experimental result show that the prediction accuracy is within 2% for co-run groups randomly generated from the SPEC CPU2006 benchmark suite. A follow up paper proposes "bubble-flu" [120], which utilizes the bubble up methodology for coarse scheduling, and at the same time utilizes on-line profiling to monitor IPC changes. And the bubble-flux engine calculates the ratio between running and pausing of a program if the group performance degradation violate the QoS requirement as a finer adjustment strategy.

"Bubble-up" creates a performance model with respect to contentions which happen at memory subsystem as a whole. Eyerman et al. [38] introduces the idea of a speedup stack to describe how a program's performance is affected by a stack of factors such as on-chip cache contention and off-chip memory bandwidth consumption. Eklov et al. adopt the idea and use cache pirating [35], a technique similar to bubble [70] to create speedup stack models [37]. Their pirate kernel is carefully designed such that the fetch rate is close to zero so that this probe-kernel always stays in cache when it co-runs with a program of interest. Therefore it can extract the effect of cache contention from memory bandwidth contention, since program of interest will compete for cache with the pirate kernel but exclusively use memory bandwidth. Moreover, for a system with cache size C, one can run n copies of a program together, thus each program will actually receive C/n cache size in a stable state. As these instances will compete for off-chip memory bandwidth, one can compare the hardware statistics collected from this scenario with the one using cache pirating to extract the memory bandwidth factor for the speedup stack model.

Zhao et al. [125, 124] distinguish SPEC CPU2006 programs into three categories, cache-bound, memory-bound and cache/memory-bound, by plotting cache miss and memory bandwidth consumption metrics. The authors use a series of contention models, each representing one shared resources, to obtain the performance degradation through the aggregate pressure on these resources. For each program A_i being characterized, first pick three co-runners to form a four core co-run group, and use performance metrics to collect each program's individual pressure on cache and memory bandwidth. Then run them as a group and record program A_i's slowdown, therefore, one sample point is collected as a mapping of aggregate pressure and slowdown. A training set is constructed from 200 co-run group executions (all including program A_i). As the theoretical memory bandwidth is known in advance, the memory consumption range is then divided into three pieces to represent cache-bound, memory-bound and cache/memory bound cases. Therefore, the training data falls into one of the three categories based on their aggregate memory bandwidth consumption, from which a piecewise model is created. The overall approach is a two phase process. In the first phase, a collection of programs are selected and trained to determine the piecewise function, with the parameters undetermined. The second phase is to instantiate a model with application-specific parameters, using the abstraction model yielded by phase one as a candidate regression model, and collect the slowdown for only a small subset of co-run program to solve the parameters. The prediction error ranging from 0 to 10.2%, with a average value of 0.1%.

In de Blanche and Lundqvist [24], four performance degradation prediction models are compared, which are one slow-down based, two contention-based and one memory bandwidth consumption based methods. These methods, use the similar idea by running programs with “bubble” on shared resources multiple times to acquire sensitivity and pressure. The slowdown-based method “Memgen” performs better than other methods, and memory bandwidth consumption based method can also match the performance of “Memgen”, suggesting that using memory bandwidth, or performance as a whole, is better than using cache miss rate as a metric or heuristic in co-scheduling prediction tasks.

The above studies provide quantitative methods to measure application performance whenever contention presents. Other studies focus on characterizing the type of contention and use this information as a heuristic to direct co-tenancy scheduling.

Focusing on the co-tenancy scheduling problem, Snavely and Tullsen [94] introduce “symbiosis” as a performance metric composed of three parameters: ’diversity’, which describes whether or not all functional unit are running throughout execution; ’balance’, which describes whether under-utilization will lower system efficiency and over-utilization will result in conflict and jeopardize performance; and ’conflicts’, which describes how programs behave toward a shared resource. In their SOS framework, during a short sampling period, the permutation of all possible co-run groups are test on the system. The hardware counters monitor IPC, total conflicts on integer queue, floating point queue, .etc, L1 data cache hit rate, diverse of instructions, and a weighted sum score of diversity, balance and conflict can be calculated for each candidate group and the one with best score is selected. With the help of SOS framework, a system with random job arrival and departure gains as much as 17% performance over a framework without symbiosis.

For characterizing the source of contention between threads of an application or between multiple applications, one straightforward way is to run application with dedicated resource first as the baseline reference, then co-run a group of applications together multiple times, letting them share one shared resources each time. The co-run cases are compared with baseline to determine if that particular resource is contented by the peer runners. In Dey et al. [27], Parsec2.1, a multi-threaded benchmark suite is selected and the characterization of contention focuses on L1 cache, L2 cache and memory FSB. Hardware performance counters are used to quantify contentions and event UNHALTED_CORE_CYCLES is selected to determine both intra and inter application contentions. The architectures used in the experiment have private L1 cache, and shared L2 cache for every two cores. Interestingly, the intra-application contention is not much for L1 and L2 cache; programs show performance improvement as they have data sharing. The contention on FSB result in performance degradation as there is an increase in memory bandwidth consumption by multiple threads. For inter-application contention, as different applications each access their own data, contention is prominent compared to the intra-application cases. Even though profiling contention over multiple applications is time consuming, characterization of applications provide significant information for scheduling.

Co-scheduling of jobs can result in contention, and with a smart scheduling scheme, the overall performance degradation due to such contention can be significantly reduced [52]. Some studies suggest using reactive scheduling, which is runtime trials, keep changing the co-runner of a program to record its performance degradation towards different peers and then schedules those programs that are benign to each other together. This method can serve as the base of an on-line, lightweight scheduling algorithm. However, without knowing the optimal scheduling solution, it is hard to compare the reactive scheduling method with the optimal one, and whether or not there will be significant improvement if one scheduling adjustment is made. Other studies try to predict performance of programs when co-tenancy exists and with that knowledge one can pro-actively schedule programs by minimizing the overall co-run performance degradation. The paper [52] first demonstrates that such a scheduling problem can be solved in polynomial time if the group size is fixed to two, otherwise it is NP-complete problem. The authors give a scheduling scheme based on minimizing the pair-wise degradation weights of a graph and also gives several approximate scheduling schemes that can provide fast yet close to optimal scheduling options. The first scheduling scheme is a hierarchical perfect matching algorithm. It is derived from the solution of two core co-run problem. First solving the dual-core pair scheduling in polynomial time. Then creates a new degradation graph with each vertex representing co-run pairs in the first step and apply a minimum weight matching algorithm on the new graph to obtain the solution to a quad-core scheduling solution. To generalize the algorithm, one can approximate the optimal solution for the K core co-scheduling problem by applying the minimum perfect matching algorithm $\log_2 K$ times. The second scheduling scheme uses a greedy algorithm. Note that for a naive greedy scheduling scheme, which first sorts all k-cardinality sets in ascending order of total degradation and always picks the minimum degradation group until the final solution covers every job, yields a poor scheduling result. The reason is clear, programs that generate less pressure are in-sensitive to pressure, and also benign to co-runners, are friendly applications. Thus, in a naive greedy algorithm, the top set is likely to contain friendly applications; after a while, this set is depleted as the scheduler has no choice but to pick jobs with aggressive memory behaviors. Therefore, a more reasonable greedy job scheduling algorithm is to first obtain the performance degradation of all k-tuple groups that contain each job, and sort the groups to get the 'politeness' of each job, then each time pick the co-run group with minimum degradation which contains the program with the current best politeness score while other programs in the co-run group has not yet been selected into the final scheduling set. This yields much better scheduling results as polite programs and unfriendly ones are scheduled together while at the same time total performance degradation is minimized.

The above study tried to use different grouping mechanisms to alleviate the effect of resource contention. Blagodurov et al. [11], Zhuravlev et al. [130] survey grouping mechanisms based on program behavior characterization, comparing these approaches and proposing both off-line and on-line solutions to the scheduling problem. There are three different grouping mechanisms discussed: the Stack Distance Competition (SDC) algorithm, the Animal class algorithm and the Pain algorithm. SDC is based on the profiling of the memory trace on a LRU cache. Rather than obtain the memory access (cache access) histogram of each individual program, one can monitor the access of two candidate co-run peers, decide who wins (cache hit) on a specific cache line position and mark the combined LRU stack position with the winner information. After multiple iterations, one can determine the effective cache size of each co-runner and make proper performance degradation prediction. The Animal class algorithm classifies programs into different categories in terms of their sensitivity towards contention. Turtle, sheep, rabbit and devil; these four types of animal categories are sorted in ascending order with respect to contention sensitivity. With stack distance profiling, one can put a program into the appropriate class and give each pair of animals a score, which serves as information for scheduling. Pain classification introduces the sensitivity and intensity of a program, where sensitivity depict how a program react towards pressure and intensity describe how much pressure a program can stress onto the shared resources. By combining the two metrics for a given co-run pair, one can calculate the pain score thus direct scheduling. Once again, the author make use of LRU stack distance profiling. At each stack position, one can associate a loss probability to indicate how likely a hit will become a miss when contention exists. Then scale the hit by the probability to calculate extra misses to obtain the sensitivity score. Intensity is measured as last level cache accesses per million instructions as the measurement. Thus the Pain score for a program A when co-run with a program B is the product of A's sensitivity and B's intensity. The pair Pain score of program A and B is the sum of their Pain scores. Experimental result show that the Pain and the animal class algorithms perform well, as the performance degradation is only slightly worse than perfect scheduling. However, the SDC algorithm is only slightly better than a random scheduling scheme. It is possible that because the SDC algorithm doesn't consider the access speed of the two co-run peers, and only focuses on cache contention without considering other factors that will affect performance. The authors run a series of experiments to breakdown the factors that result in performance degradation, and discover that cache contention itself does have an effect on performance degradation, but sometimes contention for the memory controller, front-side bus (FSB), and prefetcher resources play dominant roles in performance degradation. They found that cache miss rate turns out to be a good heuristic for performance degradation prediction in a contention scenario. Though Pain classification delivers the best performance, it has certain complexity when stack distance is calculated on the fly. In contrast, using last level miss rate as heuristic is much simpler and yields slightly worse result. Based on those observations, the author implements two scheduling algorithms using miss rate as a scheduling heuristic. The first is Distributed Intensity (DI), which uses stack distance profiling to estimate last level cache misses, then use the last level cache miss estimation to make a classification. The second one is Distributed Intensity On-line algorithm (DIO), which is a user level implementation of DI with only on-line profiling of last level cache miss as the heuristic to direct classification. Experimental result suggest that the DIO algorithm is within 2% of the optimal scheduling solution.

Xu et al. [117] propose a scheduling mechanism based on balancing memory bandwidth contention. The situation is simplified as there is no shared cache on their testing platform. In selecting a candidate job to be scheduled, traditional approach usually picks candidates using the following fitness function:

$$FITNESS^P = \frac{1}{|\frac{BW_{remains}}{CPU_{remains}} - BW_{required}{}^p|}$$

That is, the traditional approach always finds a job which maximizes the fitness function, which always finds the candidate job p, whose memory bandwidth requirement is as close to average memory bandwidth as possible. Experimental results show that performance degradation starts before the aggregated memory bandwidth reaches peak value. Thus an accurate BW_{remain} is necessary to make good scheduling decision, which in this paper, is estimated by $IdealAverageBandwidth$. Ideally, programs run on a platform with infinite memory bandwidth. Those programs will run without contention, with finishing time $idealturnaroundtime$, which is the shortest time a program can finish its execution in a contention environment. The ideal average bandwidth is the ratio between the total number of memory accesses and the ideal turnaround time. Combined with the fitness function mentioned above, one can schedule a job whose combined memory bandwidth requirement is close to the $IdealAverageBandwidth$. In other words, rather than targeting peak bandwidth utilization when scheduling a job, one should keep the total bandwidth requirement close to the ideal average bandwidth of the entire workload to avoid unnecessary contention. Feliu et al. [41] extends the idea to situations with multiple level of shared memory. They propose the PC-Degradation Cache-Hierarchy Contention-Aware Scheduler algorithm, which achieves almost double the average speedup compared with state of art memory-aware scheduling algorithms.

As memory contention has been identified as the main cause for system wide unfairness [76, 30, 77]. Xu et al. [118] propose a fair-progress scheduling (FPS) policy. FPS uses the PMU to monitor hardware statistics for programs being executed and derives forward progress throughout the scheduling quantum. To be specific, FPS monitors number of instructions a program has executed during the quantum, estimates the IPC of a program phase running alone during a time window of quantum length, and calculates the ratio between them to obtain the progress. To maintain fairness, the policy always picks the program with the lowest progress value and lets them run during the next scheduling quantum. The policy improves overall system fairness but has a slight overhead in system throughput.

Different programs have different access speeds to cache, and the program with the fastest access speed usually places more pressure than programs with slower access rates. However, giving programs with fast access rate a larger cache size might not improve performance. For example, video streaming access programs visit cache fairly fast, but the data is unlikely to be accessed again. Qureshi and Patt [84] argue that one should allocate cache based on utilization rather than demand, that is, when a program's speedup stops increasing when the allocated cache size increases, one should not allocate more cache to the program. They categorize different programs into low utility, high utility and saturating utility based on how performance changes when allocated cache size increases. Hardware performance counters can collect statistics while programs compete for shared resources. However, the events are per-core based and cannot reflect the interactions between these applications. Zhao et al. [126] propose CacheScouts, which motivates a hardware design to better understand cache behaviors among contention peers. Counters and data structures are added to the cache and are associated with monitor identity so that one can learn cache occupancy and cache interference/share per application. To reduce overhead, only partial cache sets are tagged with counter/structure as one can still obtain accurate cache behavior through set sampling [56, 103]. The framework provides insight for future architecture design for CMP machines. Implementing a circuit to monitor cache utilization will bring significant hardware overhead. For an N-way associative cache of size C, the cache is divided into C/N sets. To estimate cache way utilization, one can profile cache utilization for full N ways, and derive the situation by assigning less cache way to the program based on LRU property. To record hit miss information at each cache position, a counter need to be attached. The UMON circuit simplifies this by only attaching a global counter to each cache position for all sets, and UMON-SS further simplifies this by only monitoring partial sets to represent a full cache set view. As [56, 103] point out, only less than 10% of the cache set needs to be sampled to have an accurate estimation of cache behavior. For the two programs co-run case, the scheduling problem can be solved by finding a cache way partition such that the summation of cache misses is minimized. The experiments use a simulator that assigns certain cache ways to different applications. Result show that partitions based on utilization are better than partitions based on demand.

Page coloring can be used to ensure continuous virtual memory can be evenly distributed across cache [101]. Later it is realized that this technique can also be used to enforce that an application only uses a subset of the cache space, which is a form of cache partitioning. Studies [21, 66, 95] demonstrate that performance, as well as fair utilization of cache resources in co-tenancy contention cases can be improved by applying this page color constraint. As applications running on a machine change over time, a dynamic re-coloring mechanism is needed. Zhang et al. [123] propose a hot-page based coloring mechanism such that only those frequently used pages are mapped with colors to guarantee good distribution across cache. As hardware page table entry has an access bit that can be set whenever a page is accessed, and page access can also be captured by page fault event, these features are employed for hot page identification.

As program phase transition can result in different cache utilization over time, a static coloring technique cannot capture this dynamic behavior, on the other hand, dynamic re-coloring involves allocating new page, page copying, and reclaiming old pages, which will introduce significant overhead. Intuitively, each application should be assigned a different color to provide isolation, but this may lead to low utilization. Therefore, assigning color based on an application's cache demand is more reasonable. Ye et al. [121] propose a page-coloring mechanism, COLORIS, which is embedded into the Linux kernel similarly to [123] but use a new definition of hotness. Instead of traversing the page table to determine hot pages, they redefine hotness as the number of processes that share the same color. Colors are redistributed based on this demand by monitoring the cache miss rate of every process running on the machine. Initially, applications running on different cores are assigned to different groups of colors to enforce isolation, applications running on the same core share the same group of colors as they will run in an interleaved fashion, and thus there won't be contention between each other. During execution, the re-coloring engine emits a signal to expand the color assignment if process miss rate exceeds a threshold, or emits a signal to indicate an application can sacrifice subset of its cache partitions (color) if the cache miss rate below a threshold.

As cache partitioning is already supported by hardware, whether to partition or sharing the cache for a co-run group is discussed [13]. The scheduling problem is converted into a dynamic programing problem, such that, for a machine with cache size C and for each joining program p_i, the program is assigned with c_i cache that minimize the miss count of current program and the total miss count of an optimal partitioning for the first i-1 programs, whose allocated cache size is $C - c_i$. Suppose the scheduling of i programs with cache size k is $S_{k,i}$ and mc is miss count, then the scheduling problem is equal to solving the DP problem below:

$$c_i = argmin\{mr(S_{C-c_i,i-1}) + mr_i(c_i)\}$$

$$S_{C,i} = S_{C-c_i,i-1} + c_i$$

The group miss ratio can be derived from individual footprints using the approach discussed in [115].

The above research focus on performance modeling or scheduling in a contention environment. A sizable body of literatures focuses on identifying pathological portions in source code that will result in contention and makes appropriate modifications to negate the effect.

Even when a single application is running on a multi-core machine, contention can still happen, such as when a multi-threaded program is scheduled on multiple cores. One study [127] uses memory shadowing and on-line monitoring to identify true and false sharing of memory/cache for multi-threaded programs. The overhead is $5x$ slowdown but it is still better than using cache/memory simulator to simulate the memory hierarchy. Suppose a program uses two threads but binds them to different cores, it is possible that each thread updates a local variable that is mapped to the same cache-line, resulting in false sharing and eventually resulting in a slowdown in performance. Such a scenario is inevitable for some cases. A common way to tackle this is to schedule threads that are frequently updating the same data onto cores with lower communication cost. In this paper, author tracks memory behavior (cache-line ownership) of multi-threaded program at the granularity of cache-lines. On assumption that no more than 32 threads/cores are active simultaneously, one can keep a 32 bits bitmap to track the ownership of cache-line segment. If the bit is set, it means the corresponding core/thread has a copy of the cache-line or owns the cache-line. By checking the bitmap at every memory access, one can determine how much cache contention this particular program causes. Moreover, one can easily determine the instructions that trigger lots of cache-line invalidation and make further optimization. Eizenberg et al. [31] propose REMIX, a modified version of the Oracle HotSpot JVM that detects contention and fixes false sharing bugs at runtime. The framework distinguishes contention sources, such as true sharing, where multiple cores make contended access to same bytes within a cache line and false sharing which multiple cores access different bytes within the line. Similar to [127], a 64-bit map per cache line is maintained, where each bit represents one byte in a cache-line. Each thread writes to the bitmap to identify the access. Using this structure, it is fairly easy to distinguish between false sharing and true sharing. The framework can either automatically or manually add the annotation @*contended* to code, so that additional care can be applied to the annotated section to avoid false sharing. Padding is one common mechanism to avoid contention, but programmers are responsible for identifying the contended code. Moreover, the annotation can not be added into standard library or existing third party code. REMIX pads data only when it finds necessary at runtime. The framework is tested on various benchmark suite: Dacapo 2006 and SPEC JVM2008 do not have much contention, and receive no speedup, while Spring Reactor and LMAX Disruptor are high-performance inter-thread java messaging library programs with a lot of false sharing cases, with speedups ranging up to two times using the REMIX framework.

Tang et al. [100] make use of performance monitoring and regression modeling to identify pathological code regions for programs running in a shared environment and modify those code regions to make them less contentious, which in turn can significantly reduce performance degradation. They first collect hardware statistics and feed them into a regression model to calculate a contention score, and if the value exceeds certain threshold, it pinpoints the critical portion that results in shared resource contention. Whenever such location is identified, padding or nap insertion is applied to transform the code and alleviate the contention. Pinpointing contention region in source code is straight forward. while monitoring hardware statistics during execution to calculate the contention score, at the same time, the number of instructions executed is recorded during each sampling window. As the number of instructions is fixed for the entire execution, one can replay the program in a software instrumentation tool and locate the high contention code section matching the instruction counts, and apply padding or nap transformation to the code. Experimental result suggests that this contention-aware code modification/compilation scheme improves overall program performance by 21% and utilization by 36%. A similar code transformation scheme is proposed by Bao and Ding [5], where code regions of loops are tiled based on their locality, aiming to reduce inclusive cache misses due to contentions. Pathological program behavior adversely affects performance, those "impolite" behavior will in turn affect the peer during a co-run scenario. Yoo et al. [122] propose a two-phase decision tree based approach to identify problematic code segments. In the first phase, a set of micro-benchmark are created with parameters such as working set size and accessing pattern, which can be tuned to mimic certain pathological behavior. As the pathological behavior is known in advance, one can collect a set of hardware performance events whose attribute-class correlation (representative) is high and attribute-attribute correlation (redundancy) is low, to characterize program behavior, from which a decision tree is trained. In the second phase, actual programs are run along with hardware performance monitoring, and the data is fed into the decision tree to diagnose pathological behavior. Random forest are employed to not only avoid over-fitting problem but also identify most dominate pathological factor of a program in a time-slice during execution. Their framework can accurately identify pathological code as well as the type of pathological behavior compared to manual examination conducted by domain experts.

2.1.2 Modeling various contention with machine learning techniques

Machine learning, closely related to computational statistics, is a field in computer science that gives a machine the ability to learn patterns without being explicitly programmed. These techniques have seen great success in computer vision, robotics, economics and marketing, linguistics, and bio-informatics, and can be applied to performance modeling in contention environment. For example, authors [64] use artificial neural networks (ANN) and support vector machines (SVM) to model the performance of a VM-hosted application as a function of the resources allocated to the virtual machine (VM) and the resource contention experienced. The authors argue that while it is common practice to use hardware performance counter statistic to predict performance, they are difficult to use in a virtualized environment as those model specific registers are not exposed to user. There are two other ways to make predictions: using queuing and control theory and using machine learning. A control theory based approach is built upon the assumption that performance and resource allocation are linearly related. However, due to contention and non-linear application characteristics, this assumption results in poor prediction. Machine learning approaches are well explored in the literature. CARVE [63, 111] uses a regression model to relate memory allocations and performance. Kundu et al. [63], Wood et al. [111] create regression models for application resource allocation between a physical machine and a virtualized machine, but the prediction result is poor when applied to virtualized environments. Cohen et al. [23] propose a tree-augmented bayesian network to reveal the relationship between resource allocation and quality of service degradation. The authors use performance statistics to query the signature of an application, cluster programs with similar behavior, and find previous allocations of same type to guarantee Service Level Agreement (SLA). In a similar work, Bodik et al. [12] use a logistic regression model to replace the Bayesian network. Even though the above works help predict whether certain resource allocations to an application will result in SLA violation, they fail to make accurate prediction on how much performance degradation would be with given statistics.

In contrast, recent studies borrow the ideas from the Netflix challenge and recommendation system. Collaborate filtering is a technique employed by recommendation system to make predictions. The most common examples are video recommendations provided to uses by YouTube or merchandise suggested to interested users by Amazon. These methods usually involve large amount of big data and sparse data collected from a recommendation provider's database. Similarities are found by comparing one user's behavior with other users. It can be expressed in mathematic form as a 2-D array, where each row represents an user, and each column represents an item, and each entry in the matrix represents a specific user's score for a specific item. The matrix have dense part as it accumulates rating information from other user over time, and the matrix also has sparse portion as an active user only give partial scores on a subset of all items and the recommendation system tends to guess this user's score on other items and provides recommended items with higher scores. Singular Value Decomposition(SVD) is often employed to tackle above task. SVD is applied onto dense matrix D_{NxM}. The matrix D, $D = U\Sigma V^T$, where U_{NxN} and V_{MxM} are orthogonal and Σ_{NxM} is diagonal matrix. Column of U are eigenvectors of DD^T and column of V are eigenvectors of D^TD. Entries in Σ are eigenvalues ordered according to eigenvectors. With the SVD solution, we can estimate $\hat{D} = U_k\Sigma_kV_k^T$ is the best rank k approximation of D. In actual, D is sparse as some rows contain entries with missing values. And SVD is undefined if there are missing entries. One solution to this is to assign missing entries with weight either 0 or 1. And minimizing the weighted difference between approximation and actual values. Numerical optimization using gradient descent in U and V or Expectation Maximization(EM) are employed to solve the problem.

As contentions exist when co-scheduling happens, different applications react differently toward pressure. One can create a sparse matrix [25] where each row represents a different application, each column represents one type of the contention resources, and each entry corresponds to a score for the application's reaction towards a specific contention. The dense part is thorough profiling of training programs running against micro-kernels that stress different aspect of contention resources. The sparse part is the program need to be scheduled and only a small subset of the micro-kernels are profiled against application of interest so that SVD is applied and entry value can be estimated. With such framework, program contentions are quantified, not only the framework provides user with a valid co-run groups scheduling solution but also actually provides user with information how much performance degradation would be if the candidate scheduling group is executed.

Non-uniform memory access (NUMA) [119] is widely used for multi-socket machines. Though different from CMP architectures, accessing remote memory from a local processor can also result in contention and thus result in performance degradation. In this paper, the author proposes DR-BW, a framework for identifying bandwidth contention on NUMA machines using supervised learning techniques. By co-running applications with micro-kernels designed to stress memory bandwidth and measuring the number of remote memory accesses, remote access latency, and other hardware statistics, a decision tree structure is built to classify if an application is remote memory bandwidth friendly or a contention source. By tracking thread ID during a sampling period, the framework also identifies the critical section/instructions of a program that will result in remote memory access contention. This study once again shows that for a contention-related task, one can tackle it using three methods/tools: 1) using the PMU to collect hardware statistics on the fly to understand runtime behavior, 2) making use of micro-kernels to understand how application react with different level of contention and 3) if necessary, dividing applications into groups and treating each group separately to improve prediction/scheduling.

2.2 Machine learning: clustering and regressions

As machine learning techniques are frequently employed in the co-tenancy scheduling and modeling frameworks, a brief introduction to those approaches are given in this section.

Clustering is a group of algorithms, root from mathematics, statistics and numeric analysis, that gather data together who has similar properties or patterns [49, 74, 3, 2]. Putting data into groups might lose fine details but brings simplicity. Without explicit indication, clustering manages to reveal the hidden patterns among given data and it falls into the category of unsupervised learning from a perspective of machine learning technique. This is a well-studied field and those algorithms in the area has been put into practice of real-world tasks such as image processing, pattern recognition, data mining, biological analysis and even medical diagnosis. A various of clustering method are listed below.

- hierarchical clustering
- partitioning method

- others: grid based method, constraint based methods, scalable clustering, clustering in machine learning, clustering in high-dimension data

2.2.1 Hierarchical clustering

Hierarchical clustering in general builds a tree structure. Nodes sharing a common parent indicate they share some common property, while at the same time, those siblings further divides the cluster into finer details, which is one of advantaged of this method, as user can view the the clustering result at different granularity. Creating the hierarchical tree is straightforward by calculating similarity distance. However, the stop criterion for further split/aggregate the cluster is vague and usually is a user-defined value, and this can result in sub-optimal clustering result. Further more, once the data is assigned to a cluster, it is fixed and won't be able to move to other clusters, which loses some improvement opportunity.

Hierarchical clustering includes agglomerate and divisive clustering. Agglomerate clustering builds the tree bottom up: each data point is a unique cluster at initial stage and the algorithm tries to combine two or more data points together to form a larger cluster. The divisive, on the other hand, builds the clustering tree from top to bottom: all data points belong to a single cluster at the very beginning and algorithm splits the whole into half by calculating distance metrics. Calculating distance between two points is straightforward, but it needs to be generalized to the distance between two clusters. Linkage distance is proposed to serve as the measurement of (dis)similarity between two intermediate clusters. In this metric, the distance between every pair of points, one from each cluster is calculated to indicate how close the two clusters are. Variation exists as some metrics use the summation, some metrics use the average distance and others use the largest distance.

In the field of document clustering, the mathematical method Singular Vector Decomposition (SVD) is employed as a way to split the data into different clusters. SVD is known for collaborative filtering and recommendation system. Take document clustering as an example, the problem is first translated into a matrix, where each entry in the matrix is the frequency on an attribute in one document, then the algorithm splits the data by creating a hyper-plane passing through data centroid, and orthogonal to the eigenvector direction with largest singular value. In general, the data can be split into k clusters by considering k largest singular values.

2.2.2 Partitioning Relocation Clustering

Divisive clustering algorithm needs to calculate distance metric between clusters. However, sometimes the computational complexity of checking all possible subsets of a split is high and heuristics need to be employed. Partitioning relocation clustering coarsely separates data set into initial groups and iteratively refines the clustering by either adding in points into one group or moving points from one group to others. One way to do this is probabilistic clustering, where data are assumed to be randomly chosen from several models of different distributions. One can estimate the probability of the assignment of a data point to a specific cluster. Thus the overall likely-hood of all training data points is its probability to be drawn from a mixture of models. By maximizing the likely-hood function, one can solve the parameters of the model and refine the model and re-assign each data point with a new cluster ID if necessary, so that the clustering result is refined iteration by iteration. An other way is to establish an objective function so that the value of such function is minimized/maximized after each split iteration by iteration, which leads to the k-medoid and k-means algorithm. In the k-medoid clustering, a single data point is selected and represents the cluster that includes it. Points close to that medoid are considered as they belong to same group. k-means is another popular clustering method: it uses the average of all points in one cluster, which is the centroid, to represent the cluster. variation exists as one can use cluster radii or cluster standard deviation rather than mean of the cluster, which is more reasonable when dispersion exist. As part of this dissertation work employs the k-means algorithm to categorize programs into different groups according to their cache contention behavior. We will discuss more on implementation of the algorithm. Most commonly used k-means implementation is Lloyd's algorithm. In general, the algorithm alternatively runs in two steps such that for all clusters, the summation of squared distance of points within each cluster and its mean is minimized.

$argmin \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$

Initially, k points are selected as the centroid of each cluster.

- Assignment: For each data point, assign it to the cluster who has the smallest mean of squared euclidean distance.

- Update: After assigning each data point with label information, re-calculate the cluster centroid.

The algorithm keeps running until the result converges, which means the assignment of data points no longer changes.

Despite the popularity, k-means suffers from following drawbacks:

- The result is largely depend on the initial centroid selections. Always yield local optimal rather than global.

- The number of cluster, which is the value of k is not easy to choose.

- Algorithm sensitive to outliers.

Several attempts have been made to alleviate the problem such as:

- Randomly pick several subsets of the entire data set and assign random positions as the initial centroid guess on each of them and run k-means on these sub-systems to create multiple k-means clustering result. Selecting the centroid of the best subsystem as the initial guess and run k-means on whole data set.

- Rather than explicitly moving a data point from a cluster to another, soft assignment is used as each data point has a weight associated with a cluster. The method considers how well a data point fit into the candidate cluster, and it is known as harmonic means clustering algorithm.
- Other study also suggest using simulated annealing

Studies also suggest using k-d trees structure as cluster representation to accelerate k-means algorithm. Suppose data point has n dimensions. Then calculate the standard deviation and find the direction with largest value and split the space from this direction. Then iteratively split the sub-space and construct the k-d tree. After splitting the space into k sub-spaces. search each data point using the k-d tree and assign them with the cluster number whose representative node is the closest one to it. Similar algorithm such as x-means goes one more step further, it tries to determine the best k while in the process.

2.2.3 Others: Grid-based clustering

Grid based clustering is similar to k-d tree to some extent, but it focuses more on space rather than data set. The algorithm STING [108] splits the space and constructs an corresponding tree structure, the nodes store the statistic information of the data associated in the space, such as points count, and attribute-dependent measurements: mean, standard deviation, minimum, maximum, and distribution type. The algorithm Wave-cluster [90] borrows the idea from the field of signal processing, where the edge of data corresponds to the high frequency part of data and lower frequency with high magnitude part represents the dense part of data, which in other words, data points inside one of the clusters. With wavelet filtering, it highlights the high density area and blurs the boundary and those outliers. All these methods allow users to view the data space in different resolutions. It has the advantage of finding cluster of irregular shape and has low complexity when data attribute is low. On the other hand, the result of the algorithm is affected by the initial clustering assignment and the parameters such as the threshold that defines the connectivity.

2.2.4 Regression

In statistics, regression is a set of algorithms or processes aiming to learn and predict the relation between the dependent variable (value) and one or more independent variable (features) [75]. The dependent variable is usually a continuous value, when the variable is a discrete one (label), the estimation turns into classification rather than regression. In general, the data is fit into a model with unknown parameters and through the training process, those parameters can be learned by minimizing the difference between observed value and predicted value. And the model is expected to make correct estimation on dependent variable for new data with only independent variable information as long as they are drawn from the same distribution. There are several different regression models such as linear regression, non-linear regression, robust regression, step-wise regression and logistic regression model. A brief description of these models is given in the following section.

In linear regression, it is assumed that the relationship between dependent variable y_i and independent variables(can be multi-dimension input data) x_i is linear. However, the linear model can be applied to non-linear distributed data such as fitting 2-D data drawn from a circular distribution. To be specific, suppose the training data is of circular distribution and independent variables is expressed as coordinates x_1, x_2. The radius is R. It is clear that $x_1^2 + x_2^2 = R^2$. Therefore, one can transform the input vector from x_1, x_2 into x_1^2, x_2^2 and still apply the linear model to it.

Least squares is the most commonly used approach for linear regression model parameter estimation. It calculates the parameters by minimizing the sum of squared residuals, where residual is the difference between the observed y value and the one generated by the the linear function. The minimum of the sum of squared residuals is at the location where the gradient is equal to zero. Therefore, for a linear model with m parameters, there are m gradient equation. One can set the partial derivatives for each directions to zero and solve the equation, which will yield close form solution to the parameters.

Non-linear least square is used to fit data observations with a non-linear model. Setting the gradient to zeros still applies. However, the derivatives are functions of both model parameters and the independent variable, so there is not a closed solution to the gradient equation. With a initial guess of all parameters, the process iteratively refines them.

Least squares estimation is straight forward and easy to implement. However, it is sensitive towards outliers. Data points drawn from different distribution can yield same or even identical linear functions. As oppose to least squares estimation, robust regression is proposed to deal this problem. One way to make model less sensitive towards outliers, one can employ least absolute deviations rather than least squares estimations. Further study proposes maximum likelihood, it is robust to outliers in the dependent variable dimensions, but sensitive toward outliers in independent variable domain. Least trimmed squares is later proposed to overcome the problem. S-estimate is proposed as it is highly resistant towards outliers but is found to be inefficient. MM-estimate is proposed as it tries to take advantage of the resistance property of S-estimation while at the same time to be as efficient as maximum-likelihood estimation.

Logistic regression: In statistics, the distribution of dependent variable can be categorical. As the value can only be 0 or 1. And if the possible value is more than 2, it is called multi-nominal logistic regression. Logistic function estimates the probability of a given data belong to one of the categories. Stochastic gradient descent (SGD) is usually employed to perform the logistic regression estimation, which can be summarized as a following 2-step procedure: given a instance from the training set, first calculate logistic function value using the current values of the coefficients, then refine coefficient values based on the error in the prediction. To be specific, suppose $Q(w)$ is model function and w are the parameters need to be estimated. SGD update these parameters at each observations.

for $i = 1 : n$ (n observations)

$w := w - \eta \nabla Q_i(w)$

where η is the learning rate and $\nabla Q_i(w)$ is the gradient of the cost function at each observations.

2.2.5 Performance modeling using machine learning

Bitirgen et al. [10] focus on resource allocation for a quad-application group. They use artificial neural network to build a model between resource allocated to an application and its performance gain. To reduce the huge search space in training the network, modified hill climbing algorithm [104] is employed. Ould-Ahmed-Vall et al. [81] focus on creating a model between micro-architectural event statistics and program performance(CPI). As programs have different phases that exhibits independent behaviors, model for each specific phase should be trained separately, which is tackled by model tree approach. Model tree is an extension of regression tree and it divides the input space into tree structure and place predictive regression models at leaf nodes. To be specific, for each new instance, the tree structure is use to categorize the instance into corresponding class, and predict its CPI using the linear model stored at leaf node of corresponding class. The prediction error is 5%. Similar idea such as using hardware events to predict user request for TCP-H workload is proposed in [91]. And the study also discuss how correlation between request and hardware event statistics are affected due to contentions.

2.3 Performance monitoring and binary instrumentation

Previous studies mentioned in this chapter model contentions through either on-line profiling or off-line profiling. Each of these two methods have its own advantages and drawbacks. On-line profiling captures program statistics on the fly, it reflects program behavior much more accurate compared to off-line profiling, which often pass the program execution trace through hardware simulators or probabilistic model to acquire access behavior. Modern architecture equipped with performance monitoring unit, which are essentially several counters that can record specific event happened during the execution of a program. Each CPU core has 2-4 counters that can monitor over hundred of hardware events. The overhead of extracting hardware event statistics is low, and usually the PMU can monitor 4 events at a time. PAPI [14] is proposed to provide tool designer and application engineer a standard interface to use hardware monitor unit. PerfSuite [62] is a later proposed open source tool for linux user to explore PMUs. OProfile [65] is a software that allows user to monitor hardware events by specifying the event name and unit mask, however, it can only monitor a few number of events simultaneously. Therefore one must program those counters to monitor different events in a interleave style to obtain statistics for more than 4 events. Intel VTune [69] provides such flexibility with multiplexing so that users can collect an arbitrary number of events. HPCTOOLKIT [1], PerfExpert [18] and periscope [43] are recently proposed that make use of these hardware counters to detect the bottleneck in parallel applications.

One the other hand, off-line profiling usually runs slow but it captures program behavior in a more understandable way as long as the parameters of simulator or instrumentation software matches the actual hardware architecture. Early researches such as Paraver [83] and VAMPIR [78] use instrumentation to examine and visualize execution trace of MPI applications. Intel's PIN [68] is a dynamic binary instrumentation tool that allows users to analyze programs at runtime by injecting instrumentation code into the compiled binary files at different granularity, such as instruction based or basic block based. It provides a rich collection of APIs that allows user to analyze memory trace, thus building up knowledge of a program's accessing behavior. Functionality such as tracking function calls as well as system calls and intercept signal are also provided by PIN. Other instrumentation software include DynInst [16], namoRIO [15], JIFL [80],.etc.

To sum up, previous studies focus on different scheduling schemes in alleviating contention due to co-tenancy. The heuristics behind each scheme are information acquired either from off-line traces or on-line hardware statistics. Performance degradation prediction can be a reasonable heuristic as it provides job scheduler or cloud administrator quantitative information on how applications are affected by different peer(s). In this dissertation, we propose a performance degradation model for two-core co-run scenarios. Various machine learning techniques are employed to categorize applications into different groups in terms of their contention behavior. We also adopt this idea to cluster programs according to their sensitivity/pressure characteristics and create dedicated model for each cluster. Moreover, most studies focus on two-core co-tenancy problems, we propose a new way to predict slowdown of applications in a group with more than one co-run peer. We test the approach for three-core and four-core co-run cases. Nevertheless, the approach can be applied to predict the performance of a co-run group whose size is equal to the number of cores on an architecture that shares the last level cache and memory bandwidth.

Chapter 3

Cross-architecture performance modeling for two cores

This chapter uses contents from a published conference paper [61] by author. The author conducted all the experiments and both advisors, Dr. Zhenlin Wang and Dr. Laura Brown provided advices on the work and they are fully aware that the work is reorganized and written in this dissertation. For a detailed acknowledgement of the permission granted by the holder of the conference paper copyright, please see Appendix.

3.1 Bubble-up approach for performance degradation prediction

The Bubble-Up approach is a general methodology designed to predict co-run application's performance interference [70]. Suppose there are two applications A and B that are co-run on a multi-core machine. Application A's performance degradation when co-run with B can be predicted if the *sensitivity curve* of A and the *pressure* of application B are known. This overall methodology is illustrated in Figure 3.1. First, application A is co-run with a *bubble* program. The bubble is a program that can "inflate" or "deflate" so that different levels of contention pressure can be added into the memory subsystem. Then by recording application A's performance degradation at each bubble pressure level, a sensitivity curve can be constructed. The sensitivity curve for application A plots bubble pressure on the x-axis versus A's performance degradation, which is measured as normalized execution time, on the y-axis. In Figure 3.1(a)-①, application A's sensitivity curve is plotted where the performance drops by 20%, 30% and 60% at bubble pressure of 1MB, 2MB and 10MB, respectively.

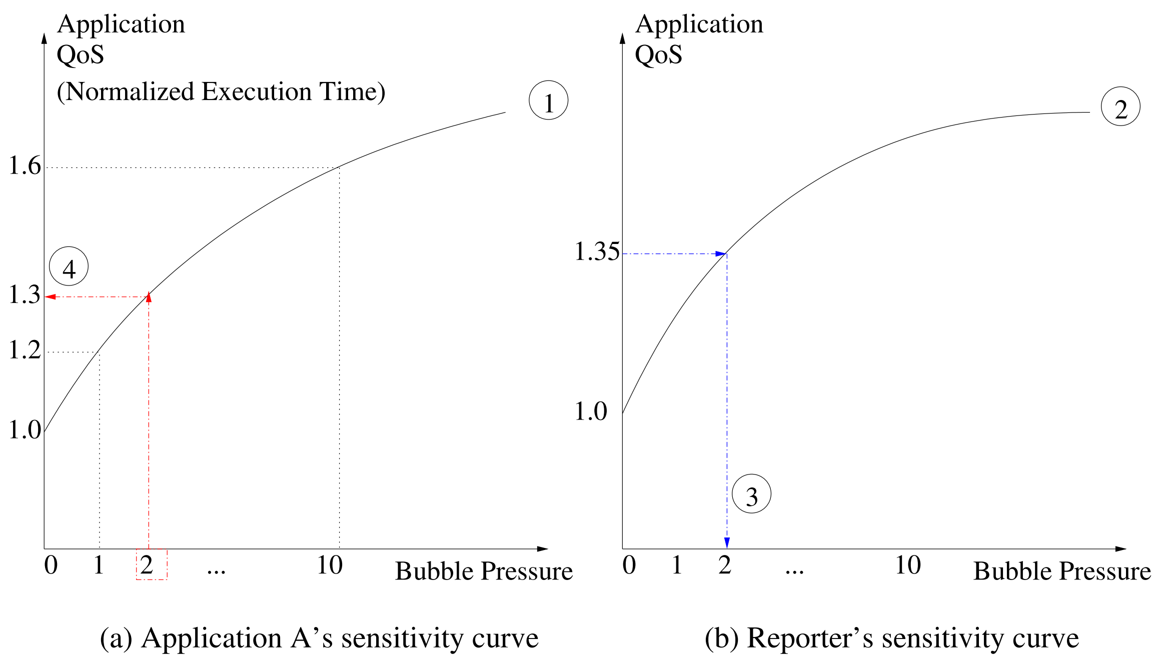

Figure 3.1: Bubble-Up approach to predict co-run performance degradation

Next, an application's pressure on the memory subsystem is characterized by using a program called a *reporter*. The reporter is a program designed to use both last level cache and memory bandwidth. The reporter's sensitivity curve is found by co-running the reporter and the bubble; Figure 3.1(b)-② shows the reporter's sensitivity curve. After the sensitivity curve of the reporter is obtained, co-run the reporter with application B. The observed performance degradation along with the reporter's sensitivity curve is used to determine application B's pressure score (application B gives as much as the corresponding bubble pressure towards the memory subsystem). In Figure 3.1(b)-③, the 1.35 performance degradation value from B co-run with the reporter is used to determine that application B's pressure score is 2.

Finally, with application A's sensitivity curve and application B's pressure, the performance degradation of A can be predicted when the two applications are run together. In Figure 3.1(a)-④

application A's sensitivity curve is again used with application B's pressure, 2, to predict that A's execution time will increase, 1.3 times, when it co-run with B compared to running alone, without co-run pressure. With the bubble-up methodology, the average prediction error is around 1% [70].

We implement the bubble and reporter as introduced by Mars et al. [70]. The current bubble and reporter design stresses the memory subsystem, with a focus on cache and memory bandwidth. Therefore, our design targets memory and CPU intensive applications. However, the general design in the bubble-up approach and the methodology proposed in this paper can be extended to predict co-run performance when an application's performance depends on other shared resources such as I/O, network bandwidth. The design focus then would be to find a bubble and a reporter that stress these shared resources, which we leave as future work.

3.2 System framework

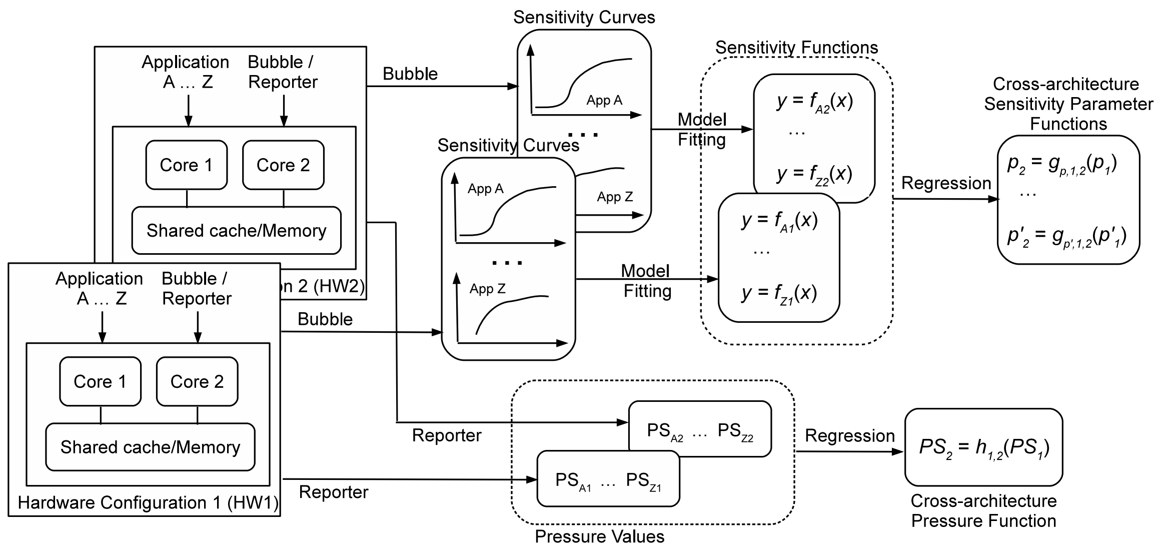

Figure 3.2: Development of cross-architecture prediction models for sensitivity parameters and pressure

The overall methodology is illustrated in Figure 3.2. The methodology begins with data collection. Profiling is used to collect sensitivity curves and pressures for a collection of benchmark programs, $X \in \{A, \ldots, Z\}$, on multiple architectures with different hardware configurations, $\# \in \{1, \ldots, N\}$. In general, benchmarks/applications are denoted with upper-case letters, A or B, architectures are denoted with numbers, 1 or 2, parameters of the sensitivity functions, f, with lower-case letters, a or b, and parameters of the cross-architecture functions, g, with lower-case Greek letters, α or β.

3.2.1 Cross-architecture sensitivity models

Simple functional models of the sensitivity curves are fitted for each program and architecture, $f_{X\#}$: bubble pressure $\rightarrow$ performance degradation. The functional representation of a sensitivity curve involves a small number of parameters, $p \in \{a, b, \ldots\}$. From all the benchmarks, a set of parameters is collected for each machine, $a_1 = \{a_{A1}, \ldots, a_{Z1}\}$, $a_2 = \{a_{A2}, \ldots, a_{Z2}\}$, $b_1 = \{b_{A1}, \ldots, b_{Z1}\}$, etc. Then, for each parameter, p, a function is fit describing the relations between architectures, $g_{p,1,2}$: $p_1 \rightarrow p_2$, the cross-architecture sensitivity parameter model.

Example: A linear function can be used to model the benchmark's sensitivity curves. Applications A's sensitivity curve on the first machine's hardware architecture, HW1, is represented as $y = a_{A1}x + b_{A1}$, where x is bubble pressure size, y is normalized performance degradation, a_{A1} and b_{A1} are the benchmark- and architecture-dependent parameters. The same benchmark's sensitivity curve on a second machine, HW2, can be represented as $y = a_{A2}x + b_{A2}$. From the set of benchmarks, each parameter is collected for each machine ($a_1 = \{a_{A1}, \ldots, a_{Z1}\}$, $b_1 = \{b_{A1}, \ldots, b_{Z1}\}$) and ($a_2 = \{a_{A2}, \ldots, a_{Z2}\}$, $b_2 = \{b_{A2}, \ldots, b_{Z2}\}$), resulting in two cross-architecture sensitivity parameter functions to be fit $a_2 = g_{a,1,2}(a_1)$ and $b_2 = g_{b,1,2}(b_1)$. For a linear model, the functions would be: $a_2 = \alpha_a a_1 + \beta_a$ and $b_2 = \alpha_b b_1 + \beta_b$, where α_a, α_b, β_a and β_b are architecture-dependent parameters.

3.2.2 Cross-architecture pressure models

The pressure, PS, of a program can be determined through the process described in Sec. 3.1. Through the data collection process, the pressure, $PS_{X\#}$ is determined for each benchmark X and architecture #. A cross-architecture pressure function is fit to describe the relationship between pressures on different architectures, for example two architectures HW1 and HW2: $PS_2 = h_{1,2}(PS_1)$.

3.3 Cross-architecture prediction

With the cross-architecture sensitivity and pressure models identified, they can be used to make predictions. Consider a new program Y run on HW1 whose sensitivity curve and pressure, PS_{Y1}, are found. The sensitivity function, $y = f_{Y1}(x)$, can be fit to the values from the sensitivity curve. Using the sensitivity function on HW1 and the cross-architecture sensitivity parameter models, $g_{p,1,2}$, the parameters of the sensitivity function can be predicted for HW2, $\hat{p}_{Y2} = g_{p,1,2}(p_{Y1}), \ldots, \hat{p}'_{Y2} = g_{p',1,2}(p'_{Y1})$. With the predicted parameters, the sensitivity function of Y for HW2 is predicted.

The cross-architecture pressure model is also used for prediction. Given the new program Y's pressure on HW1, PS_{Y1}, it's pressure on HW2 is predicted using the cross-architecture pressure model $\hat{PS}_{Y2} = h_{1,2}(PS_{Y2})$. The prediction methodology is shown in Figure 3.3.

This methodology allows the prediction of a program's sensitivity curve parameters and pressure for a new architecture. This information, using the bubble-up approach, can be used to predict co-run performance degradation with other programs. The proposed model can be used in a cloud data center to assist job scheduling and ensure SLA. We can bundle a *scale* that includes the bubble, the reporter and a script to collect the sensitivity curve and pressure of an user application. The *scale* can run either in-house or a data center benchmark machine. In a sense, the *scale* pre-measures a user application's performance-centric resource demand. The data center can build a database of predictive functions across all types of its machines. With the scale measurement and the database,

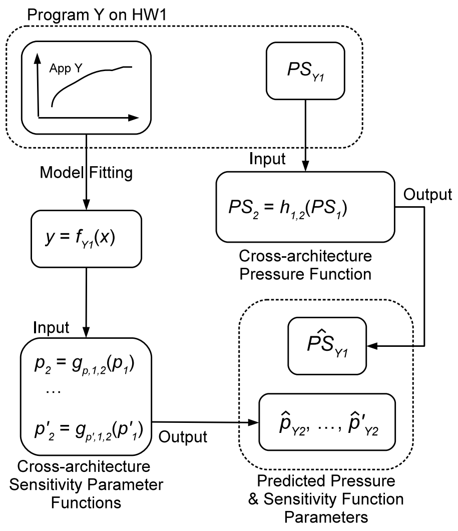
Program Y on HW1
App Y
PS_{Y1}
Model Fitting
Input
$PS_2 = h_{1,2}(PS_1)$
Output
Cross-architecture Pressure Function
$y = f_{Y1}(x)$
Input
$\hat{PS}_{Y1}$
$p_2 = g_{p,1,2}(p_1)$
…
$p'_2 = g_{p',1,2}(p'_1)$
Output
$\hat{p}_{Y2}, \ldots, \hat{p}'_{Y2}$
Cross-architecture Sensitivity Parameter Functions
Predicted Pressure & Sensitivity Function Parameters

Figure 3.3: Use of cross-architecture prediction models

the data center scheduler can predict any application's co-run performance on any machine and thus accurately schedule jobs to maximize overall system throughput or to guarantee one application's minimum performance.

One significant advantage of cloud computing is that it provides computing as a utility. However, the metric of computing utility is through resource allocation not through actual performance delivery about which the end users care. The major reasons include disparity of heterogeneous hardware and multi-tenancy of multiple servers that compete for resources. For example, AWS often sells a high-end server as a number of Elastic Computer Units (ECUs) where a user can subscribe a portion of them that lead to co-tenancy. It has been reported the performance of VM of x ECUs is not x times of that of the one-ECU benchmark machine [47]. The model proposed in this paper can help the cloud provider to accurately predict a user application's slowdown when it shares a physical machine with another application. The prediction thus helps estimate the computing power of a x-ECU VM when the rest of the machine is used for another co-run VMs. Thus, the SLA can be contracted based on performance not just resource allocation.

3.4 Evaluation settings and experimental result

Table 3.1: Hardware configurations of experimental machines

Hardware Configuration	**Intel Processors**				**AMD Processors**	
	Core2 Duo	**i5**	**i7**	**Xeon**	**A8**	**A10**
Processor Num.	E8200	760	920	E5345	3850	7850K
CPU clock (GHz)	2.66	2.8	2.67	2.33	2.90	3.70
Cores per CPU	2	4	4	8	4	8
LLC size (MB)	6	8	8	4	1	2

3.4.1 Hardware architectures

We have several machines with different hardware configurations used in the evaluation; see Table 3.1. Intel i7's hyper-threading is disabled to avoid intra-core contention. To emulate thermal control in data center, we also run the Intel i7 at a lower clock rate of 1.6GHz. The Intel Xeon CPU has two sockets, each with 4 cores. In each socket, every two cores share a 4MB last level cache. AMD A8 has 1MB private last level cache and AMD A10 has 2MB shared last level cache. In general, we use *Core2 Duo* as the base machine, HW1, and predict sensitivity and pressure for other machines, HW2.

3.4.2 Training and test benchmarks

In order to perform predictions, we select a set of benchmark programs including SPEC CPU2006 [45], a subset of PARSEC 3.0 [9], as well as a subset of CloudSuite 2.0 [42]. We use *SPEC CPU2006 as the training set* and test the prediction models using PARSEC 3.0 and CloudSuite 2.0.

We run each of the programs with the bubble at various levels. The bubble expands from 0M to 10MB with an interval of 1MB. Therefore, the bubble can stress the cache resource from essentially no pressure to fully occupying the cache and eventually competing for the memory bandwidth. While collecting the program performance degradation along with bubble size, the Intel performance counters are used to monitor program behavior. Specifically, the number of the last level cache misses per kilo instructions (LLCMPKI) and the number of instructions retired are collected, at a two-second interval, to help us understand the relationship between performance degradation and cache contentions.

3.4.3 Sensitivity curves and regression models

We profile each training program and record its execution times along with the different bubble pressures. The run times are normalized using the execution time of the program at zero co-run pressure and combined with bubble pressures to form the program's original sensitivity curve. In order to find a relationship between bubble pressure and performance, we fit the original curve with a continuous function.

Different regression models are tested to model the sensitivity curves and fulfill the cross-architecture prediction tasks. In our experiment, three functions have been used as regression models for the sensitivity curves: a linear model, a degree 2 polynomial model (d2poly), and a logistic model with 3 parameters (logistic3). The corresponding formulas are as follows,

$$y = ax + b, \tag{linear}$$

$$y = ax^2 + bx + c, \tag{d2poly}$$

$$y = c/(1 + e^{-b(x-a)}), \tag{logistic3}$$

where x is the bubble pressure score and y is the normalized performance degradation. For the logistic function, c is the maximum asymptote, b is the slope which describe the steepness of the curve, and a is the inflection point, where the curve changes directions.

These functions were considered because each model is simple and can be expressed using a small number of parameters. For a program that requires little cache, the performance degradation is not

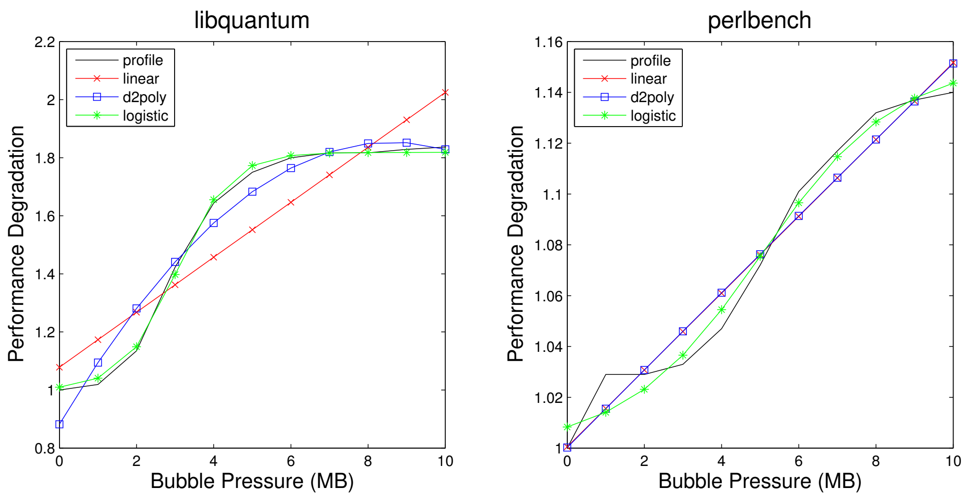

Figure 3.4: Sensitivity curves for *libquantum* and *perlbench* fitted to regression models

significant so that a linear function can characterize the sensitivity curve. For a program that is very cache sensitive, its performance degradation rises rapidly as the cache contention increases, and become rather flat when the pressure level is larger than the last level cache size, therefore a degree 2 polynomial can characterize such scenario. Finally, the programs in between previous two scenarios may have a logistic-like sensitivity curve. In Figure 3.4, the sensitivity curve and models for *libquantum* are shown on the left; from the plot the logistic3 function fits the data the best. For *perlbench*, although the logistic3 function still shows the best fit, a linear function may suffice in describing the profiled sensitivity curve as shown on the right.

3.4.4 Model selection criteria

The R-square value, root-mean squared error (RMSE), and Akaike information criterion (AIC) were calculated for selecting the best regression model. In statistics, the R-square value, which is also called the coefficient of determination, denoted R^2, is calculated to indicate how well the data fits to a model. Suppose a data set has n points, each data has value y_i, and each data also has an associated predicted value $\hat{y}_i$. R^2 is calculate as,

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}, \tag{3.1}$$

where

$$\bar{y} = \tfrac{1}{n}\textstyle\sum_{i=1}^{n} y_i,$$

$$SS_{tot} = \textstyle\sum_{i=1}^{n}(y_i - \bar{y})^2,$$

$$\text{and } SS_{res} = \textstyle\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

RMSE is a measurement of the difference between the value predicted by a model, $\hat{y_i}$, and the actual observed value, y_i, as,

$$RMSE = \sqrt[2]{\frac{\sum\limits_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}. \tag{3.2}$$

AIC, specifically the second-order corrected AIC_C, uses K as the number of parameters for the model plus 1 [17]. The AIC_C is then found as,

$$\text{AIC}_C = n \ln \frac{SS}{n} + 2K + \frac{2K(K+1)}{n-K-1}. \tag{3.3}$$

To compare and select the best model, choose the model with the minimum AIC_C value.

Note, the linear function has 2 parameters and the d2poly and logistic3 function have 3 parameters. It is expected that the more parameters used, the more accurate the function fits the data. A F-test can be used to determine whether the linear or degree 2 polynomial functions better fits the data (taking into account the number of parameters). However, the F-test is not used to select between models that are not nested (the linear is nested with the degree 2 polynomial function; however, the polynomial functions do not nest with the logistic function). Therefore, AIC is used for model selection between all models.

Table 3.2: R^2 for SPEC06int models on Core2 Duo and Intel i5

	Core2 Duo			**Intel i5**		
Program	**linear**	**d2poly**	**logistic3**	**linear**	**d2poly**	**logistic3**
astar	90.54	96.16	**99.83**	94.10	**99.58**	98.46
bzip2	93.80	95.17	**99.90**	96.99	97.02	**99.52**
gcc	90.63	96.17	**99.88**	96.46	**99.49**	98.00
gobmk	96.60	97.03	**99.47**	98.73	**98.77**	98.16
hmmer	87.06	91.64	**98.90**	87.03	87.21	**98.10**
h264ref	92.78	92.98	**98.54**	94.48	97.28	**98.86**
libqauntum	83.22	95.62	**99.78**	74.16	96.36	**99.54**
mcf	82.85	98.06	**99.86**	84.51	**99.34**	98.94
omnetpp	88.10	97.08	**99.91**	90.08	97.73	**99.74**
perlbench	96.12	96.13	**98.28**	96.72	**99.61**	98.05
sjeng	94.05	94.38	**98.35**	90.77	97.04	**98.91**
xalancbmk	90.82	96.14	**99.87**	89.80	98.39	**99.45**

3.4.5 Selection of sensitivity function model

First, we profile the training benchmarks (SPEC CPU2006) on both Core2 Duo and Intel i5 machines independently. The sensitivity curves are fit to the three models described in Section 3.4.3. We only discuss the results for SPEC integer programs (SPEC INT). The results for SPEC floating point programs (SPEC FP) are similar, and lead to the same conclusions.

From Table 3.2, we observe that the R-square values are always higher for the logistic function model on the Core2 Duo. For the Intel i5, the logistic function is highest in a majority of benchmarks; when the degree 2 polynomial has the highest R-squared value the logistic model often delivers a quite similar value. Similarly, the RMSE values are smaller for the logistic model on the Core2 Duo and a majority of programs on the Intel i5; see Table 3.3. In Table 3.4, AIC_C values are reported. Note, that the lower AIC value indicates the preferred model to be selected. For these benchmarks, the criteria reveal the logistic model best fits the data for almost all programs. The few exceptions are that of *astar*, *gcc*, *gobmk*, *mcf* and *perlbench* on the i5 architecture. We thus pick the logistic3 function to model sensitivity curves.

Table 3.3: RMSE value for SPEC06int models on Core2 Duo and Intel i5

	Core2 Duo			**Intel i5**		
Program	**linear**	**d2poly**	**logistic3**	**linear**	**d2poly**	**logistic3**
astar	0.040	0.029	**0.005**	0.006	**0.002**	0.003
bzip2	0.036	0.031	**0.004**	0.005	0.006	**0.002**
gcc	0.060	0.043	**0.007**	0.006	**0.002**	0.004
gobmk	0.008	0.008	**0.003**	0.001	0.001	**0.001**
hmmer	0.003	0.002	**0.001**	**0.001**	**0.001**	**0.001**
h264ref	0.013	0.013	**0.006**	0.002	0.002	**0.001**
libquantum	0.135	0.071	**0.015**	0.028	0.011	**0.004**
mcf	0.099	0.033	**0.009**	0.022	0.005	**0.005**
omnetpp	0.098	0.049	**0.008**	0.020	0.010	**0.003**
perlbench	0.010	0.010	**0.007**	0.004	**0.002**	0.003
sjeng	0.012	0.012	**0.006**	0.005	0.003	**0.002**
xalancbmk	0.075	0.054	**0.009**	0.021	0.009	**0.005**

3.4.6 Cross-architecture predictions for sensitivity curves

Since we selected the logistic3 function for modeling the sensitivity curves, we need to predict the three parameters in the logistic3 function for cross-architecture sensitivity curve prediction. The cross-architecture function, $p_{i5} = g_{p,c2duo,i5}(p_{c2duo})$, is fit to a degree 2 polynomial function, which outperforms other functions we have tested. We follow the design described in Sec. 3.2.2 using the parameters from HW1, Core2 Duo, to predict the parameters of the sensitivity function for the i5 machine, HW2. Using these predicted parameters, the performance degradation is predicted for each benchmark on the i5 machine. The predicted performance degradation, $\hat{y}_{i5}$ is compared to the known profiling value, y_{i5} via the absolute and relatives errors to show prediction accuracy (note, the absolute error is reported as a percentage, value*100). The errors are calculated for each bubble input values of 0 to 10MB. The mean absolute and relative errors are reported for each benchmark.

In Table 3.5, the training model can predict the sensitivity curve for program running on Intel i5 with an overall average error lower than 2% across all discrete bubble pressures. The worst case is *cactusADM* for which the error is still within 5%. The cross-architecture sensitivity parameter function is then tested on new benchmarks, those not used to train the function. We test the cross-

Table 3.4: AIC_C value for SPEC06int models on Core2 Duo and Intel i5

	Core2 Duo			**Intel i5**		
Program	**linear**	**d2poly**	**logistic3**	**linear**	**d2poly**	**logistic3**
astar	-61.2	-63.3	**-100.7**	-102.0	**-126.0**	-112.2
bzip2	-64.0	-61.6	**-104.7**	-108.5	-103.4	**-123.8**
gcc	-52.6	-54.7	**-95.7**	-103.5	**-119.5**	-105.3
gobmk	-95.4	-91.6	**-111.1**	**-137.8**	-132.9	-128.9
hmmer	-122.3	-122.1	**-143.5**	-141.0	-135.9	**-156.8**
h264ref	-85.5	-80.5	**-97.1**	-124.8	-127.5	**-137.3**
libquantum	-34.6	-43.7	**-77.3**	-68.9	-85.4	**-108.3**
mcf	-41.5	-62.2	**-89.4**	-74.6	**-103.9**	-99.2
omnetpp	-41.8	-51.8	**-91.2**	-76.6	-85.7	**-112.2**
perlbench	-92.7	-87.5	**-96.2**	-110.4	**-128.7**	-111.8
sjeng	-88.1	-81.9	**-96.5**	-109.3	-116.8	**-128.2**
xalancbmk	-47.6	-49.4	**-89.5**	-75.1	-88.5	**-102.6**

architecture prediction for a subset of the PARSEC and CloudSuite programs; results are given in Table 3.6. The errors on the test benchmarks suggest the cross-architecture prediction model delivers high accuracy with a maximum relative error $< 2\%$.

We observe similar accuracy from Table 3.7 for predictions from Core2 Duo to other architectures, Xeon, Intel i7, AMD A8, and AMD A10. The mean relative error of the cross architecture bubble versus performance degradation from Core 2 Duo to Xeon are 3.75% and 3.45% for SPEC INT and FP, respectively. The mean relative error for the other architectures are all within 3.8%. For almost every pair of cross-architecture sensitivity prediction, we observe that the prediction accuracy is always low for a certain set of benchmark programs, such as bzip2, libquantum, and lbm. Such observation suggests that those programs be treated as a different group for cross-architecture prediction to improve accuracy.

3.4.7 Cross-architecture predictions for program pressure

As described in Section 3.1, using the reporter and its sensitivity curve, a pressure score can be determined for every benchmark when the benchmark co-runs with the reporter. In Table 3.8, the benchmarks' pressure is reported on multiple machine architectures. For the Intel i7 machine, the

Table 3.5: Mean errors of cross-architecture sensitivity curve prediction for Intel i5 on SPEC06

Program	**absolute error (%)**	**relative error (%)**
astar	2.04	1.94
bzip2	2.24	2.13
gcc	3.51	3.30
gobmk	0.73	0.72
hmmer	0.21	0.21
h264ref	1.01	0.99
libquantum	2.04	1.83
mcf	2.04	1.84
omnetpp	1.98	1.81
perlbench	2.26	2.16
sjeng	0.58	0.57
xalancbmk	3.16	2.82
bwaves	0.45	0.44
milc	3.09	2.90
zeusmp	1.40	1.36
gromacs	0.22	0.22
leslie3d	3.35	3.17
namd	0.15	0.15
dealII	1.42	1.38
soplex	1.64	1.50
povray	0.29	0.29
GemsFDTD	0.53	0.50
tonto	1.29	1.26
lbm	2.61	2.31
sphinx3	3.96	3.53
gamess	0.17	0.17
calculix	0.78	0.75
cactusADM	4.84	4.46

clock is set to two values: 2.6GHz and 1.6GHz. Note, how the pressure can change with different hardware configurations. Therefore, we need to predict the pressure of programs across architecture.

The cross-architecture pressure function can be fit using the pressures of programs running on Core2 Duo to predict the pressure of programs running on other machines, such as an i5 or i7. The cross-architecture functions considered are linear, degree 2 polynomial, and degree 3 polynomial functions.

We look at the prediction performance for the cross-architecture function of $\hat{PS}_{i7} = h_{c2duo,i7}(PS_{c2duo})$, compared to the actual pressure as measured by the reporter on i7. The mean RMSE value of the

Table 3.6: Mean errors of cross-architecture sensitivity curve prediction for Intel i5 on test benchmarks

Program	**absolute error (%)**	**relative error (%)**
bodytrack	0.19	0.19
swaptions	0.12	0.12
ferret	0.46	0.45
fluidanimate	0.89	0.87
freqmine	0.62	0.62
streamcluster	0.53	0.51
x264	0.56	0.54
graphic analysis	2.05	1.96
software testing	1.38	1.31
data caching	0.90	0.87

pressure, mean relative error of the pressure, and mean absolute error are calculated, where the average is over all of the training benchmarks. The linear model's RMSE is 0.526, relative error is 22.5%, and absolute error is 38.7%. The degree 2 polynomial model performance is 0.4461, 36.6%, 33.5% and the degree 3 polynomial model performance is 0.391, 19.1%, 30.0% for the RMSE, relative and absolute errors, respectively. Though relative error is around 20%, the absolute error is less than 0.4 compared to the range of pressures from 0 to 8MB (the maximum cache size of the given architectures). The Xeon predicted pressure from the Core2 Duo pressure results in a relative error of ∼15% and an absolute error of less than 0.25.

We see a program exhibits different pressures on different machines. Let the pressure a program gives on Core2 Duo be a baseline reference. The same program gives ∼1MB less pressure for most cases on Intel Xeon machine, suggesting there might exist a relationship between pressure change and different cache sizes in the two machines. On the other hand, the pressure fluctuates on Intel i5 machines. For some of the programs, the pressure decreases and the rest of the programs had increases in pressure. The watershed for such discrepancy is close to 3MB, which is half the cache size of Core2 Duo. This may relate to the fair use of cache and further research may be needed.

3.4.8 Cross-architecture co-tenancy performance prediction

In order to test the correctness of our cross-architecture prediction models, we run the benchmark programs pair-wise to observe the co-run performance of real programs. We consider several methods to predict the co-run performance. First, we use the sensitivity curve created by the profiling result and the pressure generated by the reporter to predict the performance degradation (the bubble-up methodology or Bubble). In PredPres, we use the sensitivity curve created by the profiling result and the pressure predicted by the cross-architecture model to predict the performance degradation. Finally, we use both the sensitivity curve and pressure predicted by cross-architecture models to predict the performance degradation, PredSens+PS. The three methods are evaluated as the mean absolute and mean relative error comparing each predicted performance degradation with the actual performance degradation averaged over the pair-wise co-run benchmarks.

Table 3.9 shows the average prediction errors when each individual benchmark co-runs with each SPEC program. Bubble's largest relative error is 1.94% on the training benchmarks which is consistent with the conclusion of the original results by Mars et al. [70]. The largest relative error for the test benchmarks is 2.26%. For PredPS, the relative error is always of equal or greater value than Bubble, the baseline approach. However, the maximum relative error of 2.57% is of similar magnitude. Lastly, PredSens+PS's maximum relative error is merely 5.30%, although a few see a substantial increase in error over the baseline Bubble approach, e.g., *libquantum*, *mcf*, *omnetpp*, *xalancbmk*.

In Table 3.10, the detailed prediction error (presented as a percentage, value*100) is given for PredSens+PS on the pair-wise co-run benchmarks. Overall, the prediction error is relatively small for most SPEC integer programs. However, the prediction error is high whenever a program co-runs with *libquantum*. One possible reason for this may be in the design of the bubble program. We need to look up the reporter's sensitivity curve to generate a pressure score comparable with a certain bubble size for a given program. However, the sensitivity curve doesn't change much beyond the point where bubble size equals the last level cache size. This may not accurately characterize the actual pressure level of *libquantum* which also stresses the memory bandwidth. As a result, the reporter will give a small value for *libquantum* than its actual pressure level.

Table 3.11 shows the prediction accuracy when SPEC FP is chosen as the co-run programs. Most predictions are close to actual performance degradation within 2% error, while when *libquantum* co-run with *bwaves*, the predict value is 12% away from real value. This is because the predicted *bwaves* pressure is 5.6MB and the actual *bwaves* pressure is 3.1MB. Moreover, the sensitivity curve of *libquantum* changes rapidly along with bubble pressure, thus the predicted value is very different from the actual value.

The "bubble-up" methodology uses 2 ruler to quantify the performance of two co-run programs. For most cases, the predicted degradation are as close as the actual co-run result. Our framework extent the idea to a cross-architecture settings, using simple machine learning techniques to transfer knowledge from a source machine to multiple target machine with different hardware configurations. Yet the framework uses the same ruler as a quantitative measure, thus to make accurate prediction, one must make sure that these 2 measurements are predicted accurately, moreover, this requirement also implies that the profiled sensitivity should be also accurate in the first place or otherwise the error can be propagated through logistic regression and several following prediction step. In the next section, we discuss how to elevate the prediction accuracy by categorize programs into different access behavior groups and how to generate program specific bubble/reporter to make the ruler even more accurate.

Table 3.7: Mean relative errors of cross-architecture sensitivity curve prediction for Intel Xeon, i7, AMD A8 and A10 on SPEC06

Program	**xeon error (%)**	**i7 error (%)**	**A8 error (%)**	**A10 error (%)**
astar	1.46	2.60	2.46	2.07
bzip2	17.80	3.06	2.14	13.62
gcc	4.83	3.43	0.52	1.35
gobmk	0.42	1.00	0.77	0.55
hmmer	0.63	0.24	0.62	1.46
h264ref	2.72	2.12	0.89	1.12
libquantum	4.25	5.93	5.66	5.47
mcf	3.26	2.75	2.59	2.69
omnetpp	1.56	4.06	3.06	6.57
perlbench	2.05	1.32	1.20	2.52
sjeng	3.64	2.52	1.31	5.11
xalancbmk	2.38	4.56	3.52	1.61
Average	3.75	2.80	2.06	3.68
bwaves	9.31	4.98	0.95	1.47
milc	5.79	3.05	3.24	4.29
zeusmp	4.00	0.72	1.12	1.09
gromacs	1.71	0.74	0.06	2.10
leslie3d	8.29	2.55	2.45	6.89
namd	0.24	0.10	0.15	0.93
dealII	1.12	0.80	2.12	1.85
soplex	3.63	6.29	0.31	6.10
povray	1.99	1.14	0.31	0.32
GemsFDTD	4.52	3.31	1.12	2.50
tonto	1.81	1.18	0.69	0.78
lbm	2.37	14.59	0.70	2.95
sphinx3	3.18	2.03	0.39	7.93
gamess	1.10	0.46	0.43	2.21
calculix	2.35	0.36	0.04	1.18
cactusADM	3.79	4.00	1.28	0.89
Average	3.45	2.89	0.96	2.72

Table 3.8: Benchmark programs pressures on different hardware configurations

						i7	
Program	**Core2**	**Xeon**	**i5**	**A8**	**A10**	2.6Ghz	1.6Ghz
astar	2.5	1.5	2.2	1.1	1.8	3.2	2.1
bzip2	2.3	1.3	1.8	1.2	0.8	2.6	2.0
gcc	2.5	1.5	3.4	1.5	0.7	3.7	3.0
gobmk	2.0	1.0	1.2	0.5	0.5	1.6	1.0
hmmer	2.2	1.2	1.4	1.0	0.8	1.1	1.0
h264ref	2.1	1.1	1.5	0.4	0.1	1.8	1.3
libquantum	3.9	4.0	5.1	8.0	8.0	8.0	8.0
mcf	3.2	2.1	8.0	1.2	0.9	4.3	4.7
omnetpp	2.8	1.8	4.6	1.4	1.2	3.4	3.6
perlbench	1.9	0.9	1.2	0.5	0.3	0.9	0.9
sjeng	1.6	0.6	0.9	0.2	0.4	0.8	0.6
xalancbmk	2.5	1.6	3.4	1.2	1.1	3.4	2.4
bwaves	3.2	2.4	3.1	1.2	1.8	4.1	3.4
milc	3.8	4.0	8.0	8.0	8.0	5.1	6.2
zeusmp	2.6	1.5	2.4	1.2	0.6	3.4	2.8
gromacs	1.5	0.6	0.7	0.3	0.2	0.9	0.7
leslie3d	3.7	3.0	8.0	4.0	3.1	5.2	5.6
namd	1.2	0.3	0.5	0.2	0.1	0.5	0.3
dealII	2.2	1.3	1.5	1.2	0.9	2.7	1.5
soplex	3.2	2.5	8.0	8.0	3.3	5.0	6.0
povray	1.2	0.4	0.5	0.2	0.1	0.3	0.5
lbm	3.5	1.5	8.0	8.0	8.0	8.0	8.0
tonto	2.2	1.3	1.4	1.1	0.6	2.5	1.8
sphinx3	2.8	2.5	3.2	1.9	1.1	3.7	3.8
cactusADM	2.3	1.2	2.1	1.0	1.0	2.9	1.9
calculix	1.4	0.4	0.4	0.2	0.3	0.6	0.5
gamess	1.4	0.4	0.4	0.2	0.2	0.5	0.5
GemsFDTD	4.8	4.0	8.0	3.0	8.0	5.6	5.3
blackscholes	1.3	0.3	0.5	0.4	0.4	0.7	0.7
bodytrack	1.3	0.5	0.5	0.4	0.4	0.6	0.7
ferret	2.3	1.2	0.5	0.9	0.8	2.1	1.4
fluidanimate	2.0	1.0	1.6	1.2	1.0	2.2	1.5
streamcluster	3.0	2.7	2.1	2.0	1.7	4.4	4.6
swaptions	1.0	0.6	0.1	0.2	0.2	0.1	0.3
x264	2.0	1.1	1.5	1.1	1.1	1.3	1.3

Table 3.9: Co-tenancy performance degradation prediction error on training (SPEC06int) and test benchmarks

	Bubble		PredPS		PredSens+PS	
Program	**absolute error (%)**	**relative error (%)**	**absolute error (%)**	**relative error (%)**	**absolute error (%)**	**relative error (%)**
astar	0.93	0.89	1.15	1.10	2.00	1.87
bzip2	1.87	1.69	2.19	1.97	2.25	2.03
gcc	1.20	1.07	1.65	1.48	2.37	2.14
gobmk	2.16	1.94	2.57	2.30	0.77	0.74
hmmer	0.54	0.52	0.69	0.67	0.70	0.66
h264ref	0.09	0.09	0.14	0.14	0.59	0.58
libquantum	1.22	1.18	1.43	1.38	3.55	3.19
mcf	1.98	1.80	2.27	2.09	5.43	4.78
omnetpp	0.01	0.50	0.67	0.65	4.96	4.25
perlbench	1.51	1.24	2.30	1.93	2.88	2.75
sjeng	0.76	0.71	1.13	1.06	1.07	1.02
xalancbmk	2.02	1.70	3.01	2.57	6.21	5.30
bodytrack	0.29	0.28	0.33	0.33	0.89	0.87
swaptions	0.12	0.12	0.12	0.12	1.17	1.16
ferret	0.49	0.47	0.66	0.64	1.00	0.98
fluidanimate	0.38	0.36	0.42	0.40	0.72	0.70
freqmine	0.69	0.67	0.70	0.67	0.85	0.83
streamcluster	0.96	0.92	1.14	1.09	2.07	1.99
x264	0.63	0.60	0.78	0.75	1.13	1.08
graphic analysis	1.20	1.17	1.06	1.05	1.76	1.68
data caching	2.44	2.26	2.64	2.46	2.85	2.67
software testing	1.10	1.01	1.24	1.14	2.02	1.95

Table 3.10: Detailed performance degradation prediction error using PredSens+PS against SPEC INT06

Program	**astar**	**bzip2**	**gcc**	**gobmk**	**hmmer**	**sjeng**	**libquantum**	**h264ref**	**omnetpp**	**perlbench**	**xalancbmk**
astar	2.24	0.91	0.05	0.54	1.54	0.12	4.92	0.48	1.04	0.09	0.78
bzip2	2.71	2.14	0.02	1.08	2.38	1.89	10.52	1.48	1.13	0.80	1.24
gcc	3.95	1.57	1.03	0.79	2.22	2.22	7.30	1.17	3.47	0.61	1.64
mcf	1.40	0.26	3.03	0.17	1.26	2.02	13.35	0.34	0.53	0.10	1.23
gobmk	0.57	0.10	0.48	0.08	0.40	0.07	2.93	0.12	0.12	0.17	0.23
libquantum	4.67	1.64	0.48	0.35	2.12	2.87	4.75	0.58	4.70	0.33	3.26
h264ref	0.92	0.67	0.37	0.44	0.86	0.71	2.17	0.47	0.35	0.23	0.31
sjeng	0.21	0.26	0.86	0.06	0.36	0.38	5.30	0.27	1.16	0.14	0.95
perlbench	1.49	2.12	2.56	1.81	1.51	1.65	6.38	1.33	3.49	1.89	2.71
omnetpp	5.08	0.64	2.03	0.65	1.56	2.45	15.88	0.33	0.46	0.35	0.27
xalancbmk	3.74	0.10	4.85	0.59	0.59	2.18	17.86	0.07	3.97	0.26	5.86
bodytrack	0.18	0.32	0.19	0.29	0.33	0.10	4.60	0.48	2.01	0.59	0.49
swaptions	0.48	0.02	0.31	0.09	0.04	0.00	8.25	0.09	3.23	0.00	0.31
ferret	0.89	0.50	1.77	0.20	0.10	0.30	2.14	0.30	2.77	0.30	1.48
fluidanimate	0.30	0.20	0.50	0.20	0.20	0.10	4.11	0.30	0.99	0.20	0.60
freqmine	0.30	0.20	0.30	0.70	0.50	0.20	5.30	0.60	0.30	0.40	0.30
streamcluster	1.68	1.01	3.55	1.85	2.19	0.89	1.57	1.46	2.51	0.79	4.37
x264	1.18	0.88	1.58	1.28	1.13	0.49	0.98	0.73	1.01	0.24	2.38
graphic analysis	0.99	0.63	1.95	0.02	0.00	0.14	7.42	0.00	3.00	1.47	2.88
data caching	0.95	3.22	4.18	0.32	0.69	2.73	8.49	1.67	4.01	0.37	2.75
software testing	1.75	1.22	2.97	0.52	2.45	0.35	2.15	1.57	4.04	1.22	3.15

Table 3.11: Detailed performance degradation prediction error using PredSens+PS against SPEC FP06

Program	**bwaves**	**milc**	**zeusmp**	**leslie**	**namd**	**povray**	**sphinx**	**cactus**	**calculix**	**gamess**	**GemsFDTD**
astar	4.47	0.94	1.02	1.11	0.39	0.84	0.33	0.02	0.50	0.23	0.44
bzip2	1.49	1.34	2.29	0.56	1.21	1.16	2.16	1.15	0.35	0.56	1.49
gcc	8.55	1.24	2.46	3.36	1.27	1.44	2.68	1.43	0.25	0.57	0.53
mcf	5.68	1.23	0.67	1.46	1.24	1.16	2.47	0.69	0.43	0.40	3.34
gobmk	1.81	1.12	0.14	0.12	0.18	0.34	0.07	0.50	0.03	0.01	0.59
libquantum	12.10	3.47	3.46	6.71	1.95	1.23	2.76	2.61	0.04	0.13	1.92
h264ref	1.02	0.00	0.61	0.50	0.35	0.21	0.74	0.46	0.27	0.03	0.12
sjeng	2.10	2.51	0.56	1.74	0.32	0.30	0.38	0.73	0.02	0.15	2.67
perlbench	1.61	3.91	3.47	3.24	0.33	0.11	2.86	2.20	0.75	0.51	4.36
omnetpp	3.50	3.22	1.09	1.10	1.88	1.78	0.90	2.25	0.07	0.32	5.08
xalancbmk	4.11	5.19	0.02	5.19	1.43	1.70	6.39	2.30	0.18	0.28	6.24

Chapter 4

Cross-architecture performance modeling with clustering

4.1 Reasons for prediction errors and how we tackle the problem with a K-means clustering algorithm

In our proposed framework, a profiled sensitivity curve is first fit to a logistic function for both source and target hardware configurations, whose parameters are modeled using a degree-2 polynomial function for cross-architecture sensitivity prediction. The assumption behind this is that all programs follow exactly the same transition pattern from a source machine to a target machine. However, due to different access patterns and memory bandwidth consumption behaviors, this assumption may not always hold. We select a few profiled sensitivity curves to illustrate such phenomenon.

Figure 4.1 shows the profiled sensitivity curves of `astar`, `bzip2`, `perlbench` and `sjeng` on both Intel core2duo and i5 machines with solid and dash line respectively. The sensitivity curves become flat when the bubble size is larger than 8MB as the the program saturated the entire last level cache. By comparing `astar` with `bzip2`, we find both programs have approximately the same slowdown at 10MB and the overall sensitivity curve shapes are similar. A cross architecture model can be applied to both programs as they basically have roughly the same transition behavior and we can expect that the model will generate accurate prediction.

However, as shown in Figure 4.1, at 10MB, the gap for program `perlbench` and `sjeng` are much narrower than that for `astar` or `bzip2`. If we were to use the cross architecture model

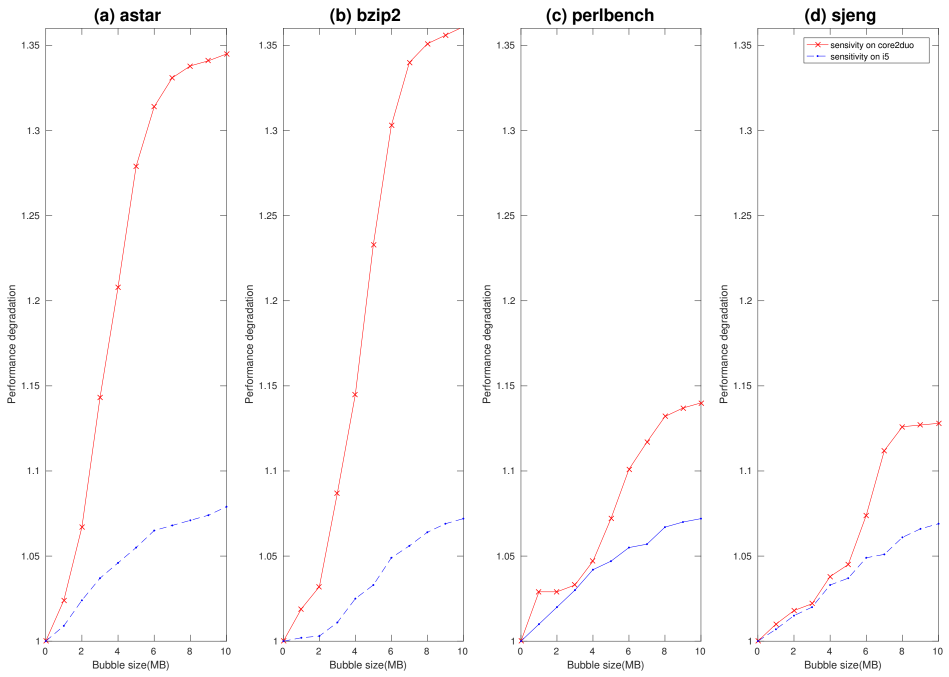

Figure 4.1: Cross architecture sensitivity curve for (a) `astar`, (b) `bzip2`, (c) `perlbench`,(d) `sjeng` on core2duo and i5

obtained from the first two programs and try to make prediction, `astar` and `bzip2` might have reasonable prediction, while the slowdown of `perlbench` can be overestimated and thus it will lose some potential co-run opportunities. Similarly, underestimation can also happen vice versa and thus bring unacceptable slowdown which can violate the QoS requirement. This observation suggests that programs should be put into correct categories while making cross-architecture predictions. We therefore introduce clustering as a preliminary preparation for cross-architecture regression modeling. Notice that this additional step wouldn't violate the $O(n)$ time complexity since eventually a certain quantity of benchmark programs need to be collected as the training set, and clustering and cross-architecture modeling within these sampling is a one time off-line process. Instead of maintaining just one single cross-architecture prediction model for sensitivity, we keep k models, where k equals the number of clusters we have. Whenever a new program comes in, profiling is still needed on the source machine. In addition, we use this information to compare with different clusters and put it into correct class and use the corresponding model to predict sensitivity on the target machine. We adopt a k-means algorithm to form clusters. As all sensitivity curves on the source machine (core2duo) show how each benchmark reacts to different bubble pressures, it preserves certain information about what transition pattern type each program will be on the target machine. Therefore, the curves of those benchmark programs are served as the input of the k-means algorithm and we test various k values for the best result. Intuitively, the more models we keep, the more accurate prediction result would be. However, due to the limit number of training programs, we test k from 2 to 5. Silhouette criterion values [86] and Calinski-Harabasz criterion [19] (CH index) are used to find the optimal k values. The silhouette of a data instance is a measure of how closely it is matched to data within its cluster and how loosely it is matched to data of the neighboring clusters. And CH index evaluates the cluster validity based on the average between- and within-cluster sum of squares. CH index is calculated as follows:

$$CH(k) = \frac{\frac{TraceB}{K-1}}{\frac{TraceW}{N-K}} \tag{4.1}$$

$$TraceB = \sum_{k=1}^{K} |C_k| ||\bar{C_k} - \bar{x}||^2 \tag{4.2}$$

$$TraceW = \sum_{k=1}^{K} \sum_{i=1}^{N} w_{k,i} ||x_i - \bar{C_k}||^2 \tag{4.3}$$

$$|C_k| = \sum_{i=1}^{N} w_{k,i} \tag{4.4}$$

where N is the number of observations, and K is the number of clusters. The value $w_{k,i}$ equals 1 if observation x_i belongs to cluster k and equals 0 otherwise. The value B denotes the error sum of squares between different clusters (inter-cluster), and W the squared differences of all objects in a cluster from their respective cluster center (intra-cluster). After finding the optimal k value, all the training benchmark programs are then categorized into the corresponding cluster.

We apply the framework shown in Figure 3.2 to each cluster independently. Experiment results for cross-architecture sensitivity prediction with clustering will be presented in Section 4.2.

Both sensitivity curve and program pressure prediction error can result in performance degradation prediction errors. As different programs react differently towards pressure, some of programs' sensitivity curve will be steeper compared to others. Therefore, even though a small program pressure prediction error will bring large performance degradation prediction error. Table 4.1 illustrates this interesting discovery. It shows several programs' partial sensitivity profiling results and the actual performance degradation when it co-runs with `lbm`. The `lbm`'s pressure is larger than 5MB bubble kernel when it co-runs with `perlbench`. However, when `gcc` running as the peer, `lbm`'s pressure is in-between 4MB to 5MB. And at last, when 2 `lbm` co-run together, the pressure even drops below 4MB, suggesting that a program's pressure can change whenever its co-running peer changes. We can explain this with some domain knowledge. `Perlbench`, `gcc` and `lbm` are listed in ascending order with respect to its own pressure, therefore, as the co-runner, `lbm`, which working as an elastic rubber band, its own pressure will change from a higher value to a lower score. To be specific, when `lbm` co-run with `perlbench`, `lbm`'s pressure exceeds a 5MB bubble. `gcc`'s pressure is higher than `perlbench`, so when it co-run with `lbm`, `lbm`'s pressure score should be close to 4MB bubble. For the extreme case, when two `lbm` co-run together, the pressure is way below 4MB. As a program's pressure will change according to its competing peer, if we were to predict the pressure with a unique reporter, it is clear that the prediction will more or less introduces error depending on the steepness of sensitivity and the pressure score itself.

Table 4.1: lbm pressure changes with different peer co-runner

Program	Program execution time(s)		
Name	@pressure 4MB	@pressure 5MB	co-run with `lbm`
perlbench	484	488	500
gcc	470	485	471
lbm	557	666	520

Similarly to maintain a certain number of cross architecture sensitivity models, we should have multiple reporters to represent programs with aggressive contention power or mild ones rather than using a single reporter and create cross-architecture models for different reporters. Whenever a new program comes in, we first profile its sensitivity on source machine and put it into proper cluster based on this information. Then the reporter corresponding to such cluster is used to measure the pressure score of the program on source machine. At last we apply the cross architecture model to predict the sensitivity and pressure on target machine to make final slowdown prediction. The training and testing steps for this cluster-based cross architecture model is summarized as a pseudo-code style in Algorithms 1 and 2. Experiment results for cross-architecture performance degradation prediction with clustering will be shown in Section 4.2.

First we select a variety of programs as the training set. Then we collect the sensitivity of these programs on both source machine and target machines. With k-means algorithm, we categorize programs into different groups based on their sensitivity. Then for each cluster, extract program sensitivity parameters using logistic function regression for both source and target, at last, create the cross architecture sensitivity model using quadratic model regression from source to target. Similar steps can be applied to pressure prediction model. As we need different reporter to represent different peer pressure, we collect the sensitivity of three different reporter kernel, which are a binary search tree kernel, a stream access kernel and a matrix access kernel. Calculate the distance between a reporter and different cluster centroids to determine which group it belongs to. Collect the pressure score for each program on both source and target machine and then create the cross-architecture pressure model for the cluster this very reporter corresponds using quadratic model regression.

Predicting performance using the cluster-specific model is straightforward. For each new programs, first collect the sensitivity on source machine and extract the logistic parameter using regression. Then, depending on the pressure type of the peer this program co-runs with, we choose the corresponding reporter and collect program's pressure on source machine. Then, calculate the sensitivity distance between the program with our database and pick the cross-architecture sensitivity/pressure model with shortest distance. At last, generate the sensitivity and pressure on target machine to make final prediction.

Algorithm 1: Training for cluster-based cross-architecture prediction model

```
Input  : A set of programs and machines with different configurations served as source and
         target machines
Output: Cross-architecture sensitivity/pressure models
step 1: Select a variety of programs as training set;
step 2: Sensitivity profiles on source/target machines;
step 3: Clustering based on sensitivity with k-means;
step 4: Create cross-architecture models
for each cluster
a. sensitivity model
From profiled sensitivity on source/target to logistic parameters;
Train a degree 2 polynomial model from source to target with the logistic parameters.
b. pressure model
Find a reporter for the cluster by measure the distance between reporter's own sensitivity and
 cluster centroid;
Measure training programs' pressure on source/target machine with the reporter determined
 in previous step;
Train a degree 2 polynomial model from source to target with the pressure scores.
```

Algorithm 2: Testing/Using cluster-based cross-architecture prediction model

```
Input   : New program A and B
Output: Slowdown for A when it co-runs with B on target machine
step 1: Sensitivity prediction for A;
a. Profile A' sensitivity on source machine
b. Find cluster whose centroid is closest to A
c. Predict A's sensitivity using the model corresponding to cluster determined in b
step 2: Pressure prediction for B;
a. Profile B's pressure using the corresponding reporter associated with cluster for A on
  source machine
b. Predict pressure for B on target machine using model associated with cluster for A
step 3: Co-run slowdown prediction for A;
```

4.2 Evaluation settings and experimental results

Table 4.6 shows the cross-architecture sensitivity curve prediction from source machine, core2duo, to five target machines with different hardware configurations. Even though the average error are all below 3.8%, certain program gets as high as over 10% prediction error. Therefore, we divide programs into different clusters as discussed in previous section and make the predictions independently for each cluster.

Table 4.2: Sensitivity prediction for different machines

	prediction errors %	
Program name	int	fp
INTEL i5	1.69	1.52
INTEL i7	2.80	2.89
INTEL Xeon	3.75	3.45
AMD A8	2.06	0.96
AMD A10	3.68	2.72

As we separate SPEC CPU2006 integer and floating point programs into training set and testing set, respectively, the scaled sensitivity of Integer programs on Core2Duo are fed into the k-means algorithm for initial clustering. Both silhouette and CH index values show that three is the optimal cluster number for the k-means algorithm. Therefore, programs are clustered into three groups, from which three cross-architecture prediction models are generated. Given a new program (floating point programs in our example), we calculate the Euclidean distances between the program and three different clusters' centroids, and put it into the one with closest disparity. The clustering result is shown in Table 4.3.

Table 4.3: SPEC2006 clustering result using sensitivity as input

cluster	training	testing
1	astar,bzip,gcc	milc,GemsFDTD leslie3d,sphinx3
2	omnetpp,xalancbmk mcf,libquantum	bwaves,soplex lbm
3	gobmk,hmmer, h264ref,perlbench sjeng	zeusmp, gromaces,gamess dealII,povray,tonto,namd calculix,cactusADM

Figure 4.2 shows all three parameters of logistic sensitivity curve in different clusters. Programs `mcf`, `omnetpp`, `soplex`, `lbm`, are in the same cluster, which is consistent with the plot, as they all have small knee point and steep slope, as well as high upper bound, suggesting they are sensitive to pressure and will have large slowdown even with relatively small co-run pressure. Programs `sjeng`, `povray`, and `namd` are in same category for they all have low upper bound and large knee point, indicating those programs are insensitive to pressure changes. This classification is reasonable as

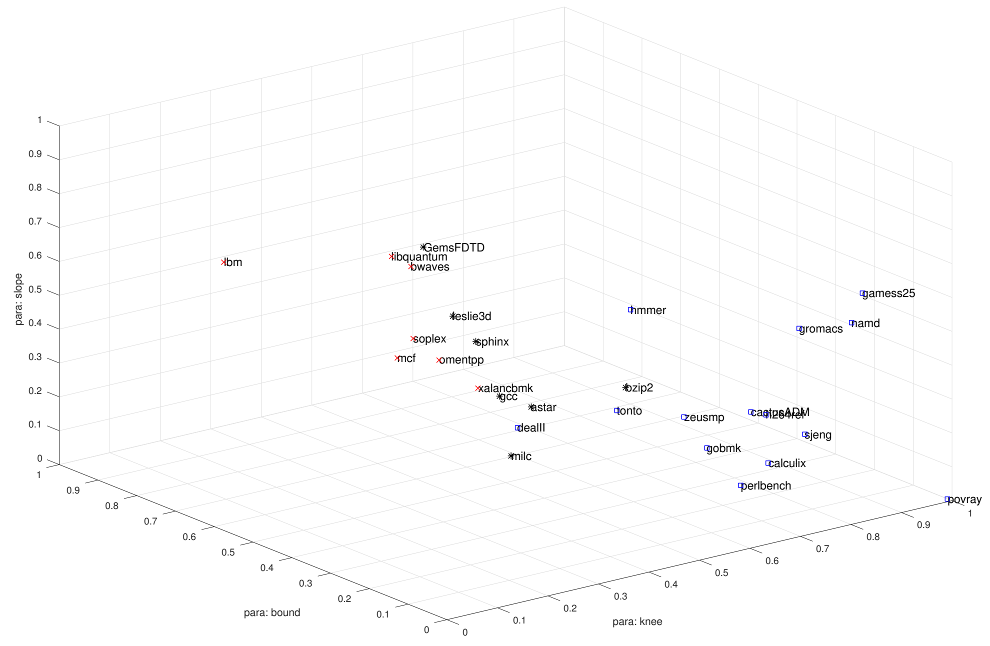

Figure 4.2: logistic parameters distribution

previously shown in Figure 4.1, **`astar`** and **`bzip2`** are very similar in terms of cross architecture transition pattern and they belong to the same category. This is also true for other programs, especially for those programs which are sensitive towards pressure changes in cluster two, which we might need a much more accurate model compared with the ones who has negligible pressure as programs in cluster three. The sensitivities of those programs within cluster two change significantly at each granularity unit, so the prediction must be accurate so that the QoS requirement will not be violated.

The detailed program sensitivity prediction on target machines using the corresponding model and the comparison with actual profiled curves are shown in Table 4.4. As we have five groups of cross-architecture configurations, for simplicity, the rest are given average prediction error in Table 4.6. We also show the result for $k = 1$, which treats all programs as a whole, meaning no clustering is applied to the programs and only one model was generated.

Table 4.4: Sensitivity prediction from c2 to i5 using cluster

	prediction errors %	
Program name	1 cluster	3 clusters
astar	1.94	1.12
bzip2	2.13	1.12
gcc	3.30	2.46
gobmk	0.72	0.99
hmmer	0.21	0.10
h264ref	0.99	1.16
libquantum	1.83	0.57
mcf	1.84	0.51
omnetpp	1.81	0.48
perlbench	2.16	1.99
sjeng	0.57	0.72
xalancbmk	2.82	2.27
avg of int	1.69	1.13
bwaves	0.44	0.28
milc	2.90	1.24
zeusmp	1.36	1.50
gromacs	0.22	0.22
leslie3d	3.17	2.69
namd	0.15	0.15
dealII	1.38	1.67
soplex	1.50	0.54
povray	0.29	0.30
GemsFDTD	0.50	1.51
tonto	1.26	1.52
lbm	2.31	0.50
sphinx3	3.53	3.10
gamess	0.17	0.15
calculix	0.75	0.77
cactusADM	4.46	4.34
avg of fp	1.52	1.28

In Table 4.4, program `libquantum`, `mcf`, `omnetpp`, `soplex` and `lbm` have relatively high improvement, strongly suggesting these program follow the same transition pattern. By once again examining Figure 4.2, we can observe that `GemsFDTD` and `dealII` are visually mis-classified, so they have slightly decreasing in prediction accuracy. `leslie3d` might need to put into `libquantum` group to have a better prediction result. Other programs' prediction accuracy either stays the same or is slightly improved. Nevertheless, the overall prediction accuracy has been improved, indicating clustering definitely helps. There is no unique distance metric to fulfill the clustering tasks, therefore additional information might be incorporated to yield a better clustering result.

Table 4.5 compares the results using two different criteria for clustering programs. The overall prediction accuracy is about the same. To be more specific, programs such as `leslie3d`, `dealII` are in cluster one using sensitivity as criterion. But they are in cluster two if using parameter as criterion, which yields better accuracy. This suggests that by only looking at sensitivity curve itself might not necessarily gives enough information. Other than that, the rest prediction accuracy with these two different criteria are very similar to each other. Further study may be needed to give a more reasonable clustering criterion.

Table 4.6 and Table 4.7 show the sensitivity curve prediction on INTEL i5 machine with seven clusters and three clusters respectively. Using clusters improves the overall prediction result. Prediction accuracy further improved as the number of clusters increases. However, having too many clusters can result in over-fitting problem, therefore, we use a partition of three clusters for the rest of experiments in this chapter.

By observing those five groups of cross architecture sensitivity prediction result, we find certain programs are always have low prediction accuracy no matter what hardware it runs on, such as `bzip2`, `bwaves`, `libquantum`, `leslie3d` and `lbm`, which also suggest that those benchmarks are of different access pattern compared to others. Experiment shows that sensitivity prediction improved significantly for these programs and so as the overall prediction as number of clusters increases. Programs such as `mcf`, `omnetpp`, `libquantum`, `xalancbmk`, are very sensitive toward pressure, so they are grouped as one cluster. `sjeng`, `perlbench`, `gobmk`, `hmmer`, `h264ref`, are grouped together as these programs are insensitive toward contention. `lbm`, because of its unique accessing pattern, stand alone as a different cluster. Therefore, the prediction accuracy improvement for this program may not fair since it both serve as the training and testing program. Nevertheless, as more and more programs profiled on the source machine, some of programs' accessing pattern might match `lbm` and thus yield good prediction accuracy.

Clustering should also apply to cross-architecture pressure prediction. As mentioned in previous example, when it comes to actual co-run scenario with real applications, the pressure score of a program varies according to the pressure its peer runner gives. For a better prediction result, each application should have its own representative reporter. However, this will jeopardize O(n) complexity and moreover, this is not necessary as the pressure of program varies by at most 2MB and some of the program is even not sensitive to such changes. The trade-off between prediction accuracy and complexity determine how many clusters it should be for this pressure prediction task. We adopt three type of reporters from Smashbench [70], one is a blockie bubble, one a binary search tree bubble and one is a er-naive bubble. These 3 bubbles are all accessing approximately 20MB array space, however, they have different last level cache miss ratios and memory bandwidth consumption. As shown in Figure 4.3, the distribution of last level cache misses and memory bandwidth consumption of the 3 kernels at different sizes is given. BST has high miss ratio but low memory bandwidth consumption, and blockie consumes much more bandwidth, and er-naive is in-between of the two. They resemble aggressive, mild and average programs, respectively in terms of pressure. The actual program are then divided into 3 clusters and corresponding reporters are selected to generate the score.

Table 4.9 shows the cross-architecture performance degradation prediction result with clustering as preliminary step. The table lists both the predicted slowdown and the actual slowdown of a subset of SPEC CPU2006 integer programs when they co-run with different programs, including `gobmk`, `libquantum`, `h264ref`, `bwaves`. Table 4.10 further compares the performance prediction with and

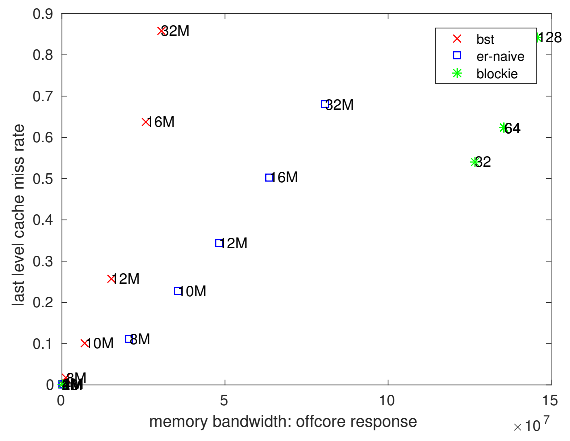

Figure 4.3: Smashbench kernel distribution

without clustering. Each workload corresponds to 2 columns, where the left one is the prediction error with the original method and the right one is the prediction error using cluster-based cross-architecture prediction framework. Without clustering, `gobmk` and `h264ref` have lower error rate but `libquantum` and `bwaves` have significant prediction error. Some of them even have an error over 15%. And notice that the sensitivity prediction without clustering is already within 2-3% error, thus the huge error must result from the inaccuracy of pressure score the reporter gives, which strongly suggests that using a unique score for a program towards different co-runners is not appropriate. With the clustering and multiple cross-architecture modeling for both sensitivity and pressure, the prediction accuracy has been significantly improved.

Table 4.5: Sensitivity prediction from c2 to i5 using cluster extra

	prediction errors %		
Program name	1 cluster	3 clusters	3 clusters (para)
astar	1.94	1.12	0.75
bzip2	2.13	1.12	1.14
gcc	3.30	2.46	2.62
gobmk	0.72	0.99	1.08
hmmer	0.21	0.10	0.26
h264ref	0.99	1.16	1.29
libquantum	1.83	0.57	0.78
mcf	1.84	0.51	0.58
omnetpp	1.81	0.48	0.51
perlbench	2.16	1.99	1.87
sjeng	0.57	0.72	0.82
xalancbmk	2.82	2.27	1.84
avg of int	1.69	1.13	1.13
bwaves	0.44	0.28	0.79
milc	2.90	1.24	1.65
zeusmp	1.36	1.50	1.58
gromacs	0.22	0.22	0.22
leslie3d	3.17	2.69	1.83
namd	0.15	0.15	0.14
dealII	1.38	1.67	0.19
soplex	1.50	0.54	1.06
povray	0.29	0.30	0.31
GemsFDTD	0.50	1.51	0.94
tonto	1.26	1.52	1.67
lbm	2.31	0.50	0.93
sphinx3	3.53	3.10	3.21
gamess	0.17	0.15	0.14
calculix	0.75	0.77	0.79
cactusADM	4.46	4.34	4.21
avg of fp	1.52	1.28	1.29

Table 4.6: Cross architecture sensitivity prediction for different machines w/o clustering

	Int prediction errors %		FP prediction errors %	
Architecture	w/o	w	w/o	w
INTEL i5	1.69	0.36	1.52	1.13
INTEL i7	2.80	1.74	2.89	1.15
INTEL Xeon	3.75	1.75	3.45	1.41
AMD A8	2.06	0.43	0.96	0.43
AMD A10	3.68	1.84	2.72	2.01

Table 4.7: Cross architecture sensitivity prediction for different machines w/o using 3 clusters

	Int prediction errors %		FP prediction errors %	
Program name	w/o	w	w/o	w
INTEL i5	1.69	1.13	1.52	1.28
INTEL i7	2.80	2.32	2.89	1.61
INTEL Xeon	3.75	2.80	3.45	3.01
AMD A8	2.06	1.38	0.96	0.84
AMD A10	3.68	2.78	2.72	2.52

Table 4.8: Performance degradation prediction for i5 using clustering

	prediction error%			
Program name	gobmk	libquantum	h264ref	bwaves
astar	0.17	1.62	1.53	3.54
bzip2	0.31	2.02	0.48	2.54
gcc	2.12	3.06	2.54	3.51
mcf	0.43	2.22	0.43	0.80
gobmk	0.17	2.25	0.34	0.17
libquantum	1.05	1.18	1.29	2.19
h264ref	0.12	0.83	0.12	0.62
hmmer	0.00	0.98	0.00	0.00
sjeng	0.45	2.02	0.45	2.50
perlbench	1.25	1.57	0.84	0.62
omnetpp	0.25	3.68	0.21	2.10
xalancbmk	1.30	5.11	0.32	1.20

Table 4.9: Performance degradation prediction for i5 using clustering

	prediction error%			
Program name	gobmk	libquantum	h264ref	bwaves
astar	0.46	1.57	0.42	1.47
bzip2	0.68	3.52	0.98	0.49
gcc	0.90	1.80	0.77	2.55
mcf	0.13	2.33	0.24	2.86
gobmk	0.17	2.61	0.34	1.00
libquantum	0.20	1.48	0.99	2.49
h264ref	0.19	0.83	1.02	0.12
hmmer	0.00	0.38	0.00	0.00
sjeng	0.05	3.30	0.07	1.10
perlbench	1.11	1.88	0.53	0.90
omnetpp	0.65	3.56	0.21	1.50
xalancbmk	2.30	3.31	0.32	2.11

Table 4.10: Performance degradation prediction original versus cluster-based method

Program name	prediction error%							
	`gobmk`		`libquantum`		`h264ref`		`bwaves`	
	w/o	w	w/o	w	w/o	w	w/o	w
astar	0.54	0.46	4.92	1.57	0.48	0.42	4.47	1.47
bzip2	1.08	0.68	10.52	3.52	1.48	0.98	1.49	0.49
gcc	0.79	0.90	7.30	1.80	1.17	0.77	8.55	2.55
mcf	0.17	0.13	13.35	2.33	0.34	0.24	5.68	2.86
gobmk	0.08	0.17	2.93	2.61	0.12	0.34	1.81	1.00
libquantum	0.35	0.20	4.75	1.48	0.58	0.99	12.10	2.49
h264ref	0.44	0.19	2.17	0.83	0.47	1.02	1.02	0.12
hmmer	0.00	0.00	0.50	0.38	0.00	0.00	0.00	0.00
sjeng	0.06	0.05	5.30	3.30	0.27	0.07	2.10	1.10
perlbench	1.81	1.11	6.38	1.88	1.33	0.53	1.61	0.90
omnetpp	0.65	0.65	15.88	3.56	0.33	0.21	3.50	1.50
xalancbmk	0.59	2.30	17.86	3.31	0.07	0.32	4.11	2.11

Chapter 5

Performance modeling for more than two cores

In Chapters 3 and 4, we show that by using a cross-architecture model, we can predict performance slowdown due to contention on both source and target machines with relatively high accuracy. Therefore we try to apply the framework onto a contention scenario of more than two participating applications. In this chapter, we show the limitation of our previous approach and propose a new bubble design. The newly constructed bubble can accurately simulate actual programs in terms of memory subsystem contention behavior. Moreover, the profiling cost can be significantly reduced while building the performance model using this new design.

5.1 Scalability issues for cross-architecture modeling for contentions

In light of the bubble-up approach, Chapters 3 and 4 show how to transfer knowledge of a program's sensitivity and pressure profiled on a source machine onto a target machine using a logistic and a polynomial model, respectively. Clustering techniques help to further improve the prediction accuracy by dividing programs into groups such that the ones within a group share similar behavior in terms of cache and memory contention characteristics.

The next question is how to expand the framework so that one can predict the performance degradation in a contention scenario for more than two cores. One way is to simply apply the same framework into a more-than-two-cores scenario. This simple approach involves reconstructing a program's sensitivity by co-running it with multiple bubbles rather than a single bubble. This also applies to the construction of the reporter's sensitivity curve. We take three-core co-run as an example. Suppose one needs to predict A's slowdown when it co-runs with B and C, the following steps need to be done:

1. Obtain A's three core co-run sensitivity curve: co-run A with two bubbles at different pressure scores. Originally, if the bubble pressure ranges from 1MB to 10MB, now there are 100 combinations from (1MB, 1MB) to (10MB, 10MB), as there are two co-run peers alongside A.

2. Run reporter with B and C to report a combined pressure score: co-run the reporter with two bubbles at different pressure score to obtain the reporter's three cores co-run sensitivity curve. Then co-run the reporter with B and C and look up the combined pressure score in the reporter's sensitivity curve.

3. Predict A's slowdown when co-run with B and C: Find A's performance degradation (y-axis) at the point where the pressure score (x-axis) matches the value obtained in step 2.

Note above steps are exactly the bubble-up methodology in a more than two core setting. Additional profiling needs to be done on a target machine and a cross-architecture model needs to be trained if one needs to predict performance degradation on that machine. We choose a small subset of programs to verify the framework. Initial experiments show that the prediction error is still within 2-3%.

By examining the steps stated above, we find a scalability issue in the framework. The profiling cost of sensitivity increases because of an increasing in contention pressure as the number of peers increases. And this process needs to be done for different number of co-run peers, which ranges from 2 to 8. Moreover, as the program co-runs with two or even more bubbles, the number on x-axis of the sensitivity curve becomes ambiguous. The bubbles in each combination can be of different values, thus a sensitivity curve becomes sensitivity grid, surface, or even hyper-surface.

Preliminary experimentation shows that multiple combinations of bubble pressures result in the same performance slowdown. To keep the sensitivity simple, we only profile a subset of those values, for example, bubbles in the combination are of same values in each profiling run. The performance degradation corresponding to bubble pressure of different values can only be obtained through interpolation, which will introduce errors. The profiling of combined co-run peer pressure is even more problematic. The SPEC CPU2006 benchmark suite contains 29 programs, there are 406 possible combinations of 2 co-run peers, and 3654 combinations of 3 co-run peers. Profiling the 29 single ones is time consuming. 406 to 3654 combinations will yield a prohibitive cost. Profiling time doubles if we need to build the cross-architecture prediction model.

One merit of bubble-up methodology is to decouple/linearize the co-run problem into sensitivity and pressure. Does the combined pressure of co-run programs have the linear property so that we can decouple it? Thus in a more than two core co-run scenario, one question is if we can decouple the combined program pressure as a simple summation. To be specific, taking the example of programs A, B and C co-running together again and supposing that B yields 3MB pressure and C yields 4MB when they co-run with A separately, can we directly use A's sensitivity at 7MB pressure to predict its slowdown? Experiments show that the prediction is way off the actual performance degradation as even though it is possible that 7MB matches the effective cache size B and C together, these two programs will consume memory bandwidth simultaneously and a single 7MB bubble cannot take as much memory bandwidth as B+C do. Then the next question is can we predict using A's sensitivity at the position where one bubble is at 3MB and the other one is at 4MB? Unfortunately, this also fails as the predicted slowdown is much longer than the actual one, which suggests that B and C's pressure changes (becoming smaller compared to bubble).

We use the performance counter while running the bubbles and actual programs in both solo-run and co-run scenarios to examine what happened during execution. Bubbles are small kernels in which an array is accessed in random/sequential pattern and the size of the array can be tuned, usually from 0 to the last level cache size. The memory bandwidth consumption is usually extremely small as most accesses are covered by last level cache. However, the actual program often has higher memory bandwidth consumption, and for a two-core co-run case, the actual program and bubble reach a convergence point where their the memory bandwidth consumption and last level cache misses match. (In more detail: the memory bandwidth consumption of the bubble increases sharply and memory bandwidth consumption of an actual program increases or decreases slightly). It is expected that the prediction is accurate as the bubble behaves like the real program. In three cores co-run cases, the memory bandwidth consumption of the bubble keeps increasing and these hardware statistics diverge again and bring in prediction errors. The starting point of a bubble and an actual program is not matching and it is coincidence they intersect in two-core co-run case, there is no guarantee the change of their behavior can intersect again at three or four cores co-run cases, thus it is likely to make a wrong prediction as the number of peers increases. A better way to solve this problem is to create a bubble that simulates the critical code segments of an actual application so that their behavior matches no matter how many co-run peers there are in the group. In such settings, we use a bubble with specific parameters, such as array size, accessing stride size, etc., to represent the actual program. Therefore, we eliminate the steps required for combined pressure profiling while at the same time eliminates the execution alignment issue as bubble has no phase changes and the execution time can be controlled.

5.2 Program-specific bubble design

By instrumenting application memory traces, we can identify hot sections that trigger most last level cache hits and misses. Moreover, related cache access pattern and working set size can also be obtained. The code segments can be extracted from the source file and put into a bubble kernel to represent the actual application. The kernel is expected to behave as the actual program as they share same data structure, accessing pattern and memory footprints. Compared to the bubble-up framework, our program-specific bubble is created off-line and can be used without measuring bubble pressure score to match the actual one no matter if co-run peers are present or not. The overall design is shown in Figure 5.1.

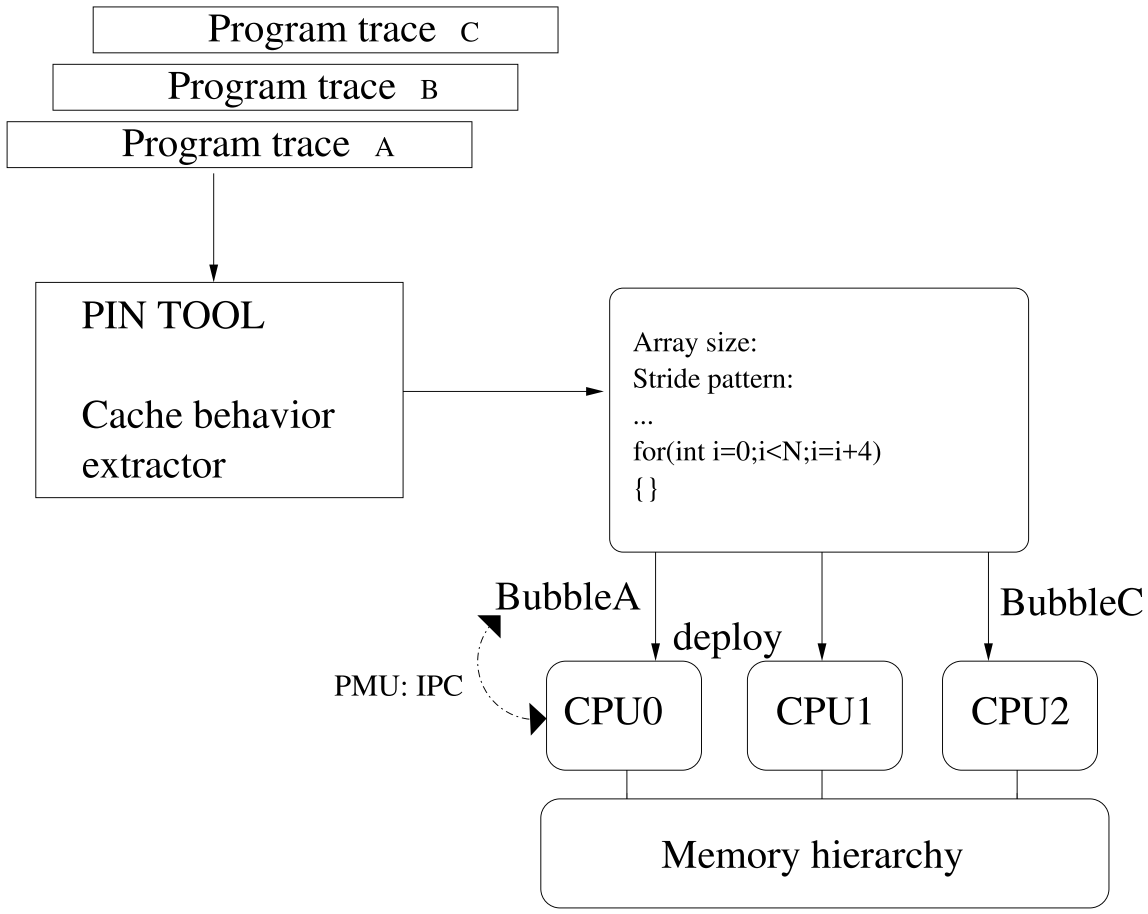

Figure 5.1: Program-specific bubble prediction framework

Suppose we have an interest in how program A performs in terms of execution slowdown when it co-runs with peer runners B and C. Individual programs A, B, C are then fed into the instrumentation module to have their cache access behavior extracted. The module locates the critical code segments in the source file and copies them into the bubble kernel associated with the program-specific access pattern. The closeness of solo run hardware statistics between an actual program and its corresponding bubble indicates how well the match is. Oftentimes, the bubble exhibits higher pressure and minor adjustment needs to be injected to alleviate the effect. As the program IPC is strongly related to the execution time, we can deploy bubbles onto cores to simulate a corresponding program co-run group and make prediction by monitoring IPC changes. In the next section, we will show how to extract the hot code segments and their corresponding accessing pattern using instrumentation tools in detail.

5.3 Identify critical code segments with instrumentation

There are several ways to profile a program's accessing behavior. We choose Intel's Pin tool to fulfill the task as it provides users with a significant amount of useful APIs which allows one to profile a program at different granularity. It also implements the basic structure of the cache simulator which is a core utility in understanding cache behavior. One can instantiate the cache simulator by assigning parameters such as set associativity and size corresponding to the L1, L2 and last level cache, and make sure that the cache replacement policy matches the actual hardware policy. The simulator can update statistics related to all memory accesses, which reflects the runtime cache behavior as if the program is running on actual hardware architecture.

5.3.1 Identify instruction miss/hit last level cache the most

Modern architecture usually has two to three levels of processor cache. The first one or two layers of cache are smaller but faster and private to each core. The third layer is larger but slower and is usually shared by all cores on the chip. One could create a cache simulator exactly matching the actual hardware. We argue that one can just simulate the last level cache for simplicity as the shared cache accessing behavior is of focus. We execute all SPEC CPU2006 benchmark programs through a simplified cache simulator following algorithm 3. Note that every program in the benchmark suite has been re-compiled with "-ggdb" option on so that information such as the line numbers in the source code associated with the identified hot instructions will be obtained through the debugging tool. Previous experimental results show that programs with a higher miss rate usually exert higher pressure toward peer runners. This implies that instructions that trigger cache misses will bring pressure. This suggests that we should identify instructions that misses the cache most and extract them to simulate the actual program. A hit-miss balance issue needs to be considered since the bubble will give higher pressure than an actual program if only those last level cache sensitive access patterns are simulated. Similar to adding water to dilute concentrated juice to make it tastier, we might need to add top hit instructions, which access memory with good localities, so that we can dilute bubble pressure in order to make the hardware statistics match the actual program.

Algorithm 3: Identify instructions hit/miss cache the most

```
Data: Program executables and ref input
Result: top 50 instructions that trigger most last level hits/misses
initialization: create 2 hash tables, one for cache hits, one for cache misses;
the key is instruction program counter(PC) and value is the hit/miss count throughout the
  execution of program;
while application running do
    examine current instruction;
    if Is a memory read/write then
        extract memory address associated with this load/store;
        run it through cache simulator;
        if Is a cache hit then
            hash[pc][hit]+1
        else
            hash[pc][miss]+1
        end
    else
        continue;
    end
end
sorting hash table by values;
output top 50 hit/miss instructions
```

5.3.2 Profiling access behavior of last level related instructions

With the gdb debugging tool, we can identify the source code locations that trigger most last level cache hits/misses. But a significant piece of information is missing as we barely have any information about the accessing pattern of that line of code. Therefore, another round of profiling needs to be done to tile the jigsaw puzzle. To be specific, the identified instructions are usually array or pointer calculations across a memory chunk. Therefore, we should understand the boundary of the memory related to that array access and how the program iterates through the array, either randomly or sequentially.

Boundary and range: this information is easy to obtain, during the execution of a program, we can keep track of the lowest memory address and highest memory address and calculate the range by doing subtraction between the two. Special care needs to be taken for some cases. For example, suppose the program is accessing multiple data chunks. Each data chunk is small enough that it can be fit into the cache, but there can be a huge gap between chunks. This scenario is common in programs, as those tight data chunks are allocated dynamically and indexed by pointers. Therefore, a much larger range will be obtained than actual one if we use simple subtraction. This can be detected by the algorithm 4 and we will explain in detail later.

Random/Sequential access pattern: we can keep track of the difference between two consecutive accesses and create a histogram of such difference. This helps to understand the access pattern of a memory related instruction. Moreover, the maximum/minimum difference in the histogram also reveals the boundary of a tight memory chunk, as a program tends to access more often within each tight data chunk than jump between memory chunks. We can filter out those "jump differences" whose count value is smaller than a pre-defined threshold. Hardware prefetcher always monitors the memory access pattern and will fetch data in advance if it identifies some fixed accessing pattern. We can incorporate this idea and capture the fixed accessing pattern by recording historical data.

To be specific, we record the last memory address and last memory difference, which is the difference between the last memory address and the one an instruction referred to before last memory address. We calculate the current memory address difference and compare it with the last memory difference. If they are identical, it is highly possible that the instruction is running in a stride pattern. If they don't match, it means the last continuous stride sequence has ended, thus we may start a new sequence and add the total number of sequences corresponding to the last stride value by one. With this information, we can also obtain the average sequence length of a certain stride value by using the total number of occurrences of that stride divided by the total number of segments corresponding to it. This also reveals the changing pattern of the loop carried variable of the innermost loop which contains the instruction.

```
Algorithm 4: Identify accessing behavior
Data: Program executables, ref input with program counter for each identified instructions
Result: accessing range, stride information for each candidate instruction
initialization:
while application running do
    if instruction PC matches one of the candidates then
        extract memory address associated with this load/store;
        update mem_max and mem_min if neccessary
        ; if (current_mem-last_mem == last_diff) then
            stridemap[diff].cnt++;
            stridemap[diff].curseg_cnt++;
        else
            if stridemap[last_diff].curseg_cnt< threshold then
                randommap[last_diff]+=stridemap[last_diff].curseg_cnt;
                stridemap[last_diff].cnt -= stridemap[last_diff].curseg_cnt;
                stridemap[last_diff].curseg_cnt=0;
            else
                stridemap[last_diff].curseg_cnt=0;
                stridemap[last_diff].seg++;
                randommap[diff]++;
            end
        end
    else
        continue;
    end
end
```

Note in algorithm 4, two hash tables are used for recording the histogram for stride and random access patterns. For the random access hash table, *key* is memory difference, and *value* is the count. For the stride access hash table, *key* is stride and *value* is a structure including: count (cnt), which is the total number of occurrences of a specific stride access, segments (seg), and current segment count (curseg_cnt), which is the total number of consecutive accesses corresponding to a specific stride value in the current segment.

5.4 Bubble creation using instrumentation results

Following the steps mentioned in algorithm 3 and 4, we profile every program in SPEC CPU2006 benchmark suite using Pin. In Chapter 3, we show that prediction error is high if the co-run group contains certain programs, such as **libquantum**, **lbm**, etc. In this section, we pick four of them and show their accessing pattern and how we generate program-specific bubbles out of the profiling result.

5.4.1 case study: soplex

450.soplex is based on SoPlex Version 1.2.1. SoPlex solves a linear program using the Simplex algorithm. Linear program is to maximize/minimize an object function with several constraint conditions as the following linear algebra form.

$$\begin{aligned} &\text{minimize} \quad && C^T X \\ &\text{subject to} && AX \leq b \\ &\text{with} && X > 0 \end{aligned}$$

where X is the vector that needs to be solved for, constraints are expressed in the form of multiplication of a sparse matrix A and a vector X.

The solution of finding vector X is straightforward. As the constraints define the searching space where we can only find the minimum objective function value within the area (inclusive). In fact, the area is a convex polygon, the optimal value is usually located at a vertex of the polygon. The algorithm iteratively finds an updating direction through which the objective function gains the most, moves toward to one vertex each time and keeps finding the next candidate direction until no candidate direction is possible. Due to the nature of this problem, the matrix usually is a sparse one.

450.soplex has two reference inputs and we take the second input as our case study example. The program accepts the input and transforms it into linear algebra form. We feed the program into Pin and identify that code section in Listing 5.1 has either top last level cache hit count or top last level cache miss count.

Listing 5.1: critical code in soplex

```
//ssvector.cc
SSVector& SSVector::assign2productFull(const SVSet& A, const SSVector& x)
{
        ...
        for(i=x.size(); i-- >0; ++xi)
        {
                svec = const_cast<SVector*>(& A[*xi]);
                elem = &(svec->element(0));
                last = elem + svec->size();
                y = vl[*xi];
                for (; elem < last; ++elem)
                v[elem -> idx] +=y * elem->val; //985

        }
        ...
}
```

Line 985 is one of the hot instructions, located in ssvector.cc. From the name of the function containing that instruction we can guess this function calculates multiplication of matrix A with vector X. Each time a row from A is picked as svec. And the inner loop iterates through that row and calculates the result of the multiplication. As the matrix is a sparse one, to save space, the representation of a row in such matrix is a dense but much shorter array, with each element recording the value and index corresponding to the sparse one, elem->idx stores the position information and elem->val is the actual value at corresponding position. The Pin tool finds three assembly instructions out of this code section as two loads and one store, which are reads to elem->val, elem->idx and stores of v[elem->idx]. Let's examine the access pattern of these three instructions.

elem->idx and elem->val's accessing range is 128MB and the stride is 16 byte long. elem is a structure that only contain two members, idx and val, each member is an 8-byte long variable. We can clearly see from listing 5.1 that the accessing of elem is purely sequential.

v[elem->idx] accessing range is 7.3 MB. The accessing pattern is a mixture of sequential access of 8 byte for each stride (65% of time during execution) and random access. Array v is a Real type array, which is a double type that is 8-bytes long. Thus the memory difference of 8 indicates a sequential access pattern, and other memory difference suggests the array is accessed with a stride of several elements. The matrix A's size is 2586*920683. A simple calculation of 920683*8 = 7365464 (byte) shows that v's size is identical to the size of a single row in A. The fact that average stride sequence length of v is much smaller than A's column size also indicates the sparsity of A.

Listing 5.2: critical code in soplex

```
//svector.h
Real operator*(const Vector& w) const
{
        Real x=0;
        int n = size();
        Element* e = m_elem;

        while(n--)
        {
                x += e->val * w[e->idx]; //295
                e++;
        }
        return x;
}
```

Similar to the previous code segment analysis, line 295 is identified as a hot instruction, which is located at svector.h file. The access range of e is 142MB and range of w is 20680 bytes. The function overloads the operator "*" for vector multiplication. Throughout the execution, the piece of code iterates through the sparse matrix, the resulting access range is similar to the previous code segment. The access pattern is sequential access with a 16-byte-long stride as e shares the same structure with elem. w is an array whose length is the same as the row size of A as 2586*8 = 20688 (bytes). The accessing pattern of w is a mixture of sequential access of 8-bytes (20%) and random access (80%) with majority ranging from 8-bytes to 104-bytes. Array v is a Real type array, which is actually a double type of 8 bytes long. Thus, the memory difference of 8 indicates a sequential access pattern, and an accessing range from 8 to 104 bytes suggests the array is accessed mostly within a stride ranging from one element to 13 elements. Function *assign2productFull* in **soplex** is identified as a pathological region in [122]. This is consistent with our Pin tool result.

5.4.2 CASE STUDY: LIBQUANTUM

462.libquantum is a C library simulating quantum computer. The program exhibits significant high pressure to other peer runner and due to the limited capability of bubble used in two-core cross architecture prediction framework. The slowdown prediction for co-run groups including **libquantum** usually yields large error. We run this program through Pin to understand run-time cache behavior, which will be described in listing 5.3.

Listing 5.3: critical code in libquantum

```
//gate.c
void quantum_toffoli(int control1, int control2, int target, quantum_reg * reg)
{
        ...
        for(i=0; i<reg->size; i++)
    {
        if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
        {
                if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
            {
                reg->node[i].state ^=((MAX_UNSIGNED) 1 << target);
            }
            }
        }

            ...
}
```

Listing 5.4: critical code in libquantum

```
//gate.c
void quantum_sigma_x(int target, quantum_reg * reg)
{
        ...
    for(i=0;i<reg->size;i++)
    {
        reg->node[i].state ^=((MAX_UNSIGNED) 1 << target);
    }

    ...
}
```

These two pieces of code are identified by our Pin tool and located in gate.c. These code segments stand out not only because of their high last level hit/miss count, but also their high execution frequency, which shows at least 10x more than other hot instructions. We identify this section as the critical part of the program. Interestingly, the structure of **libquantum** is one of the simplest one as the access range is 64MB and access pattern is sequential access of 16-bytes long, which can be confirmed by inspecting the source code directly. reg->node points to an array with each element being a structure, the structure only contains two fields, a floating point variable "amplitude" of 8-byte long and an unsigned long long type variable "state" of 8-byte long, a sequential access of such array will have a stride of 16-byte long.

5.4.3 CASE STUDY: LBM

470.lbm is another program which gives high contention pressure and results in the worst prediction accuracy. It implements the lattice boltzmann method to simulate fluid dynamics.

Listing 5.5: critical code in lbm

```
void LBM_performStreamCollide( LBM_Grid srcGrid , LBM_Grid dstGrid ) {
...
SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
...
u2 = 1.5f * (ux*ux + uy*uy + uz*uz);
DST_C(dstGrid) = (1.0f-OMEGA)*SRC_C(srcGrid) + DFL1*OMEGA*rho*(1.0f- u2);
...
SWEEP_END
}
```

In fact, multiple lines of code in this LBM_performStreamCollide function are identified as hot instructions. The program makes extensive use of macros and it is hard to understand them without knowing their definitions. We track these macros and in general, the program performs calculation within a 3D grid, with its size equal to 100*100*130. Every point in this grid has 20 directions. The program simulates particle movement and it involves with two grids, one serves as source and the other one serves as destination and these two grids are flipped in each iteration. The for loop iterates through every point in the grid sequentially. However, due to additional calculations to the index in source and destination array, within each iteration, the source grid has both good temporal and spatial locality while the destination grid have bad locality.

We attempt to create the corresponding bubble by copying the source to destination assignment block in the for loop and give explicit index to these two arrays. However, this results in a bubble with extremely low IPC compare to the actual **lbm** itself. We then adopt all the macro definition in the original header file and copy the content in listing 5.5, the bubble yields very close imitation in terms of hardware statistics. These macros generate a large of computing instructions which yield a high IPC.

5.4.4 CASE STUDY: MCF

429.mcf is derived from MCF, which is a program used for the simulation of a single-depot vehicle scheduling problem in a public mass transportation setting.

Listing 5.6: critical code in mcf

```
//mcfutil.c
#ifdef _PROTO_
long refresh_potential( network_t *net )
#else
long refresh_potential( net )
    network_t *net;
#endif
{
    node_t *node, *tmp;
    node_t *root = net->nodes;
    long checksum = 0;
```

```
root->potential = (cost_t) -MAX_ART_COST;
tmp = node = root->child;
while( node != root )
{
    while( node )
    {
        if( node->orientation == UP )
            node->potential = node->basic_arc->cost +
            node->pred->potential;//86
        else /* == DOWN */
        {
            node->potential = node->pred->potential -
            node->basic_arc->cost;
            checksum++;
        }

        tmp = node;
        node = node->child;
    }

    node = tmp;
```

```
            while( node->pred )
            {
                tmp = node->sibling;
                if( tmp )
                {
                    node = tmp;
                    break;
                }
                else
                    node = node->pred;
            }
        }

        return checksum;
}
```

The instruction that triggers most last level cache misses is located at line 86 in mcfutil.c file. The single-depot vehicle scheduling problem is expressed as tree nodes and arcs connecting them. The function traverses the entire tree structure and updates *potential* for each node. MCF uses an n-nary tree where each node can have multiple children rather than two. Therefore, it uses a left-child, right-sibling implementation to construct the tree, where the left child node is current node's very first child and the right child node is the sibling of current node. Even though the allocation of all nodes is done at the initialization stage and are within continuously memory chunk, the connectivity of the tree structure changes over time thus the traversal usually can be considered as a random access.

Our Pin tool finds that the memory access pattern associated with this instruction is random, but the memory difference is of a multiple of 104 bytes and the access range is 5228496 bytes. 104 bytes is the structure size of a single node and based on input file, and there are 25137 nodes. For calculation purpose, the program doubles the node numbers and 25137*2*104 = 5228496 bytes.

The program also involves arc calculation. The program allocates an array of arc structure (64 bytes) at the initialization stage with an upper-bound of 0x1a1000L total arcs. $0x1a10000L * 64 \approx 1.75GB$, which is consistent with the footprint size during execution.

Listing 5.7: Critical code in mcf

```
//pbeampp.c
#ifdef _PROTO_
arc_t *primal_bea_mpp( long m,  arc_t *arcs, arc_t *stop_arcs,
                                    cost_t *red_cost_of_bea )
#else
arc_t *primal_bea_mpp( m, arcs, stop_arcs, red_cost_of_bea )
     long m;
     arc_t *arcs;
     arc_t *stop_arcs;
     cost_t *red_cost_of_bea;
#endif
{
     ...


NEXT:
     /* price next group */
```

```
        arc = arcs + group_pos;
        for( ; arc < stop_arcs; arc += nr_group )
        {
            if( arc->ident > BASIC )
            {
                red_cost = arc->cost - arc->tail->potential +
                arc->head->potential;
                    ...
            }
        }
}
```

The calculation with red_cost involves several assembly instructions. Pin tool finds that the accessing range is 1.75GB and accessing pattern is stride access as 90% of the access is a stride of 5830144 byte. By examining the source code, we find that a total number of 0x1a1000L arcs are divided into groups and each group contains 300 arcs. The variable nr_group is equal to 0x1a1000L divided by 300. And the value of nr_group multiplies 64 bytes yields 5830144 bytes. With the Pin tool result, we don't have to go into details of source file and just create a stride access bubble to simulate such hot block.

SPEC CPU2006 benchmark programs are analyzed using Yoo et al. [122] approach and pathological code segments are identified. The functions *replace_weaker_arc* and *refresh_potential* in **mcf** are identified as pathological regions: the first one is array access- or memory-intensive and the second one is linked-list last level cache intensive. This is consistent with our PIN analysis result, as there are a significant number of nodes are allocated and organized in a tree structure, and the function *refresh_potential* traverses the tree and updates tree node values, which is a linked-list style access pattern. The number of nodes multiplies the size of each node is close to last level cache size, which makes it a last level cache intensive. Moreover, there are a huge number of arcs created between those nodes and it occupies up to 1.75GB memory space, which makes the *replace_weaker_arc* a memory intensive pathological region.

5.5 Evaluation for performance prediction using new bubbles

Ideally, the bubble is a perfect matching in every aspect and can be used alternatively with the actual applications without changing the overall cache accessing behavior of a co-run group. Thus, the performance prediction can be done by co-running all representative bubbles together and observing IPC changes. The merit of this design is it extremely reduces the prediction overhead to a constant time, whereas profiling the actual program takes a much longer time, and the cost grows exponentially with the number of participating peers. Nevertheless, a significant portion of source code in an actual program is trimmed out in our bubble kernel, and there might exist discrepancies for the bubble in simulating the actual one except for cache/memory-related behavior. As we cannot guarantee other metrics such as IPC changes at the same pace with the actual program's IPC changes, using bubbles alone might bring prediction error.

Therefore, in the remaining section, we conduct the experiment using the below steps.

1. Create the program-specific bubbles using Pin.
2. Randomly pick programs from clusters to form co-run groups.
3. Run actual programs together to record performance degradation.
4. Pick corresponding bubbles and co-run them together to predict performance based on IPC changes, and compare with the result generated in step 3.

5. Keep one actual program, co-run it with bubbles that represent other programs within the group, and record the actual program's slowdown. Compare it with the result generated in both step 3 and step 4.

There are 29 programs in the SPEC2006 benchmark suite. To verify our framework for a more than two-core scenario, let's first examine how many combinations there are for a possible co-run group. There are $\binom{29}{2} = 406$ possibilities for two-core co-run, $\binom{29}{3} = 3654$ possibilities for three-core co-run , and $\binom{29}{4} = 23751$ possibilities for four-core co-run cases. The number of combinations sharply increases as the number of co-run peers increases. Moreover, most programs exhibit phase behavior and they give higher pressure in one or more phases and give lower pressure in other phases. To guarantee the accuracy of the profiled co-run performance degradation result, all the programs in a group should be run at least once to prevent a biased pressure. As the execution times of programs are not aligned perfectly, some of the programs need to be run multiple times and this extends the profiling times even longer. Due to these limitations, we can only pick a subset of the combinations to verify our framework.

We use clustering to identify representative co-run groups. The 29 benchmark programs are clustered in groups based on three distance metrics. As we are using newly designed bubbles, it might be meaningless if we use logistic function parameters of the sensitivity curves of those programs as the clustering criterion. In contrast, we can make use of performance counters and categorize programs as cache-benign ones or aggressive ones based on hardware statistics. We still use the k-means algorithm and set k to be the optimal number and randomly pick one program from those candidate clusters. Therefore, for a five-cluster separation, we can have 10 combinations for a three core co-run group or five combinations for a four core co-run group. To create even more combination, and we can pick more than one program from a cluster. For extreme cases, we can pick everything from just one cluster, which might generate special cases whose group pressure is extremely small or high.

We program the performance counter to monitor hardware events L3_LAT_CACHE.MISS, L3_LAT_CACHE.MISS, OFF_CORE_RESPONSE_0 and INST_RETIRED.ANY_P, which provide us with the metric for memory bandwidth consumption and cache access behavior. The statistics for all SPEC CPU2006 benchmark solo executions on an Intel i7-920 machine are shown in Table 5.1. The values shown in the table are the average count of the corresponding event for every two seconds.

Table 5.1: SPECCPU06 benchmark programs solo execution hardware statistics for Intel i7

Program	**MBW**	**miss ratio (%)**	**miss rate**	**IPS**
astar	$8.92 * 10^6$	10.40	0.0036	$4.4 * 10^9$
bzip2	$1.06 * 10^6$	1.71	0.0001	$7.39 * 10^9$
gcc	$3.88 * 10^7$	27.30	0.006	$5.51 * 10^9$
gobmk	$4.58 * 10^6$	27.70	0.0006	$5.84 * 10^9$
mcf	$6.33 * 10^7$	40.40	0.038	$1.69 * 10^9$
hmmer	$0.25 * 10^6$	1.27	0.000005	$5.87 * 10^9$
h264ref	$0.35 * 10^6$	2.25	0.000027	$8.47 * 10^9$
libquantum	$1.40 * 10^8$	94.00	0.0039	$5.77 * 10^9$
omnetpp	$4.59 * 10^7$	45.70	0.013	$3.11 * 10^9$
perlbench	$3.41 * 10^6$	17.00	0.00025	$8.98 * 10^9$
sjeng	$2.71 * 10^6$	44.00	0.00039	$6.90 * 10^9$
xalancbmk	$2.37 * 10^7$	27.00	0.0031	$7.48 * 10^9$
bwaves	$6.52 * 10^7$	64.00	0.0021	$6.39 * 10^9$
gamess	$0.034 * 10^6$	1.05	0.0000025	$9.87 * 10^9$
milc	$1.31 * 10^8$	97.00	0.016	$4.90 * 10^9$
zeusmp	$2.66 * 10^7$	64.00	0.0021	$5.88 * 10^9$
gromacs	$0.55 * 10^6$	1.54	0.000033	$6.07 * 10^9$
cactusADM	$1.10 * 10^7$	56.60	0.003	$3.83 * 10^9$
leslie3d	$7.77 * 10^7$	60.20	0.003	$6.28 * 10^9$
namd	$0.53 * 10^6$	18.00	0.000017	$8.18 * 10^9$
dealII	$2.04 * 10^7$	35.50	0.00064	$7.78 * 10^9$
soplex	$1.06 * 10^8$	43.00	0.0075	$4.68 * 10^9$
povray	$0.016 * 10^6$	0.67	0.0000025	$7.27 * 10^9$
calculix	$1.89 * 10^6$	25.00	0.0000426	$9.77 * 10^9$
GemsFDTD	$7.55 * 10^7$	57.00	0.0054	$4.58 * 10^9$
tonto	$2.61 * 10^6$	2.59	0.0001	$8.00 * 10^9$
lbm	$1.18 * 10^8$	34.80	0.0026	$6.5 * 10^9$
sphinx3	$1.04 * 10^7$	7.84	0.00055	$8.47 * 10^9$

We use the memory bandwidth consumption (MBW), cache miss ratio and cache miss rate as the criterion for program clustering. MBW not only reflects how many main memory accesses are due to last level cache misses but also shows the prefetcher behavior, which is strongly related to fixed stride accessing patterns. Cache miss ratio reflects the locality of a program and cache miss rate also reveals how many memory related instructions are there and hit/miss ratios per unit time. The data is three dimensional and scaled using the raw value divided by the difference between maximum value and minimum value from each corresponding dimension. The clustering results are listed in Table 5.2. There are five clustering groups in total.

Table 5.2: Clustering of programs based on hardware statistics

Cluster No.	Programs
1	libquantum, milc
2	gcc, omnetpp, xalan, zeusmp, dealII
3	mcf, bwaves, leslie3d, GemsFDTD
4	soplex, lbm
5	perlbench, bzip2, gobmk, hmmer, sjeng, h264ref, astar,gamess, gromacs, namd,tonto, povray, sphinx3, calculix,cactusADM

libquantum and **milc** are grouped together in a cluster as they both have over 90% last level cache misses and high memory bandwidth consumption. We previously show the hot sections in **libquantum** and the hot section in **milc** is shown in listing 5.8 (both **libquantum** and **milc** are implemented in C).

Listing 5.8: critical code in milc

```
//s_m_a_mat.c
for(i=0;i<3;i++){
        for(j=0;j<3;j++){
```

```
    c->e[i][j].real = a->e[i][j].real+ s*b->e[i][j].real;
    c->e[i][j].imag = a->e[i][j].imag+ s*b->e[i][j].imag;
    }
}
```

Listing 5.8 is a code segment in *s_m_a_mat.c*, identified as the hot section by our Pin tool. The accessing array is much smaller than **libquantum**, and the program has good locality as it accesses the array sequentially. The similar hot section is also identified as the *su3_projector* function in *su3_proj.c*, which executes complex number calculations in a sequential manner. Moreover, the function uses three pointers a, b and c as inputs. And the function is called inside a loop, iterating all possible a, b, c in a sequential pattern and those possible objects are allocated as a continuous memory chunk. The only difference is that **milc** is accessing multiple arrays and that might be the reason it has four times as many the last level cache misses per instructions as does **libquantum**.

459.GemsFDTD solves the Maxwell equations in 3D in the time domain using the finite-difference time-domain (FDTD) method. The code is written in FORTRAN. Two sections of code in update.F90 stand out as they trigger the most last level cache misses. The access range is around 92MB and over 95% of the memory accesses follow a stride of 8 bytes long.

Listing 5.9: critical code in GemsFDTD

```
do k=1,nz
  do j=1,ny
    do i=1,nx
      Hx(i,j,k) = Hx(i,j,k)+((Ey(i,j,k+1)-Ey(i,j,k))*Cbdz)
      +(Ez(i,j,k)-Ez(i,j+1,k))*Cbdy)
      Hy(i,j,k) = Hy(i,j,k)+((Ez(i+1,j,k)-Ez(i,j,k))*Cbdx)
      +(Ex(i,j,k)-Ex(i,j,k+1))*Cbdz)

      Hz(i,j,k) = Hz(i,j,k)+((Ex(i,j+1,k)-Ex(i,j,k))*Cbdy)
      +(Ey(i,j,k)-Ey(i+1,j,k))*Cbdx)
    end do
  end do
end do
```

Listing 5.10: critical code in GemsFDTD

```
do k=2,nz
  do j=2,ny
    do i=2,nx
      Ex(i,j,k) = Ex(i,j,k)+((Hz(i,j,k)-Hz(i,j-1,k))*Dbdz)
      +(Hy(i,j,k-1)-Hy(i,j,k))*Dbdy)

      Ey(i,j,k) = Ey(i,j,k)+((Ez(i,j,k)-Ez(i,j,k-1))*Dbdx)
      +(Ex(i-1,j,k)-Ex(i,j,k))*Dbdz)

      Ez(i,j,k) = Ez(i,j,k)+((Ex(i,j,k)-Ex(i-1,j,k))*Dbdy)
      +(Ey(i,j-1,k)-Ey(i+1,j,k))*Dbdx)
    end do
  end do
end do
```

437.leslie3d is a FORTRAN code for Computational Fluid Dynamic in 3D space. It shares a lot in common with **GemsFDTD**. **leslied3d** source code is implemented in a single file named tml.f. The code segment located at line 1640 in the source file is identified as the hot section by our Pin tool.

Listing 5.11: critical code in Leslie3d

```
//tml.f
DO L=1,5
  DO K=K1,K2
    DO j=J1,J2
      DO I=I1,I2
       II   =  I + IADD
       IBD = II - IBDD
       ICD = II + IBDD

           QAV(I,J,K,L) = R6I *(2.0D0 * Q(IBD,J,K,L,N) +
                                   5.0D0 * Q( II,J,K,L,N) -
                                                    Q(ICD,J,K,L,N)
    END DO
  END DO
END DO
```

The differences between IBD, II and ICD are always one. Therefore, the piece of code is sequential access with good locality. The Pin tool identifies that the code accessing behavior is fairly regular, as over 95% accesses are of a stride of 8 bytes. The access range is approximately 12.4MB, thus the last level cache cannot cover the entire array. Therefore, the memory bandwidth consumption will contain a great proportion coming from hardware prefetch and main memory access due to the last level cache misses.

A similar behavior is also shown in block_solver.f in **bwaves**'s source code. **bwaves** is a floating-point benchmark which is doing Computational Fluid Dynamics in 3D space. The calculation involves nine arrays with good locality inside a five-level nested loop. Therefore, it is reasonable that **bwaves**, **leslie3d** and **GemsFDTD** are categorized into the same cluster. Note that **GemsFDTD, leslie3d, bwaves** are all written in FORTRAN, which takes column-major array layout. We find those codes have good locality by examining the loop structure of the source code.

Listing 5.12: critical code in bwaves

```
//block_solver.f
do k=1,nz
   km1=mod(k+nz-2,nz)+1
   kp1=mod(k,nz)+1
   do j=1,ny
          jm1=mod(j+ny-2,ny)+1
      jp1=mod(j,ny)+1
      do i=1,nx
         im1=mod(i+nx-2,nx)+1
         ip1=mod(i,nx)+1
         do l=1,nb
            y(l,i,j,k)=0.0D0
            do m=1,nb
               y(l,i,j,k)=y(l,i,j,k)+
               axp(l,m,i,j,k)*x(m,ip1,j,k)+
                    ayp(l,m,i,j,k)*x(m,i,jp1,k)+
                    azp(l,m,i,j,k)*x(m,i,j,kp1)+
                    axm(l,m,i,j,k)*x(m,im1,j,k)+
                    aym(l,m,i,j,k)*x(m,i,jm1,k)+
                    azm(l,m,i,j,k)*x(m,i,j,k1)
                 end do
              end do
           end do
        end do
end do
```

Cluster five has 14 programs and previous experiments show that those programs have less contention power compared with others. We examine several programs using Pin, trying to explain why they are “cache-polite” ones.

416.gamess is a floating-point benchmark program doing quantum chemical computation. It is a sequential version of the parallel program GAMESS, which is written in FORTRAN programming language. There are three reference inputs and we feed them into the Pin tool and identify similar hot sections/instructions. Line 3433 in int2a.fppized.f generates one of the hot instructions. The code is calculating the index of an array GHONDO and assigning the corresponding position with value 0. The access range is 10360 bytes and the access pattern is random, but the difference is always a multiple of 8-bytes. From this information we can imply that each element in that array is 8-bytes long and even though the access pattern is random, the size of the array can fit into the L1/L2 private cache and will result in lower last level cache miss ratio. Due to the random access pattern, the hardware prefetcher cannot capture any fixed pattern, thus the program will have relatively low memory bandwidth consumption.

Listing 5.13: critical code in gamess

```
//int2a.fppized.f
      DO 260  I = MINI,MAXI
         IF  (IANDJ) JMAX = I
         DO 240  J = MINJ,JMAX
            IJN = IJN+1
            N1 = IJGT(IJN)
            LMAX = MAXL
            KLN = 0
            DO 220  K =  MINK,MAXK
               IF  (KANDL) LMAX = K
               DO 200  L = MINL,LMAX
                  KLN = KLN+1
                  IF  (SAME .AND. KLN .GT. IJN) GO TO 240
                  NN = N1+KLGT(KLN)
                  GHONDO(NN)  = ZERO//3433
  200          CONTINUE
  220       CONTINUE
  240    CONTINUE
  260 CONTINUE
```

454.calculix is a finite element code for linear and nonlinear 3D structural applications written in a mixture of C and FORTRAN programming languages.

Listing 5.14: critical code in calculix

```
...
//Utilities_DV.c
for(i=0;i<n;i++){
        register double r0i,r1i,r2i,c0i,c1i,c2i;

    r0i = row0[i]; r1i = row1[i]; r2i = row2[i]; //1245
    c0i = col0[i]; c1i = col1[i]; c2i = col2[i];
    s00 += r0i*c0i; s01 +=  r0i*c1i; s02 += r0i*c2i;
    s10 += r1i*c0i; s11 +=  r1i*c1i; s12 += r1i*c2i;
    s20 += r2i*c0i; s21 +=  r2i*c1i; s22 += r2i*c2i;
}
...
```

The miss ratio corresponding to Listing 5.14 is below 1% due to good locality. However, the absolute count of last level cache misses stands out compare to other sections because the code has been executed quite frequently during the execution of the program. It involves a significant portion of scaler operations and this may result in high IPC for this particular benchmark program. The access range is around 106MB. However, each tight memory chunk is only 4792 bytes long, and over 97% of its execution is a stride pattern equal to 8-bytes. This is consistent with the fact that row0, row1, row2, col0, col1, col2 are defined as double type arrays.

464.h264ref is C implementation of H.264 protocol, which is the compression standard for Advanced Video Coding (AVC). The program itself exerts little pressure to co-run peers, and the PMU readings show that this program has lower value in both memory bandwidth consumption and last level cache misses. The entries for instruction trigger most last level cache misses obtained from Pin have very small numbers, suggesting that the program either has good locality or is a

CPU-bound application rather than a memory-bound one. Thus we only examine instructions that trigger most last level cache hits, as these instructions also contribute to most last cache misses due to a large execution count base. These instructions are at lines 419, 985, and 1093 in mv-search.c.

Listing 5.15: critical code in h264ref

```
...
//line 419 in mv-search.c
refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
...
for(pos=0; pos<max_pos; pos++, block_sad++)
{
        if(*block_sad < min_mcost) //line 985
    {
        ...
    }


}
...
for(dd=d[k=0];k<16;dd=d[++k])//line 1093
{
        satd ==(dd < 0 ? -dd : dd);
}
```

PelYline_11 is a function pointer pointing to 2 functions FastLine16Y_11 and UMVLine16Y_11, which are defined in file refbuf.c. The assembly corresponding to line 419 is $callq * 0x240d76(\%rip)$, and the stride difference only has two values of equal count number, suggesting that both functions are evenly called throughout the execution. The remaining two hot instructions are of a regular 4-byte-long stride access pattern (over 99% execution) with an overall accessing range of less than 1MB. This is consistent with syntax as block_sad and d are integer type arrays where each element in the array is 4-bytes long.

458.sjeng is an AI program that plays chess using game tree search and pattern recognition techniques. The program only has one reference input. We feed it into PIN and obtain the following results. Line 500 in neval.c is identified as a hot instruction and evalRoutines[] is a pointer pointing to different functions. piecet(i) is the parameter evalRoutines accepts to determine whether the current piece is a king, queen, rook, bishop, knight or a pawn. The access range is actually scattered and the access pattern is random. The rest of the hot instructions in the benchmark are accessing memory chunks less than 100MB with a random access pattern. Therefore, compared to **gamess**, it has higher last level cache miss ratio, and since there is no stride access pattern, the prefetcher won't work either, therefore, the memory bandwidth consumption is higher than **gamess** due to higher last level cache misses.

Listing 5.16: critical code in sjeng

```
//neval.c
  for (j = 1, a = 1; (a <= piece_count); j++) {
    i = pieces[j];//494
    if (!i)
      continue;
    else
      a++;
    score += (*(evalRoutines[piecet(i)]))(i,pieceside(i));//500
...
```

445.gobmk is another AI program playing the game "GO". The Pin tool identifies lines 2172 through 2174 in file engine/board.c as hot instructions.

Listing 5.17: critical code in gobmk

```
//board.c
int
chainlinks2(int str, int adj[MAXCHAIN], int lib)
{
  struct string_data *s, *t;
  int k;
  int neighbors;

  ASSERT1(IS_STONE(board[str]), str);

  /* We already have the list ready, just copy the strings with the
   * right number of liberties.
   */
  neighbors = 0;
  s = &string[string_number[str]];
  for (k = 0; k < s->neighbors; k++) {   //2172
    t = &string[s->neighborlist[k]];      //2173
    if (t->liberties == lib)              //2174
      adj[neighbors++] = t->origin;
  }
  return neighbors;
}
```

The assembly code corresponding to these three lines of code are as follows.

```
        line 2172: cmp     %eax,0x60(%rsi)
        line 2173: movslq  0x64(%rsi,%rcx,4),%rax
        line 2174: cmp     %ebx,0xc(%rax)
```

The access range of these three instructions are 43896 bytes, 43904 bytes, and 46128 bytes, respectively. The indirect accessing behavior makes the access pattern a random one. To be specific, line 2173's access pattern is random, but with a multiple of 4-bytes, this is consistent with the assembly code movslq $0x64(\%rsi, \%rcx, 4), \%rax$, where the offset is $\%rcx$ multiplied by 4. Line 2174 is a random but with a multiple of 744-bytes, suggesting that *string_data* type *t* is a structure 744 bytes long once allocated. The array size of **gobmk** is small and the accessing pattern is random, making the program run with low memory bandwidth consumption and last level cache miss ratio, which is consistent with our PMU result.

482.sphinx3 is a C implementation of a speech recognition system. Hot sections are identified at lines 653 to 654 in cont_mgau.c and line 523 in vector.c file.

Listing 5.18: critical code in sphinx3

```
//cont_mgau.c
...
for(i=0; i<veclen; i++){
        diff1 = x[i] - m1[i];
    dval1 -= diff1 *diff1 * v1[i];
}
...
```

Listing 5.19: critical code in sphinx3

```
//vector.c
...
for(i=0; i<veclen; i++){
          diff1 = x[i] - m1[i];
     dval1 -= diff1 *diff1 * v1[i];
     diff2 = x[i] - m2[i];
     dval2 -= diff1 *diff1 * v2[i];
}
...
```

Source code in Listing 5.18 and Listing 5.19 are very similar, array m1, m2 as well as v1, v2 are always in one element of difference within each loop iteration. Therefore, the second piece of code is an unrolled version of the first one with a factor of 2. Pin shows that the accessing pattern is stride equals 4-bytes long and the accessing range is 7.4MB and 1.2MB, respectively. Even though second code segment is an unrolled version of the first one, these two pieces of code do not operate on the same memory space. Due to the good locality and large working set size, the program will have significant memory bandwidth consumption but low last level cache misses, which is consistent with PMU statistics.

401.bzip2 is a compression program. It has six reference inputs. Though the memory footprints of those inputs range from 100MB to over 800MB, the Pin tool in general yields similar hot section/instruction results.

Listing 5.20: critical code in bzip2

```
 blocksort.c
```

```
for (; i >= 3; i -= 4) {
    quadrant[i] = 0;
    j = (j >> 8) | ( ((UInt16)block[i]) << 8);
    ftab[j]++;
    quadrant[i-1] = 0;//825
    j = (j >> 8) | ( ((UInt16)block[i-1]) << 8);
    ftab[j]++;
    quadrant[i-2] = 0;
    j = (j >> 8) | ( ((UInt16)block[i-2]) << 8);
    ftab[j]++;
    quadrant[i-3] = 0;
    j = (j >> 8) | ( ((UInt16)block[i-3]) << 8);
    ftab[j]++;
 }
```

Line 825 in blocksort.c is identified as hot instruction. The difference between the maximum memory address and the minimum memory address is 1.1GB. However, the program iterates multiple tight memory chunks and this results in a much smaller memory accessing range. The accessing pattern is stride access equal to 8 bytes long. This is consistent with program implementations as the "quadrant" array is UInt16, which is a 2 bytes long data type and the implementation in Listing 5.20 is a loop unrolling with a factor of 4, therefore, the stride is 2*4 = 8 bytes.

Listing 5.21: critical code in bzip2

```
blocksort.c
  for (; i >= 3; i -= 4) {
        s = (s >> 8) | (block[i] << 8);
        j = ftab[s] -1;
        ftab[s] = j;
        ptr[j] = i;
        s = (s >> 8) | (block[i-1] << 8);
        j = ftab[s] -1;
        ftab[s] = j;
        ptr[j] = i-1;//862          UInt32* ptr,
        s = (s >> 8) | (block[i-2] << 8);
        j = ftab[s] -1;
        ftab[s] = j;
        ptr[j] = i-2;//866          UInt32* ftabs
        s = (s >> 8) | (block[i-3] << 8);
        j = ftab[s] -1;
        ftab[s] = j;
        ptr[j] = i-3;
    }
```

Lines 862, 866 are similar code sections identified by the Pin tool. Our Pin tool finds that the tight memory chunk size is 0.3MB. The accessing pattern is random access. In the implementation of **bzip2**, the program divides the memory into blocks. There are 256 blocks, each block has 256 elements, and each element (ptr array) is a UInt32, which is a 4 bytes long data type. Therefore, the total accessing range for a tight memory chunk is 256*256*4 = 262144 bytes, which is close to 0.3MB.

473.astar is an AI game for path finding. It has two reference inputs: BigLakes and rivers.

Listing 5.22: critical code in astar

```
RegMng.cpp
i32 regmngobj::getregfillnum()
{
   i32 i;


   regfillnum++;


   if (regfillnum==1024*1024*1024)
      regfillnum=1;


   for (i=0; i<rarp.elemqu; i++)
      rarp[i]->fillnum=0;//490


   return regfillnum;
}
```

Line 490 in the file RegMng.cpp is identified as a hot instruction. The corresponding assembly code is $movl\$0x0, 0x20(\%rax)$. The code stores value 0 to the corresponding memory locations. $0x20$ is the offset of the field “fillnum” and $\%rax$ holds the memory location of rarp[i]. The accessing range is 245MB and the accessing pattern is random with a multiple of 112 bytes. Over 30% of random access is of a multiple of 2192 bytes, suggesting the program iterates an array of a structure that is 2192 bytes long. This pattern is identified for both reference inputs, which suggests that 2192 is a fixed program-dependent allocation size.

Similar code is identified at line 8 in $RegWay\_.cpp$. The corresponding assembly loads the value in memory location $0x20(\%rdx)$ to a register $\%eax$. The accessing range is also 245MB with the same minimum and maximum memory addresses and the access pattern is random with a multiple of 2192 bytes. This suggests the object regionp is identical to rarp[i] in the previous code example.

Listing 5.23: critical code in astar

```
//RegWay_.cpp
bool regwayobj::isaddtobound(regobjpt initialregionp,
regobjpt regionp)//line 8
{
  if (regionp->fillnum==regfillnum)
    return false;

  // insert additional game logic here

  return true;
}
```

astar is calculating distance between a source point to a target point in a 2D grid. For each point it accesses, the program examines all 8 neighbors surrounding it. Related code is identified by our Pin tool such as line 187 in file *RegBounds_.cpp*. The assembly code is *cmpq* $\$0x0$, $(\%rdx,\ \%rax, 8)$, which compares the value in a memory location with 0. The offset is %rax multiply 8, suggesting the stride is 8 bytes long. The Pin tool identifies a total size of 580MB accessing range and a mixture of stride and random patterns with a multiple of 8 bytes (over 60% of overall execution), and a random pattern of 16368 bytes (over 20% of overall execution). The program actually runs on a grid map of 2048*2048 points, where each row contains 2048 points and each point is represented as a structure of 8 bytes long. The difference between a point and its neighbors located at the previous row is roughly 2048*8 = 16384, which confirms the Pin result.

Listing 5.24: critical code in astar

```
//RegBounds_.cpp
void regboundobj::makebound2(boundart& b1arp, boundart& b2arp)
{
  i32 j;
  i32 x,y;
  i32 x1,y1,x2,y2;

  b2arp.clear();

  for (j=0; j<b1arp.elemqu; j++)
  {
    x1=b1arp[j].x-1;
    y1=b1arp[j].y-1;

    x2=b1arp[j].x+1;
    y2=b1arp[j].y+1;

    if (x1<0) x1=0;
    if (y1<0) y1=0;

    if (x2>mapmaxx) x2=mapmaxx;
    if (y2>mapmaxy) y2=mapmaxy;
```

```
      for (y=y1; y<=y2; y++)
        for (x=x1; x<=x2; x++)
          if (regmapp(x,y)==nil)//187
            addtobound(b2arp,x,y);
  }
}
```

The accessing range of **astar** is greater than the cache size (8MB) and there is a mixture of stride and random accessing patterns. This makes **astar** run with higher memory bandwidth consumption and last level cache misses.

403.gcc is a program derived from GCC version 3.2. It runs a compiler and analyzes source code inputs. There are 9 reference inputs. The code segment in Listing 5.25 (line 175 in file sbitmap.c) is identified by the Pin tool as a hot section. sbitmap is a data type supported in gcc to represent sets. sbitmap_union_of_diff function is built to manipulate those sets.

Listing 5.25: critical code in gcc

```
//sbitmap.c
int       sbitmap_union_of_diff (dst, a, b, c)
     sbitmap dst, a, b, c;
{
  unsigned int i;
  sbitmap_ptr dstp, ap, bp, cp;
  int changed = 0;
  for (dstp = dst->elms, ap = a->elms, bp = b->elms, cp = c->elms, i = 0;
       i < dst->size; i++, dstp++)
    {
      SBITMAP_ELT_TYPE tmp = *ap++ | (*bp++ & ~*cp++);//sbitmap175

      if (*dstp != tmp)//177
        {
          changed = 1;
          *dstp = tmp;
        }
    }
  return changed;
}
```

The accessing range is scattered, which suggests that there are multiple tight memory chunks. The Pin tool suggests each tight memory chunk is only 1032 bytes long and the access pattern is 8-byte stride for the most of the execution time.

483.xalancbmk is a XSLT processor for transforming XML documents into HTML, text, or other XML document types. It extensively makes use of C++'s standard template library algorithms, as lines 208, 212, 216 and 220 of the stl_algo.h file are identified as hot section by our Pin tool. The accessing pattern follows a stride of 32 bytes, which is consistent with the syntax, as the program tries to find a value in an array in an unrolled manner. There are four look-ups executed in each iteration and each element is 8 bytes long. Therefore, the next access is 8*4, which is 32 bytes in difference. The piece of code operates on a memory space of 346MB. Based on the Pin tool result, each tight memory chunk is at most 32160 bytes long.

Listing 5.26: critical code in Xalancbmk

```
//file stl_algo.h
...
template <class _RandomAccessIter, class _Tp>
_RandomAccessIter find(_RandomAccessIter __first,
                       _RandomAccessIter __last,
                       const _Tp& __val,
                       random_access_iterator_tag)
{
  typename iterator_traits<_RandomAccessIter>::difference_type
  __trip_count = (__last - __first) >> 2;

  for ( ; __trip_count > 0 ; --__trip_count) {
    if (*__first == __val) return __first;//208
    ++__first;

    if (*__first == __val) return __first;//212
    ++__first;

    if (*__first == __val) return __first;//216
    ++__first;

    if (*__first == __val) return __first;//220
    ++__first;
  }

  switch(__last - __first) {
  case 3:
    if (*__first == __val) return __first;
    ++__first;
  case 2:
    if (*__first == __val) return __first;
    ++__first;
  case 1:
    if (*__first == __val) return __first;
    ++__first;
  case 0:
  default:
    return __last;
  }
}
...
```

Moreover, as the program is doing string operations for the most of the time, a function "StringLen" is called multiple times at line 1584 of the file XMLString.hpp. The access pattern is a stride of 2 bytes, which is consistent with the syntax as the program keeps accessing a memory chunk at the pace of a Wide-Char, which is 2 bytes long, until it hits a null to obtain the string length. The piece of code operates on a memory space of 325MB. Based on the Pin tool result, both the "find" function and "StringLen" function in stl_algo.h work on the same memory chunks, the latter one accesses 21MB less memory.

Listing 5.27: critical code in Xalancbmk

```
//file XMLString.hpp
...
inline unsigned int XMLString::stringLen(const XMLCh* const src)
{
        if (src == 0 || *src == 0)
    {
        return 0;
    }
     else
    {
         const XMLCh* pszTmp = src + 1;

         while (*pszTmp) //1584
             ++pszTmp;

         return (unsigned int)(pszTmp - src);
     }
 }
...
```

447.dealII is a C++ program that solves partial differential equations using the adaptive finite element method. The hot sections triggering the most last level cache hits or misses are located at line 353 of sparse_matrix.templates.h, line 92 and line 116 of dof_constraints.cc and line 515 of mapping_q1.cc.

Listing 5.28: critical code in dealII

```
//file sparse_matrix.template.h
...
for(unsigned int row=0; r<n_rows; ++row)
{
        typename OutVector::value_type s = 0.;
    const number *const val_end_of_row = &val[cols->rowstart[row+1]];
    while(val_ptr != val_end_of_row)
        s += *val_ptr++ * src(*column_ptr++) //line 353
     *dst_ptr ++ =s;
}
...
```

It is clear that the code in Listing 5.28 is doing vector multiplication for a sparse matrix, as a similar pattern is also seen in the benchmark **soplex**. In line 353, there are three memory accesses, which are val_ptr, column_ptr and src(*column_ptr). The accessing patterns for the first two are fairly regular, which are 4-bytes and 8-bytes long, and accessing range is scattered. By dividing the number of stride accesses that are is to either 4 bytes or 8 bytes by the number of continuous segments, we find that the maximum tight memory chunk is 22MB and 44MB, respectively. The access of src array is indirect an access through column_ptr and the pattern is a mixture of stride and random but at a multiple of 8 bytes, suggesting the element size is 8 bytes.

Line 92 in dof_constraints.cc files is also identified as a hot instruction with frequent last level cache hits, where an array is accessed sequentially and each element is compared with the value in a register. The access pattern is a regular stride access of 32 bytes, suggesting each element in the “lines” array is a structure of 32 bytes long.

Listing 5.29: critical code in dealII

```
//dof_constraints.cc
...
for(unsigned int i=0;i!=lines.size();++i)
{
        if(lines[i].line == line)
        return;
}
...
```

The third hot section is located at line 515 in mapping_q1.cc. A significant portion of the **dealII** is executing this piece of code with only a few last level cache misses. The Pin tool shows that the accessing range is only 6864 bytes long and most access strides are equal to 0, suggesting that the program reuses array elements frequently. The array index corresponding to the innermost loop is j, therefore, the value for $data.mapping\_support\_points[k][i]$ will not change for that loop.

Listing 5.30: critical code in dealII

```
//mapping_q1.cc
...
for(unsigned int point=0;point<n_q_points; ++point)
  for(unsigned k=0; k<data.n_shape_functions; ++k)
    for(unsigned int i=0; i<dim; i++)
      for(unsigned int j=0; j<dim; j++)
        data.contravariant[point][i][j]
        += (data.derivative(point+data_set,k)[j]
            *
            data.mapping_support_points[k][i] );//line 515
...
```

With accessing patterns gathered by the Pin tool, we could design program-specific bubbles as follows. In the very beginning, arrays or other data structures such as tree are declared with program specific sizes. Array elements or tree nodes are initialized with random values instead of 0. The value does not have to be exactly the same or even close to the actual program, what we really care about is the access pattern. We copy all hot sections into a while loop so that the kernel will run long enough to cope with co-run profiling. We make sure the ratio between stride access and random access matches the actual application as well as the stride value by manipulating the loop carried variables in a for loop. We find that by integrating sections that contribute 75% overall cache misses among top 50 cache misses instructions and partial instructions collected from top 50 hit will yield a bubble whose PMU reading matches the actual program. Occasionally the bubble will exert a higher pressure as it may have higher bandwidth consumption and cache miss ratio, and we can adjust the ratio between most misses part and most hits part, and make sure the PMU reading between the actual application and its corresponding bubble are as close as possible. Certainly, the bubble with a lower discrepancy in PMU statistics is expected to have better representativeness. In this dissertation, not every SPEC CPU2006 benchmark has its own bubble. We don't create bubbles for **omnetpp** and **xalancbmk**. The hot sections identified by our Pin tool for these programs happen to be function calls in the source code, in addition, top misses instructions are too scattered to form a tight bubble. Thus further study needs to be done for such cases, but due to the time limits, we leave it to future work.

5.5.1 Prediction using bubbles alone

With new bubbles developed, we replace the actual programs in a co-run group with bubbles and run them together to predict slowdown by examining the IPS changes and we compare prediction results with the execution times when actual programs co-run together.

As mentioned above, a significant number of code segments has been pruned out and only those last level cache sensitive instructions are preserved in our bubbles. The new kernel should perform exactly the same as the actual program in terms of cache access behavior, but there is no guarantee that other metrics such as IPC match exactly the same. Since we predict by observing IPC changes of the bubble, it is likely that errors could be introduced. Therefore, in this section, we use the actual co-run execution time of a program in a group divided by its solo run time as the baseline, and compare both prediction results using (1) purely bubble kernels and (2) actual program slowdown time when it co-runs with bubbles which represent the rest of members in a co-run group divided by the program solo run time. In Table 5.3, we show the actual programs' name in each row, and the first one in each group serves as the program of interest. For each column, we will use bubbles, actual+bubbles to represent prediction using (1) and prediction using (2) respectively. The baseline is the scaled co-run execution time of 'program of interest' versus its solo-run execution time.

In Table 5.3, the programs of interest are **libquantum** and **lbm**, respectively. As we use different bubbles to represent actual programs, each of them could bring in error. For simplicity, other than the program of interest, we use $n = 1, 2, 3$ copies of a single program for the co-run peers in the group. We co-run **libquantum bubble** with other bubbles. On one hand, when actual **libquantum** co-run with other bubbles, its slowdown matches the slowdown when we replace all bubbles by the actual programs (baseline case) within 2-3%. This demonstrates that bubbles can closely imitate programs in the group. One the other hand, prediction results obtained by examining bubble's IPC changes also match the baseline, with maximum error at 5.67%, showing the capability of make

Table 5.3: Prediction using bubbles only vs program+bubbles

	Relative prediction error(%)	
Program A + co-run peers	**Bubble only**	**Program A + bubbles**
lib + 1x lib	0.65	0.82
lib + 2x lib	0.87	1.01
lib + 3x lib	1.57	2.29
lib + 1x lbm	5.67	1.95
lib + 2x lbm	3.01	2.41
lib + 3x lbm	2.82	1.79
lib + 3x astar	1.26	0.89
lib + 3x bzip2	1.61	1.66
lib + 3x gcc	2.61	0.22
lib + astar + bzip2	1.72	0.91
lbm + 1x lib	7.59	2.27
lbm + 2x lib	4.58	1.32
lbm + 3x lib	4.10	1.01
lbm + 1x lbm	2.20	2.46
lbm + 2x lbm	4.03	2.04
lbm + 3x lbm	1.80	1.79

contention prediction using pure bubbles alone. The case for **lbm** also yields accurate prediction results. The maximum prediction error is 7.59% using pure bubbles as opposed to 2.27% using a mixture of actual **lbm** and other bubbles in the group. We find that compared to **lbm**, the bubble for **libquantum** is of better match in terms of IPC changes. We examine more programs in the Table 5.4.

As shown in Table 5.4, the prediction accuracy is still within 2-3% using a mixture of an actual program and several bubbles together. However, predictions made by monitoring the bubble's IPC perform poorly for some cases shown in Table 5.4. The maximum prediction error is as high as 17% compared to the baseline result. It shows that the bubbles, as the co-run peers, can provide similar contention power as actual programs do, but their IPC changes are not accurate.

5.5.2 Prediction with a mixture of bubbles and an actual program

As is shown in Table 5.3 and Table 5.4, predictions using pure bubbles can cause significant error whereas prediction using an actual application with bubbles together has consistently high prediction

Table 5.4: Prediction using bubbles only vs program+bubbles (continue)

	Relative prediction error(%)	
Program A + co-run peers	**Bubbles only**	**Program A + bubbles**
astar + 1x lib	3.17	2.70
astar + 2x lib	1.99	2.15
astar + 3x lib	0.60	0.71
astar + 1x lbm	3.05	3.29
astar + 2x lbm	0	0.91
astar + 3x lbm	2.48	2.66
bzip2 + 1x lib	6.50	2.68
bzip2 + 2x lib	16.46	3.60
bzip2 + 3x lib	10.93	2.48
bzip2 + 1x lbm	11.02	2.36
bzip2 + 2x lbm	4.35	2.86
bzip2 + 3x lbm	12.99	3.39
mcf + 1x lib	3.82	1.65
mcf + 2x lib	9.51	1.64
mcf + 3x lib	3.60	1.60
mcf + 3x astar	12.70	0.50
mcf + 3x bzip2	17.60	2.36
soplex + 1x lbm	4.09	1.25
soplex + 2x lbm	2.23	0.89
soplex + 3x lbm	3.18	1.55
soplex + 1x lib	9.90	2.77
soplex + 2x lib	5.50	0.31
soplex + 3x lib	2.60	0.79

accuracy. Therefore, we adopt the second prediction scheme and show detailed prediction results in this section.

Table 5.5 shows the relative prediction errors for SPEC CPU2006 running with four different co-run groups. The first co-run group includes **libquantum** and **lbm** which come from cluster one and cluster four respectively. Note that we divide the SPEC CPU2006 into five clusters. The average for cluster one, whose members are **libquantm** and **milc**, is 3.88% and **milc** has the highest prediction error which is 6.54%. Cluster two has programs **gcc**, **omnetpp**, **xalancbmk**, **zeusmp** and **dealII** and the average is 1.55%. Cluster three has programs **mcf**,**bwaves**,**leslie3d** and **GemsFDTD**, the average is 4.17%. Cluster four has **soplex** and **lbm**, its average prediction error is 1.41%. The rest of the programs are from cluster five and the average is 0.79%.

We increase the number of co-run peers by adding **bzip2** into the first co-run group to form a four-core co-run scenario. The average prediction error for all five clusters are below 2%. The highest prediction error also comes from program **milc**, which is 3.36%. We also form a four-core group by co-running SPEC CPU2006 with three **lbm**s. The average prediction error is below 2% and the highest prediction error is 3.88% from **GemsFDTD**.

We choose **soplex** from cluster four and use two instances to co-run with SPEC CPU2006. The average prediction errors for each cluster are 3.33%, 2.65%, 1.59%, 1.59% and 1.18%, respectively. The highest prediction error comes from program **gcc**, which is 5.70%.

Table 5.6 shows the result for SPEC CPU2006 co-run with two **gcc**s. The reference run of actual **gcc** has nine inputs thus it has roughly nine phases. Ideally, we should create a bubble that represents these nine phases. However, we only use one critical code segment to create the micro-kernel as its PMU statistics match the average statistics of actual **gcc** program. It turns out that the bubble can represent the program very well. The average prediction error for five clusters are all below 1.41%. The highest prediction error comes from **sphinx3**, which is 4.63%.

We further examine how the number of program-specific bubbles will affect prediction ability. In Table 5.7, a subset of programs **libquantum, zeusmp, astar, xalancbmk** are co-run with one gcc bubble, two gcc bubbles and three gcc bubbles, respectively. As for the old bubbles, whose kernel array ranges from 1 to 10MB, the error will be accumulated as the number of bubbles increases, our new bubble won't suffer from the issue as it can change its pressure as the actual program does when co-run peers accumulate.

We create a bubble for **mcf**, whose working set is the largest among all SPEC CPU2006 benchmark programs, which is approximately 1.75GB. As discussed above, the critical code segment in this benchmark is a tree traversal associated with node and arc structures. The connectivity of nodes and arcs are dynamically changed. In our bubble implementation, we preserve the post-order tree traversal, and we adjust the tree structure at initialization stage by tuning the proportion between the children and the siblings while keeping the total number of tree nodes fixed. We verify the capability of this bubble by co-running two **mcf** instances with SPEC CPU2006 together. The result is shown in Table 5.8. The average prediction errors are 1.26%, 3.73%, 0.86%, 3.00% and 1.05% for five clusters respectively. The highest prediction error comes from program **xalancbmk**, which is 6.89%.

Considering a very interesting phenomenon, our previous bubble construction follows the bubble-up methodology. We create stream or random access bubble whose array size ranges from 1 to 10 MB. Suppose for a two-core co-run case, we obtain **mcf**'s pressure score which equals 5MB. Experimental result suggests that in a three-core co-run scenario, where, to be more specific, two instances of **mcf** co-run with other programs, the **mcf**'s pressure is decreasing. This means that we should use the same bubble but decrease its array size. The PMU reading of the solo-run bubble doesn't match the actual program and it converges with the actual program in a two-core co-run case and once again diverges in three-core co-run cases. However, in this new bubble design, the bubble's PMU statistics matches the actual program in all cases. For example, **libquantum** finishes in 985s when it co-runs with **mcf** and the prediction using bubble is 987s. **libquantum** finishes in 1210 seconds while co-run with two **mcf**, while the bubble result predicts as 1200 seconds. **zeusmp** finishes in 726s when it co-runs with **mcf** and bubble predicts as 734s, while **zeusmp** finishes in 814s when it co-runs with two **mcf**, the bubble result predicts as 823s. These results suggest no adjustment should be made from two cores to three cores and the bubble acts exactly the same as the actual program, as it ‘stretches’ as the real **mcf** does, which means the pressure changes accordingly.

Table 5.9 shows the prediction results for SPEC CPU2006 when these programs co-run with two instances of **bzip2** and three instances of **bzip2** respectively. The average prediction error for both groups are below 1.5%. The largest prediction error is 4.62%, in which case, an instance of **dealII** co-runs with two **bzip2**.

Table 5.10 shows the prediction results for SPEC CPU2006 when these programs co-run with two instances of **astar** and three instances of **astar** respectively. The average prediction error for both groups are below 1.50%. We can observe an trend that whenever number of **astar** increases, the prediction error accumulates, but within a fairly small range. And most prediction error comes from **soplex** and **lbm**. Those two programs come from cluster four, whose pressure power is the highest, mostly from memory bandwidth consumption. Our **astar** bubble's PMU statistic is lower than actual one. Therefore, it exerts fewer contention power and that's why our predicted slowdown is lower than actual one, which brings in errors.

Table 5.11 shows the prediction results for SPEC CPU2006 co-run with **astar+gcc** and **mcf+gcc** respectively. The prediction error are below 1.5% for both groups, suggesting that the simulation of **gcc**, **mcf** as well as **astar** are fairly accurate. Similar result also show in Table 5.12, where SPEC CPU2006 co-run with **sphinx3+dealII**.

Table 5.13 shows the prediction results for SPEC CPU2006 co-run with two instances of **sphinx3** and **sphinx3+namd** respectively. They are all three-core co-run cases. The standard deviation is small as most program's prediction error is close to average. However, we can observe that **gcc**, **xalan**, **mcf** all have relatively larger prediction error. Both **sphinx3** and **namd** are from cluster five, whose contention power is small as programs in that group all have small memory bandwidth consumption and last level cache misses. The PMU statistic for **sphinx3** bubble solo execution has relative large gap with actual one compared to others. As it's memory bandwidth consumption is higher than the actual one. Therefore, for those memory-bounded applications such as **mcf**, **xalancbmk**, etc., they experience much more pressure than the actual **sphinx3** can exhibit. Therefore the prediction gives higher performance slowdown values.

Table 5.14 shows prediction result for another three-core co-run case. Both program **leslie3d** and **GemsFDTD** comes from cluster three. We co-run them with SPEC CPU2006 benchmarks. The prediction errors are 2.14%, 1.50%, 3.24%, 1.39% and 1.27% for the five clusters, respectively, and the highest prediction error comes from **leslie3d**, which is 5.98%.

Table 5.15 shows two groups of four-core co-run prediction result. SPEC CPU2006 is co-run with **libquantum, gcc, namd**, which are from cluster one, two, five, **bwaves, sjeng, soplex**, which are from cluster three, five, four, respectively. The average prediction errors are below 2%. We examine the results cluster by cluster and find that the prediction for cluster four are 3.76% and 4.94% respectively. The highest prediction error comes from **xalancbmk**, which is 7.58%. We observe that the highest prediction across various co-run groups are from **milc, gcc, lbm, xalancbmk**. This is also true in our two-core cross-architecture settings, which suggest that it is more difficult to make accurate prediction for those programs than others.

To sum up, the tables listed in this section show the experimental results for predicting application slowdown for more than two core co-run cases using program specific bubbles. The highest co-run performance prediction error is 7.58%, and the average prediction accuracy is around 97%. Note all experimental results for more than two core co-run cases are only collected on a single machine due to time limits. We have not applied the cross-architecture model to this scenario. We argue it is not necessary to use the cross-architecture framework as we eliminate the sensitivity and pressure measurement processes. The newly designed bubble should behave similarly to the actual application even when it is executed on a machine with different hardware configurations. We assume that the hot sections which trigger most last level cache hits and misses should stay the same if no major changes have been made to the architecture on which the application is running. However, appropriate modification in our Pin tool needs to be made to match the actual architecture as newer cache replacement policies are recently proposed [50, 112, 57, 48] as opposed to the LRU policy which is implemented in our cache simulator. Therefore hot sections need to be re-collected and bubbles need to be re-created in order to make predictions on such architecture. We leave the verification on newer architectures to future work.

Table 5.5: Performance degradation prediction using specific bubbles for more than two cores co-run scenario

	Relative prediction error(%)			
Program	**lbm + lib**	**lbm + lib+bzip2**	**3x lbm**	**2x soplex**
perlbench	0.72	2.27	1.83	0.37
bzip2	1.96	2.78	1.74	0.11
gcc	1.15	1.22	2.61	5.70
mcf	5.61	0.79	2.56	1.47
gobmk	0.15	0.43	1.69	0.60
hmmer	0	0.97	0.95	0.49
sjeng	2.19	0.76	0.80	0.52
libquantum	1.22	0.49	1.79	3.01
h264ref	1.23	1.29	1.08	0.46
omnetpp	1.04	1.03	0.62	5.00
astar	0	2.30	3.48	3.06
xalan	1.89	1.87	2.79	0.21
Geometric mean	**1.43**	**1.35**	**1.83**	**1.24**
bwaves	4.34	0	1.22	2.76
gamess	0	1.71	0	0
milc	6.54	3.36	2.56	3.64
zeusmp	1.41	1.34	2.23	0.24
gromacs	0.85	0.83	1.11	0.56
cactusADM	0.53	2.24	0.49	1.68
leslie3d	2.07	0.65	1.02	5.27
namd	0.34	0.51	1.34	0.34
dealII	2.28	1.30	0.42	2.09
soplex	1.25	1.70	1.55	0.95
povray	1.05	1.37	0.68	0.69
calculix	0.55	0.27	0.52	0.28
GemsFDTD	4.66	0.61	3.88	1.91
tonto	1.98	0	0.84	0
lbm	1.57	0.63	0	2.21
sphinx3	1.23	0.52	0.82	0.71
Geometric mean	**2.04**	**1.07**	**1.17**	**1.46**

Table 5.6: Performance degradation prediction using gcc bubble for three-core co-run scenario

Program	**Relative prediction error(%) 2x gcc**
perlbench	0.20
bzip2	0.42
gcc	3.22
mcf	1.78
gobmk	1.29
hmmer	0
sjeng	0.57
libquantum	0.12
h264ref	0.73
omnetpp	1.45
astar	0.31
xalan	0.79
Geometric mean	**0.91**
bwaves	0.82
gamess	0
milc	0.96
zeusmp	0.67
gromacs	0.44
cactusADM	0.61
leslie3d	2.02
namd	0.17
dealII	0.39
soplex	0
povray	0.36
calculix	0.58
GemsFDTD	1.00
tonto	0.88
lbm	0.24
sphinx3	4.63
Geometric mean	**0.86**

Table 5.7: Performance degradation prediction using gcc bubble for 2/3/4 core co-run scenario

	Relative prediction error(%)		
Program	**1x gcc**	**2x gcc**	**3x gcc**
libquantum	0.27	0.12	0.22
zeusmp	0.71	0.67	0.64
astar	0.65	0.31	0.86
xalan	0.88	0.79	0.73

Table 5.8: Performance degradation prediction using mcf bubble for 3 core co-run scenario

Program	**Relative prediction error(%) 2x mcf**
perlbench	0.20
bzip2	6.12
gcc	4.83
mcf	0.86
gobmk	0.16
hmmer	0
sjeng	0.57
libquantum	0.83
h264ref	0.73
omnetpp	4.69
astar	1.25
xalan	6.89
Geometric mean	**2.26**
bwaves	0.81
gamess	0.90
milc	1.69
zeusmp	1.11
gromacs	0.15
cactusADM	0.62
leslie3d	0.74
namd	0.17
dealII	1.15
soplex	5.73
povray	0.72
calculix	0.58
GemsFDTD	1.02
tonto	1.00
lbm	0.27
sphinx3	2.56
Geometric mean	**1.20**

Table 5.9: Performance degradation prediction using bzip2 bubble for 3/4 core co-run scenario

	Relative prediction error(%)	
Program	**2x bzip2**	**3x bzip2**
perlbench	0.61	0.60
bzip2	3.17	1.11
gcc	0.45	1.85
mcf	0.39	2.36
gobmk	0.33	0.33
hmmer	0.99	0
sjeng	0.73	0.58
libquantum	0.91	1.66
h264ref	0.88	0.86
omnetpp	1.58	0.64
astar	1.60	0.94
xalan	1.16	1.09
Geometric mean	**1.07**	**1.00**
bwaves	0.85	2.50
gamess	0	0
milc	0.38	2.25
zeusmp	0.14	1.11
gromacs	0	0.30
cactusADM	1.26	0.62
leslie3d	1.09	2.33
namd	0.17	0
dealII	1.17	4.62
soplex	2.16	0.78
povray	0.36	0.36
calculix	0	0.58
GemsFDTD	1.59	1.42
tonto	1.54	3.83
lbm	0.67	0.72
sphinx3	0.49	0.23
Geometric mean	**0.70**	**1.35**

Table 5.10: Program performance degradation prediction using astar bubble for 3/4 core co-run scenario

	Relative prediction error(%)	
Program	**2x astar**	**3x astar**
perlbench	0.60	2.42
bzip2	1.43	0.55
gcc	2.18	1.26
mcf	0.38	0.50
gobmk	0.66	1.63
hmmer	0	0.98
sjeng	0.86	1.71
libquantum	1.36	0.89
h264ref	0	1.47
omnetpp	0	0.59
astar	0.48	2.20
xalan	1.68	3.20
Geometric mean	**0.72**	**1.46**
bwaves	1.68	1.60
gamess	0	0.90
milc	0.87	1.65
zeusmp	0.42	0.68
gromacs	0.15	0.29
cactusADM	0.62	1.23
leslie3d	1.84	1.15
namd	0	0.87
dealII	0.39	1.14
soplex	3.10	1.21
povray	0.36	1.79
calculix	0.58	1.16
GemsFDTD	1.41	0.93
tonto	0.26	0.25
lbm	1.78	3.42
sphinx3	1.48	0.89
Geometric mean	**0.97**	**1.20**

Table 5.11: Performance degradation prediction using program specific bubbles for three-core co-run scenario

	Relative prediction error(%)	
Program	**astar+gcc**	**mcf+gcc**
perlbench	0.20	0.40
bzip2	0.71	1.74
gcc	1.10	0.19
mcf	0.76	0.48
gobmk	0.33	0.63
hmmer	0	0.98
sjeng	0.29	0.14
libquantum	0.87	0.44
h264ref	0.25	0.59
omnetpp	0.64	0.88
astar	1.28	0.73
xalan	1.11	0.47
Geometric mean	**0.63**	**0.64**
bwaves	0	1.14
gamess	0	0.88
milc	1.08	1.51
zeusmp	1.67	0.66
gromacs	0.45	0.73
cactusADM	0.62	0.90
leslie3d	1.07	1.86
namd	0.17	0.69
dealII	0.39	0.74
soplex	1.32	0.65
povray	0.72	1.05
calculix	0	0.57
GemsFDTD	0.24	2.13
tonto	1.16	1.46
lbm	2.01	3.72
sphinx3	0.59	2.44
Geometric mean	**0.72**	**1.32**

Table 5.12: Performance degradation prediction using sphinx3 and dealII bubbles for 3 core co-run scenario

Program	**Relative prediction error(%) sphinx3 + dealII**
perlbench	2.24
bzip2	0.56
gcc	2.39
mcf	1.52
gobmk	1.82
hmmer	0
sjeng	1.30
libquantum	2.48
h264ref	1.72
omnetpp	1.29
astar	0.94
xalan	3.28
Geometric mean	**1.63**
bwaves	3.39
gamess	1.82
milc	3.33
zeusmp	0.98
gromacs	0.44
cactusADM	2.52
leslie3d	2.12
namd	0.17
dealII	2.31
soplex	1.06
povray	1.07
calculix	0
GemsFDTD	3.13
tonto	2.54
lbm	1.54
sphinx3	2.11
Geometric mean	**1.86**

Table 5.13: Performance degradation prediction using sphinx3 and namd bubbles for three-core co-run scenario

	Relative prediction error(%)	
Program	**2x sphinx3**	**sphinx3 + namd**
perlbench	0.79	2.67
bzip2	2.56	2.03
gcc	3.31	3.86
mcf	3.63	2.95
gobmk	1.64	2.36
hmmer	0.99	0
sjeng	1.12	0.29
libquantum	1.16	1.83
h264ref	0.96	0.99
omnetpp	1.89	3.08
astar	2.75	1.92
xalan	3.55	3.72
Geometric mean	**2.03**	**2.14**
bwaves	0.80	1.42
gamess	0.91	0.88
milc	0.58	2.40
zeusmp	2.35	3.09
gromacs	0.74	1.89
cactusADM	3.12	2.40
leslie3d	2.33	2.90
namd	0.52	0.86
dealII	2.12	3.61
soplex	3.45	2.33
povray	1.08	1.06
calculix	1.16	1.14
GemsFDTD	2.93	2.46
tonto	1.80	1.94
lbm	1.19	0.95
sphinx3	2.33	0.90
Geometric mean	**1.71**	**1.89**

Table 5.14: Performance degradation prediction using leslie3d and GemsFDTD bubbles for three-core co-run scenario

Program	**Relative prediction error(%) leslie3d + GemsFDTD**
perlbench	1.13
bzip2	1.83
gcc	0.65
mcf	1.83
gobmk	1.52
hmmer	0
sjeng	0.81
libquantum	2.06
h264ref	1.95
omnetpp	2.96
astar	2.32
xalan	0.81
Geometric mean	**1.49**
bwaves	2.32
gamess	0
milc	2.22
zeusmp	1.35
gromacs	1.01
cactusADM	1.11
leslie3d	5.98
namd	0.34
dealII	1.74
soplex	0.31
povray	0.35
calculix	1.98
GemsFDTD	2.83
tonto	1.80
lbm	2.46
sphinx3	2.86
Geometric mean	**1.79**

Table 5.15: Performance degradation prediction using specific bubbles for four-core co-run scenario

	Relative prediction error(%)	
Program	**lib+gcc+namd**	**bwaves+sjeng+soplex**
perlbench	1.28	1.95
bzip2	3.28	1.38
gcc	2.47	2.56
mcf	0.46	1.32
gobmk	0.91	1.73
hmmer	0	0
sjeng	2.06	1.97
libquantum	2.11	0.88
h264ref	0.80	0.29
omnetpp	0.44	1.43
astar	0.65	2.62
xalan	2.97	7.58
Geometric mean	**1.29**	**1.98**
bwaves	1.88	1.54
gamess	0.87	1.75
milc	1.31	3.23
zeusmp	0.69	1.91
gromacs	1.14	1.74
cactusADM	0.27	0.87
leslie3d	0	0.76
namd	0.85	1.03
dealII	1.29	1.38
soplex	1.32	2.82
povray	0.35	1.75
calculix	0	1.14
GemsFDTD	2.52	2.75
tonto	0.22	0.23
lbm	6.19	7.07
sphinx3	2.88	0.43
Geometric mean	**1.36**	**1.90**

Chapter 6

Conclusion

From the perspective of resource utilization and power consumption, one should schedule as many workloads as possible onto available cores. However, competition for shared resources among those co-running applications results in performance degradation. A smart scheduling scheme not only keeps the system from low utilization but also guarantees applications' QoS requirement. This dissertation proposes a framework that provides cloud providers and system administrators a way to predict the performance degradation of co-run application groups. We show the promising solutions by examining techniques employed by our framework through extensive experiments. This chapter summarizes our contributions and discusses future work.

6.1 Contributions

We propose a framework that maps the sensitivity and pressure of a program from a source machine to a target machine. Using sensitivity and pressure prediction for each program on target architecture, we manage to predict two-core co-run performance degradation within an average error of 2%. We refine the cross-architecture contention prediction approach by clustering SPEC CPU2006 benchmark programs and maintaining models for each group. By using different reporters to represent programs rather than using a unique one, we make the imitation even closer as a program's pressure will change accordingly towards peers with different pressure. With these refinements, programs with over 10% prediction error previously are decreased to 2%.

Compared to building bubble programs from scratch, we propose an off-line profiling based approach for constructing bubble programs. The constructed bubbles are program-specific and the memory accessing behavior and pressure of the bubbles will change accordingly whenever a co-run peer changes, which resembles the behavior of actual programs. This eliminates the bubble score profiling step because of that property. We expand the contention prediction from the pair-wise case to more than two core cases and find that the prediction error remains within 2-3%.

6.2 Future work

Our future interests include automating the program-specific bubble creation process by generalizing an application's access behavior into one or more candidate types, such as random access, stream access, indirect access or tree traversal, and parameterizing the associated access range, stride value, random/sequential access ratio, etc. Machine learning techniques such as regression tree might be employed as one can first categorize new programs into candidate kernels with their PMU statistics as inputs to a decision tree. Then at each leaf node, a regression model associated with that particular candidate kernel is used to predict those bubble specific parameters. To increase bubble representation accuracy compared with actual applications, we might also consider factors other than memory related behavior, such as branch prediction and network IO. Recently, newer replacement policies such as RRIP, SDB, SHiP HAWKEYE are proposed as an alternative to the LRU policy. Such changes might affect the critical code segment selection. Therefore, we are interested in studying how cache replacement policies affect our contention modeling approach.

We are also interested in extending the cross-architecture framework to more than two core cases. To make the framework even more powerful, we might employ transfer learning techniques to create a mapping between a small training input to a large reference input, so further reduction to the profiling time can be achieved.

Bibliography

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper. 22,* 6 (Apr.), 685–701.

[2] Aggarwal, C. C. and Reddy, C. K. 2013. *Data Clustering: Algorithms and Applications*, 1st ed. Chapman & Hall/CRC.

[3] Alpaydin, E. 2010. *Introduction to Machine Learning*, 2nd ed. The MIT Press.

[4] Ausavarungnirun, R., Chang, K. K.-W., Subramanian, L., Loh, G. H., and Mutlu, O. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. *SIGARCH Comput. Archit. News 40,* 3 (June), 416–427.

[5] Bao, B. and Ding, C. 2013. Defensive loop tiling for shared cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. CGO '13. IEEE Computer Society, Washington, DC, USA, 1–11.

[6] Bennett, B. T. and Kruskal, V. J. 1975. LRU stack processing. *IBM J. Res. Dev. 19,* 4 (July), 353–357.

[7] Berg, E. and Hagersten, E. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *ISPASS*. IEEE Computer Society, 20–27.

[8] Berg, E. and Hagersten, E. 2005. Fast data-locality profiling of native execution. In *SIGMETRICS*. ACM, 169–180.

[9] Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. ACM, New York, NY, USA, 72–81.

[10] Bitirgen, R., Ipek, E., and Martinez, J. F. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. IEEE Computer Society, Washington, DC, USA, 318–329.

[11] Blagodurov, S., Zhuravlev, S., and Fedorova, A. 2010. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst. 28,* 4 (Dec.), 8:1–8:45.

[12] Bodik, P., Goldszmidt, M., Fox, A., Woodard, D. B., and Andersen, H. 2010. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. ACM, New York, NY, USA, 111–124.

[13] Brock, J., Ye, C., Ding, C., Li, Y., Wang, X., and Luo, Y. 2015. Optimal cache partition-sharing. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*. ICPP '15. IEEE Computer Society, Washington, DC, USA, 749–758.

[14] Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. 2000. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl. 14,* 3 (Aug.), 189–204.

[15] Bruening, D., Duesterwald, E., and Amarasinghe, S. 2000. Design and implementation of a dynamic optimization framework for Windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4*.

[16] Buck, B. and Hollingsworth, J. K. 2000. An API for runtime code patching. *Int. J. High Perform. Comput. Appl. 14,* 4 (Nov.), 317–329.

[17] Burnham, K. P. and Anderson, D. R. 2002. *Model selection and multimodel inference: a practical information-theoretic approach*, 2 ed. Springer.

[18] Burtscher, M., Kim, B.-D., Diamond, J., McCalpin, J., Koesterke, L., and Browne, J. 2010. Perfexpert: An easy-to-use performance diagnosis tool for HPC applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '10. IEEE Computer Society, Washington, DC, USA, 1–11.

[19] Caliński, T. and Harabasz, J. 1974. A dendrite method for cluster analysis. *Communications in Statistics-Simulation and Computation 3,* 1, 1–27.

[20] Chandra, D., Guo, F., Kim, S., and Solihin, Y. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture.* HPCA '05. IEEE Computer Society, Washington, DC, USA, 340–351.

[21] Cho, S. and Jin, L. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06).* 455–468.

[22] Choi, H., Lee, J., and Sung, W. 2011. Memory access pattern-aware DRAM performance model for multi-core systems. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software.* 66–75.

[23] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S. 2004. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In

Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6. OSDI'04. USENIX Association, Berkeley, CA, USA, 16–16.

[24] de Blanche, A. and Lundqvist, T. 2015. Addressing characterization methods for memory contention aware co-scheduling. *J. Supercomput. 71,* 4 (Apr.), 1451–1483.

[25] Delimitrou, C. and Kozyrakis, C. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not. 48,* 4 (Mar.), 77–88.

[26] Delimitrou, C. and Kozyrakis, C. 2014. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not. 49,* 4 (Feb.), 127–144.

[27] Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. 2011. Characterizing multi-threaded applications based on shared-resource contention. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '11. IEEE Computer Society, Washington, DC, USA, 76–86.

[28] Du Bois, K., Eyerman, S., and Eeckhout, L. 2013. Per-thread cycle accounting in multi-core processors. *ACM Trans. Archit. Code Optim. 9,* 4 (Jan.), 29:1–29:22.

[29] Duda, R. O., Hart, P. E., and Stork, D. G. 2000. *Pattern Classification*, 2nd ed. Wiley-Interscience.

[30] Ebrahimi, E., Lee, C. J., Mutlu, O., and Patt, Y. N. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. ACM, New York, NY, USA, 335–346.

[31] Eizenberg, A., Hu, S., Pokam, G., and Devietti, J. 2016. Remix: Online detection and repair of cache contention for the JVM. *SIGPLAN Not. 51,* 6 (June), 251–265.

[32] Eklov, D., Black-Schaffer, D., and Hagersten, E. 2011. Fast modeling of shared caches in multicore systems. In *HiPEAC*. ACM, 147–157.

[33] Eklov, D. and Hagersten, E. 2010a. Statstack: Efficient modeling of LRU caches. In *ISPASS*. IEEE Computer Society, 55–65.

[34] Eklov, D. and Hagersten, E. 2010b. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 55–65.

[35] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. 2011. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the 2011 International Conference on Parallel Processing*. ICPP '11. IEEE Computer Society, Washington, DC, USA, 165–175.

[36] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. 2013. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10.

[37] Eklov, D., Nikoleris, N., and Hagersten, E. 2014. A software based profiling method for obtaining speedup stacks on commodity multi-cores. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 148–157.

[38] Eyerman, S., Bois, K. D., and Eeckhout, L. 2012. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 145–155.

[39] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. ACM, New York, NY, USA, 175–184.

[40] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst. 27,* 2 (May), 3:1–3:37.

[41] Feliu, J., Petit, S., Sahuquillo, J., and Duato, J. 2014. Cache-hierarchy contention-aware scheduling in cmps. *IEEE Transactions on Parallel and Distributed Systems 25,* 3 (March), 581–590.

[42] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not. 47,* 4 (Mar.), 37–48.

[43] Gerndt, M. and Ott, M. 2010. Automatic performance analysis with periscope. *Concurr. Comput. : Pract. Exper. 22,* 6 (Apr.), 736–748.

[44] Gulur, N., Mehendale, M., Manikantan, R., and Govindarajan, R. 2014. Anatomy: An analytical model of memory system performance. *SIGMETRICS Perform. Eval. Rev. 42,* 1 (June), 505–517.

[45] Henning, J. L. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News 34,* 4 (Sept.), 1–17.

[46] Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., and Wang, Z. 2016. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* 351–364.

[47] Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. 2011. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst. 22,* 6 (June), 931–945.

[48] Jain, A. and Lin, C. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *SIGARCH Comput. Archit. News 44,* 3 (June), 78–89.

[49] Jain, A. K. and Dubes, R. C. 1988. *Algorithms for Clustering Data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[50] Jaleel, A., Theobald, K. B., Steely, Jr., S. C., and Emer, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News 38,* 3 (June), 60–71.

[51] Jeong, M. K., Yoon, D. H., Sunwoo, D., Sullivan, M., Lee, I., and Erez, M. 2012. Balancing DRAM locality and parallelism in shared memory CMP systems. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture.* HPCA '12. IEEE Computer Society, Washington, DC, USA, 1–12.

[52] Jiang, Y., Shen, X., Chen, J., and Tripathi, R. 2008. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques.* PACT '08. ACM, New York, NY, USA, 220–229.

[53] Jiang, Y., Zhang, E. Z., Tian, K., and Shen, X. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction.* CC'10/ETAPS'10. Springer-Verlag, Berlin, Heidelberg, 264–282.

[54] Karkhanis, T. S. and Smith, J. E. 2004. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 338–349.

[55] Kaseridis, D., Stuecheli, J., and John, L. K. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. ACM, New York, NY, USA, 24–35.

[56] Kessler, R. E., Hill, M. D., and Wood, D. A. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers 43,* 6 (Jun), 664–675.

[57] Khan, S. M., Tian, Y., and Jimenez, D. A. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '43. IEEE Computer Society, Washington, DC, USA, 175–186.

[58] Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M. 2010. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, M. T. Jacob, C. R. Das, and P. Bose, Eds. IEEE Computer Society, 1–12.

[59] Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 65–76.

[60] Kim, Y. H., Hill, M. D., and Wood, D. A. 1991. Implementing stack simulation for highly-associative memories. *SIGMETRICS Perform. Eval. Rev. 19,* 1 (Apr.), 212–213.

[61] Kuang, W., Brown, L. E., and Wang, Z. 2015. Modeling cross-architecture co-tenancy performance interference. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 231–240.

[62] Kufrin, R. 2005. Perfsuite: An accessible, open source performance analysis environment for Linux. In *In Proc. of the Linux Cluster Conference, Chapel.*

[63] Kundu, S., Rangaswami, R., Dutta, K., and Zhao, M. 2010. Application performance modeling in a virtualized environment. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–10.

[64] Kundu, S., Rangaswami, R., Gulati, A., Zhao, M., and Dutta, K. 2012. Modeling virtualized applications using machine learning techniques. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. ACM, New York, NY, USA, 3–14.

[65] Levon, J. 2004. *OProfile Manual*. Victoria University of Manchester.

[66] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378.

[67] Little, J. D. C. 1961. A proof for the queuing formula: $l = \lambda w$. *Oper. Res. 9,* 3 (June), 383–387.

[68] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. ACM, New York, NY, USA, 190–200.

[69] Marowka, A. 2011. On performance analysis of a multithreaded application parallelized by different programming models using Intel VTune. In *Proceedings of the 11th International Conference on Parallel Computing Technologies*. PaCT'11. Springer-Verlag, Berlin, Heidelberg, 317–331.

[70] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. ACM, New York, NY, USA, 248–259.

[71] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2 (June), 78–117.

[72] McVoy, L. and Staelin, C. 1996. Lmbench: portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. ATEC '96. USENIX Association, Berkeley, CA, USA, 23–23.

[73] Michaud, P., Seznec, A., and Jourdan, S. 2001. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Int. J. Parallel Program. 29,* 1 (Feb.), 35–58.

[74] Mitchell, T. M. 1997. *Machine Learning*, 1 ed. McGraw-Hill, Inc., New York, NY, USA.

[75] Montgomery, D. C., Peck, E. A., and Vining, G. G. 2006. *Introduction to Linear Regression Analysis.* Wiley & Sons.

[76] Mutlu, O. and Moscibroda, T. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO 40. IEEE Computer Society, Washington, DC, USA, 146–160.

[77] Mutlu, O. and Moscibroda, T. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. *SIGARCH Comput. Archit. News 36,* 3 (June), 63–74.

[78] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. 1996. Vampir: Visualization and analysis of MPI resources. *Supercomputer 12*, 69–80.

[79] Olken, F. 1982. Efficient methods for calculating the success function of fixed space replacement policies. PROGRES report LBL-12370, University of California, Lawrence Berkeley Laboratory.

[80] Olszewski, M., Mierle, K., Czajkowski, A., and Brown, A. D. 2007. JIT instrumentation: A novel approach to dynamically instrument operating systems. *SIGOPS Oper. Syst. Rev. 41,* 3 (Mar.), 3–16.

[81] Ould-Ahmed-Vall, E., Woodlee, J., Yount, C., Doshi, K. A., and Abraham, S. 2007. Using model trees for computer architecture performance analysis of software applications. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 116–125.

[82] Patterson, D. A. and Hennessy, J. L. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[83] Pillet, V., Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., Girona, S., and Computadors, D. D. D. 1995. PARAVER: a tool to visualize and analyze parallel code. Tech. rep., In WoTUG-18.

[84] Qureshi, M. K. and Patt, Y. N. 2006. 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*. MICRO 39. IEEE Computer Society, Washington, DC, USA, 423–432.

[85] Riseman, E. M. and Foster, C. C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers C-21,* 12 (Dec), 1405–1411.

[86] Rousseeuw, P. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math. 20,* 1 (Nov.), 53–65.

[87] Sandberg, A., Black-Schaffer, D., and Hagersten, E. 2012. Efficient techniques for predicting cache sharing and throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. ACM, New York, NY, USA, 305–314.

[88] Sandberg, A., Sembrant, A., Hagersten, E., and Black-Schaffer, D. 2013. Modeling performance variation due to cache sharing. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 155–166.

[89] Sembrant, A., Eklov, D., and Hagersten, E. 2011. Efficient software-based online phase classification. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*. IISWC '11. IEEE Computer Society, Washington, DC, USA, 104–115.

[90] Sheikholeslami, G., Chatterjee, S., and Zhang, A. 1998. WaveCluster: a multi-resolution clustering approach for very large spatial databases. In *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB '98. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 428–439.

[91] Shen, K., Zhong, M., Dwarkadas, S., Li, C., Stewart, C., and Zhang, X. 2008. Hardware counter driven on-the-fly request signatures. *SIGOPS Oper. Syst. Rev. 42,* 2 (Mar.), 189–200.

[92] Shen, X. and Shaw, J. 2008. Languages and compilers for parallel computing. Springer-Verlag, Berlin, Heidelberg, Chapter Scalable Implementation of Efficient Locality Approximation, 202–216.

[93] Shen, X., Shaw, J., Meeker, B., and Ding, C. 2007. Locality approximation using time. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. ACM, New York, NY, USA, 55–61.

[94] Snavely, A. and Tullsen, D. M. 2000. Symbiotic job scheduling for a simultaneous multi-threaded processor. *SIGARCH Comput. Archit. News 28,* 5 (Nov.), 234–244.

[95] Soares, L., Tam, D., and Stumm, M. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. IEEE Computer Society, Washington, DC, USA, 258–269.

[96] Subramanian, L., Seshadri, V., Ghosh, A., Khan, S., and Mutlu, O. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. ACM, New York, NY, USA, 62–75.

[97] Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., and Mutlu, O. 2013. MISE: providing performance predictability and improving fairness in shared main memory systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA '13. IEEE Computer Society, Washington, DC, USA, 639–650.

[98] Sun, X.-H. and Wang, D. 2012. APC: a performance metric of memory systems. *SIGMETRICS Perform. Eval. Rev. 40,* 2 (Oct.), 125–130.

[99] Tam, D. K., Azimi, R., Soares, L. B., and Stumm, M. 2009. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. *SIGPLAN Not. 44,* 3 (Mar.), 121–132.

[100] Tang, L., Mars, J., and Soffa, M. L. 2012. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. ACM, New York, NY, USA, 1–12.

[101] Taylor, G., Davies, P., and Farmwald, M. 1990. The tlb slice&mdash;a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.* ISCA '90. ACM, New York, NY, USA, 355–363.

[102] Thiebaut, D. and Stone, H. S. 1987. Footprints in the cache. *ACM Trans. Comput. Syst. 5,* 4 (Oct.), 305–329.

[103] Thornock, N. C. and Flanagan, J. K. 2000. Facilitating level three cache studies using set sampling. In *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165).* Vol. 1. 471–479 vol.1.

[104] Tsamardinos, I., Brown, L. E., and Aliferis, C. F. 2006. The max-min hill-climbing bayesian network structure learning algorithm. *Mach. Learn. 65,* 1 (Oct.), 31–78.

[105] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture.* ISCA '12. IEEE Computer Society, Washington, DC, USA, 213–224.

[106] Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I. 2015. Efficient MRC construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies.* FAST'15. USENIX Association, Berkeley, CA, USA, 95–110.

[107] Wang, W., Dey, T., Davidson, J. W., and Soffa, M. L. 2014. DraMon: predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA).* 380–391.

[108] Wang, W., Yang, J., and Muntz, R. R. 1997. Sting: A statistical information grid approach to spatial data mining. In *Proceedings of the 23rd International Conference on Very*

Large Data Bases. VLDB '97. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 186–195.

[109] Wang, X., Li, Y., Luo, Y., Hu, X., Brock, J., Ding, C., and Wang, Z. 2015. Optimal footprint symbiosis in shared cache. In *CCGRID*. IEEE Computer Society, 412–422.

[110] Wires, J., Ingram, S., Drudi, Z., Harvey, N. J. A., and Warfield, A. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. USENIX Association, Berkeley, CA, USA, 335–349.

[111] Wood, T., Cherkasova, L., Ozonat, K., and Shenoy, P. 2008. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. Springer-Verlag New York, Inc., New York, NY, USA, 366–387.

[112] Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely, Jr., S. C., and Emer, J. 2011. SHiP: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. ACM, New York, NY, USA, 430–441.

[113] Xiang, X., Bao, B., Bai, T., Ding, C., and Chilimbi, T. 2011. All-window profiling and composable models of cache sharing. *SIGPLAN Not. 46,* 8 (Feb.), 91–102.

[114] Xiang, X., Bao, B., Ding, C., and Gao, Y. 2011. Linear-time modeling of program working set in shared cache. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. IEEE Computer Society, Washington, DC, USA, 350–360.

[115] Xiang, X., Ding, C., Luo, H., and Bao, B. 2013. HOTL: a higher order theory of locality. *SIGARCH Comput. Archit. News 41,* 1 (Mar.), 343–356.

[116] Xu, C., Chen, X., Dick, R. P., and Mao, Z. M. Cache contention and application performance prediction for multi-core systems.

[117] Xu, D., Wu, C., and Yew, P.-C. 2010. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. ACM, New York, NY, USA, 237–248.

[118] Xu, D., Wu, C., Yew, P.-C., Li, J., and Wang, Z. 2012. Providing fairness on shared-memory multiprocessors via process scheduling. *SIGMETRICS Perform. Eval. Rev. 40,* 1 (June), 295–306.

[119] Xu, H., Wen, S., Giménez, A., Gamblin, T., and Liu, X. 2017. DR-BW: identifying bandwidth contention in NUMA architectures with supervised learning. In *IPDPS*. IEEE Computer Society, 367–376.

[120] Yang, H., Breslow, A., Mars, J., and Tang, L. 2013. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. ACM, New York, NY, USA, 607–618.

[121] Ye, Y., West, R., Cheng, Z., and Li, Y. 2014. Coloris: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 381–392.

[122] Yoo, W., Larson, K., Baugh, L., Kim, S., and Campbell, R. H. 2012. ADP: automated diagnosis of performance pathologies using hardware events. In *Proceedings of the 12th ACM*

SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. SIGMETRICS '12. ACM, New York, NY, USA, 283–294.

[123] Zhang, X., Dwarkadas, S., and Shen, K. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. ACM, New York, NY, USA, 89–102.

[124] Zhao, J., Cui, H., Xue, J., and Feng, X. 2016. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Trans. Parallel Distrib. Syst. 27,* 5 (May), 1443–1456.

[125] Zhao, J., Feng, X., Cui, H., Yan, Y., Xue, J., and Yang, W. 2013. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 201–212.

[126] Zhao, L., Iyer, R., Illikkal, R., Moses, J., Makineni, S., and Newell, D. 2007. CacheScouts: fine-grain monitoring of shared caches in CMP platforms. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 339–352.

[127] Zhao, Q., Koh, D., Raza, S., Bruening, D., Wong, W.-F., and Amarasinghe, S. 2011. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '11. ACM, New York, NY, USA, 27–38.

[128] Zhong, Y., Dropsho, S. G., Shen, X., Studer, A., and Ding, C. 2007. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers 56,* 3 (March), 328–343.

[129] Zhong, Y., Shen, X., and Ding, C. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. 31,* 6 (Aug.), 20:1–20:39.

[130] Zhuravlev, S., Blagodurov, S., and Fedorova, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. ACM, New York, NY, USA, 129–142.

Appendix A

Copyright documentation