The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

CHARACTERIZING AND OPTIMIZING ON-CHIP SHARED MEMORY RESOURCES USING MARKET-DRIVEN MECHANISMS

A Thesis in

Computer Science and Engineering

by

Shail Shah

© 2019 Shail Shah

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

May 2019

The thesis of Shail Shah was reviewed and approved* by the following:

Mahmut Kandemir
Professor of Computer Science and Engineering
Thesis Co-Adviser

John (Jack) Sampson
Assistant Professor of Computer Science and Engineering
Thesis Co-Adviser

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

Heterogeneous applications often share memory resources such as last-level caches and memory bandwidth within many-core system deployed in IaaS model. The performance of applications in such environment depends highly upon the contention caused in the shared resources by the co-runners. Resource provider wants to dynamically allocate these physical resources among various application running in the system. They aim to improve the hardware utilization of the system while maximizing the benefit of the clients. Different applications derive varying utility as a function of the amount of resources allocated to them; making it difficult for the resource providers. We present a way to evaluate the performance metric of an application under varying resource constraints along with co-runners. We present a market-driven mechanism which uses auction to allocate LLC and memory bandwidth among different application in the system. We used benchmarks from SPECCPU2006 suite to evaluate our mechanism. Experimental results show the improvement of 1.5x-2x as compared to the state-of-the-art algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all the people who have helped me throughout my graduate program and successfully completing my thesis.

Firstly, I would like to thank my advisor Dr. Mahmut Kandemir for providing this wonderful opportunity to work with him, provide invaluable advice and encouragement throughout my graduate program. I am extremely grateful to my co-advisor Dr. Jack Sampson for his detailed guidance, patient counseling and help during my research. It would have been very difficult without the help, direction and support from both the professors.

I would also like to thank my fellow members of the MDL lab who have been there to provide encouragement and were readily available to discuss new ideas, provide implementation support and technical guidance during my two years at Penn State.

Chapter 1

Introduction

With the advent of cloud computing, the allocation of shared resources among the applications running on many-core systems have been an active area of research in the past several years. The performance of the applications running in such an environment depends a lot on the co-runners. Interference from the co-runners in the on-chip shared memory resources like Last-Level Caches(LLC) and Memory Bandwidth (MB) can have a deteriorating effect on the performance of resource-sensitive applications. With the different heterogeneous co-runners, it is difficult to guarantee the isolation by the cloud providers among the applications. In these many core systems, large LLC's might be helpful in improving the scalability and throughput of the applications, but on the other hand, they may be prone to certain low priority applications which pollutes the cache and act as a noisy neighbor. Such noisy neighbors can harm the performance of the high priority applications running on different cores. In these cases, we need a software-defined infrastructure approach that could provide isolation of shared resources among various applications and which could be helpful in resource-aware orchestration.

Intel released Cache Allocation Technology (CAT) in order to tackle the issue of interference in shared memory resources. CAT provides control over on-chip shared resources like LLC's and MB from the software layer. It provides isolation and prioritization among various applications running in many-core systems by reserving cache space in LLC's. It restricts memory bandwidth usage of an application. CAT provides QoS guarantees to the application without worrying about the co-runners running on the multicore systems. It provides the flexibility to assign and reserve a chunk of LLC or MB to certain applications.

Software-defined control over LLC and memory bandwidth helps in enhancing the orchestration capabilities for the provider. Cloud provider needs to allocate the resources to the applications in order to maximize the benefit of its clients and to maximize the hardware utilization of the system. It is difficult for the provider to understand the utility of the resources to the clients and their willingness-to-pay for the resources when each application have different utility for it. Thus, it is important to characterize the latency of applications along with varying resource constraints. We have done a detailed off-line characterization of selected applications from the SPECCPU2006 benchmarks to measure their latency. The characterization involves running applications in a resource-constrained environment with varying levels of the number of ways of LLC and the percentage of memory bandwidth allocated to the application. In order to measure the effect of co-runners on the latency, we have developed a synthetic kernel, namely Bubble, which exerts pressure on the last-level caches. Bubble is a tunable parameter which put a varying level of pressure on the LLC cache. Detailed off-line profiling of applications in different resource-constrained situations with varying levels of Bubble intensity provided insightful gains in the latency of applications in various configurations.

The results of the characterization are used to understand the valuation of different resource configuration in varying co-running environment to the applications. We present an auction mechanism that is used to allocate the LLC and memory bandwidth to all the applications in the system. Our mechanism gathers the valuation metrics from different application and selects the combination which has the maximum valuation. We use Vickrey-Clarke-Groves(VCG) auction which is a sealed-bid auction that tries to maximize the social welfare of the system. We aim to maximize the social welfare of the system by minimizing the weighted slowdown of all the applications in the system.

Chapter 2

Background and Related Work

Resource allocation of last level caches and memory bandwidth have been an active area of research since many years. With the emergence of many-core systems, the capacity of last level caches has been constantly on rise. Along with the increased capacity, recent datacenter workloads have become data-intensive. Interference in the shared memory resources have been an active area of research and various methods of cache partitioning, isolation and memory bandwidth allocation have been studied in past years. Researchers have come up with many static and dynamic allocation methodology and algorithms to improve the efficiency and performance of the systems. There are three major research areas I would like to divide the related works mostly into three parts:- cache partitioning technologies and game-theory based market-driven resource allocation strategies and various characterization methods to estimate the effect of interference on the performance of the application.

Previous works in the past like that of Muli Ben-Yehuda [6] demonstrates the use of auctioning to allocate memory among the different co-runners. Works like [13], [14], [15] considers various game-theoretical mechanisms to improve utilization of resources in datacenter management. L. Funaro [5] uses auction mechanisms to distribute LLC among various applications. But, his work doesn’t take into account the effect of memory bandwidth along with the shared LLC. Also, the granularity of allocation is restricted to only exclusive caches. Lazar [12] uses second price auction for sharing network bandwidth.

Cache Partitioning have been an active field of research in the architecture community. The cache partitioning has become easier in the recent processors due to the invent of CAT [2] in some of the Intel's processor SKU. The works done in this domain till now either provide a hardware based partitioning [7], [8] or a software based allocation approaches [9], [10], [11]. Our work provides a novel way to look at the use of Intel's cache allocation technology to allocate the resources using the auction algorithm. Many past works have proposed schemes to provide QoS in datacenters. Several solutions like [19, 20, 21] looks at detecting interference and migrating the workload in datacenters. These schemes mainly work at detecting the interference and avoiding it. Many previous works provided hardware solutions to mitigate cache pollutions by isolating the workloads and using isolation or preventing from resident data from the caching [16,17,18]. Cache partitioning in software using coloring [22] has been proposed to deal with the cache pollution. Almost all the solution proposed previously except [5] used simulators and were not implemented in the real hardware. Also, many of the solutions performed rely on the performance characterization of the application without the incentive for the application to reveal their performance. We present an auction mechanism which would benefit the applications from revealing their true valuation of the resources. Also, the previous work targeted partitioning either LLC or memory bandwidth. We provide a software controlled approach to partition memory bandwidth and LLC using auction mechanism in a commodity hardware.

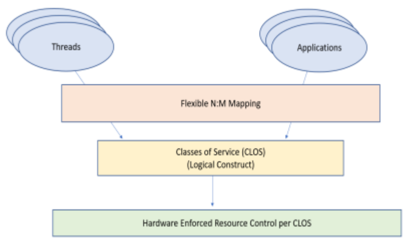
Chapter 3

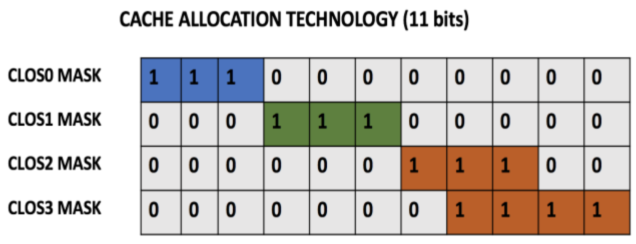
Intel Resource Director Technology

Intel released a series of technological products as a part of Resource Director Technology (RDT) which were aimed to provide more control and monitoring support over the underlying hardware. Cache Allocation Technology(CAT)[2] and Cache Monitoring Technology(CMT)[3] focused on providing control and monitoring various threads usage of on-chip caches like L2 and L3. Memory Bandwidth Allocation (MBA) focused on providing control over the memory bandwidth allocated on a per core/ per process basis. In many core system environment with large last level caches, co-running applications or VM's can place different demand pressure on the LLC cache. Since LLC is shared among all the cores on a processor, an application can often pollute the LLC affecting the performance of other applications sharing the LLC. In order to tackle the issue of cache-pollution from the noisy neighbor, CAT provides the support for reserving space for various threads, VM's, containers in the last-level caches (LLC), along with the monitoring capabilities of cache usage by the respective processes. This thesis uses CAT for allocation or the LLC's and memory bandwidth (MB) among various applications. In this chapter, we will have a look at the workings of CAT, how to use CAT to reserve LLC's and MB among various applications and the limitations of using CAT. CMT provides the feature of tracking of LLC to increase the determinism of the performance of applications in the system.

3.1. Overview

As, discussed above, Cache Allocation Technology (CAT) helps in providing isolation of LLC space on a per-core/per-process basis. This feature was enabled starting from the Xeon E5-2600 v3 family and are available in most of the latest processor sku's starting from Xeon v4 family and scalable processors family including Broadwell and Skylake processors. CAT is needed in order to restrict the applications from over-utilizing the caches in many-core scenarios in data centers. It helps in ensuring scalability and throughput of applications by protecting important processes through prioritization of processes in the datacenters. It provides the software control to reserve space in LLC and Memory Bandwidth for a thread or an application by providing logical constructs called CLOS (classes of service). CAT has been provided through kernel support (resctrl) and through software packages that provides the API's to use it in programming languages like C. Various interfaces to use CAT has different additional requirements that changes with the linux kernel versions and interface being used. There are various packages available like Linux kernel, Xen, Linux Cgroups which could acts as an interface to enable and use CAT. Intel has released a github repository called intel-cmt-cat which provides the software wrappers that could help in performing CAT operations or reserving LLC space and monitoring.


Threads
Applications
Flexible N:M Mapping
Classes of Service (CLOS)
(Logical Construct)
Hardware Enforced Resource Control per CLOS


CACHE ALLOCATION TECHNOLOGY (11 bits)

CLOS0 MASK	1	1	1	0	0	0	0	0	0	0	0
CLOS1 MASK	0	0	0	1	1	1	0	0	0	0	0
CLOS2 MASK	0	0	0	0	0	0	1	1	1	0	0
CLOS3 MASK	0	0	0	0	0	0	0	1	1	1	1

Figure 3-1 Working of CLOS-threads mapping and example of using CLOS to allocate resources in LLC

3.2. Working of CAT

CAT can be used by orchestrator framework or from any other high-level application framework from software level by enabling software support. CAT decides the amount of LLC space allocated to a thread or an application by bunch of MSR's which contain the capacity bitmasks (CBM). CBM contains the information about how many ways are assigned or allocated across threads and the degree of isolation of those ways. It helps in deciding the amount of LLC available to an application and whether is it shared or isolated from other applications. All the threads/ applications/VM's are mapped to a logical construct called CLOS (classes of service). A thread can belong to any one CLOS. They are also synonymously called as Resource Monitoring ID's (RMID) for monitoring purposes. Example explaining the working of CLOS and the capacity bit masks (CBM) have been showed in the Figure3-1 above. Each CLOS is assigned a specific portion of the resources which translates down to all the threads that are assigned to that CLOS. CLOS is the lowest form of logical construct for allotment of resources through RDT. Each CLOS can be assigned resources through MSR drivers which are responsible for modifying the capacity bitmasks in the hardware. Pqos utility tries to modify the CLOS allocation by directly using the MSR and CPUID drivers through standard file I/O API. Linux has provided the support for the CAT through RESCTRL filesystem supports which acts as a wrapper to the MSR drivers. Resources allocated to a particular CLOS and the threads and application mapped to a particular CLOS both can be dynamically changed during the runtime. The number of CLOS that are present depends on the processor sku's. They are usually fixed and could not be modified for a given processor.

Chapter 4

Characterization

When applications run on many-core systems in the cloud or datacenter environment, it is very difficult to predict their performance in the shared environment. These applications share a lot of shared resources like LLC caches, memory bandwidth, network and I/O resources. The interference cause by co-runners in these situations may vary depending on the resources as well as the co-runners. Some applications may be sensitive to the pressure in the shared resources created by the co-runners, while on the other hand they may be the polluter themselves causing interference to the others. Thus, it becomes very difficult for the cloud providers or the central resource provider in the data center to allocate the shared resources efficiently to the applications or the containers in order to optimize the hardware utilization of the system. The cloud provider tries to maximize the benefit of the client's. The question is, how would the cloud provider know the importance of the resources to the applications. Different applications can show different sensitivity to different type of resources. In order to understand the applications sensitivity to these shared resources and to understand the behavior of these applications in terms of execution time(latency), we will do the exhaustive characterization of the applications in different scenarios. Our scope would be limited to the shared LLC's and memory bandwidth (MB) for this thesis. By the end of the characterization phase, our goal is to attain an utility metric which shows how much each application value different levels of the shared resources under different co-runners. The variable parameters in our characterization are numbner of LLC cache ways allocated to the application, the number of shared LLC cache ways allocated to the application, the intensity of the pressure put on the shared LLC by the co-runners and the amount of memory bandwidth allocated to the application.

4.1. Benchmarks

In these section, we have defined a set of benchmarks (applications) that we have selected for the characterization. We have selected 10 applications from the SPECCPU2006 suite of benchmarks. They are combination of both floating point and interger benchmarks. The benchmarks that are selected show varying amount of sensitivity in terms of latency(execution time) to the amount of the LLC cache and memory bandwidth allocated to that application. These benchmarks amount to the subset of high-performance computing applications that are computationally intensive as well as hugely dependent on the shared on-chip memory resources. Table 1 shows the detailed description of all the benchmarks that are used for characterization and their use-cases. All the benchmarks are run with the reference inputs which denotes the highest size of the input available in the benchmark suite.

BENCHMARKS	DESCRIPTION
Astar	Efficient computer algorithm for path finding between multiple nodes in a graph.
Bzip2	File compression program using Burrows-wheeler algorithm
Hmmer	Program used to analyse protein sequences using pHMMs (profile hidden markov models). Used in searching gene sequences.
Leslie	Used in fluid mechanics to simulate various turbulence phenomena of combustion and fluid mechanics
Libquantum	Simulates a quantum computer running Shor's
Mcf	Solve the problem of vehicle scheduling to schedule public transport using network simplex algorithm.
Milc	Simulates four dimensional lattice gauge theory. Used mainly in the field of Quantum Chromodynamics.
Omnetpp	Used in discrete event-based simulation of Ethernet networks. Used in the field of network communication.
Sjeng	AI program to play chess again the opponents.
Tonto	Simulator used in the field of quantum chemistry to place constraint on Hartree-Fock wave function calculation

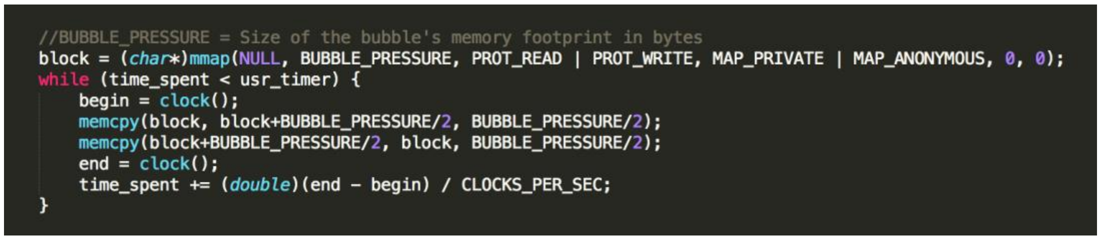
Table 1 List of benchmarks from the SPECCPU2006 suite used for characterization

4.2. Designing Characterization Experiments

The main goal of our characterization is to evaluate the effect of various allocations of the shared on-chip memory resources (LLC and bandwidth) along with the various co-runners on the performance of our benchmarks selected above. In order to mimic the behavior of the co-runner with varying level of pressures, we have created a synthetic workload called Bubble which is described in the next section.

4.2.1 Bubble

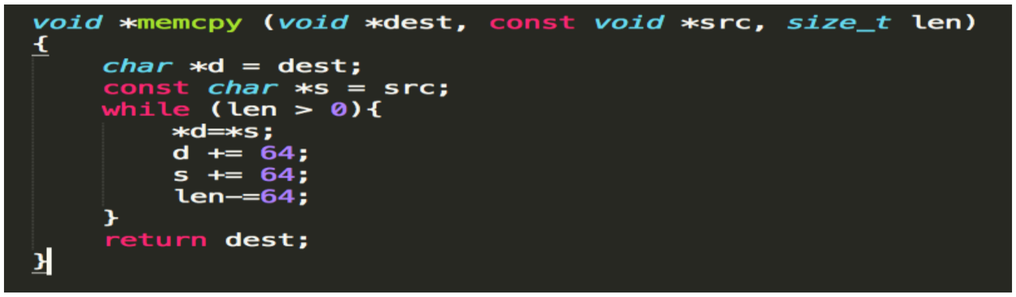
Bubble is the synthetic workload that mimics co-runners by putting tunable amount of pressure on the shared Last-level caches. The amount of pressure put on the LLC is analogous to the memory footprint in the LLC cache. The design of the bubble should be such that there has to be an inverse relationship of bubble pressure with the execution time of the application. As we increase the pressure of the bubble, it should result in the deteriorating impact on the execution time or performance of the application. In the whole thesis, increase or decrease in the bubble pressure refers to increasing or decreasing the memory footprint of the bubble in the LLC cache per a single way. So, if bubble is sharing only one way of LLC with it’s co-runner, then the bubble of higher level will have more memory footprint in the LLC cache as compared to the lower-level bubble. Bubble pressure is a function of the number of ways that are shared along with the co-runner. For ex, when we say, bubble of 160Kb and 2 ways are shared with the co-runner, it means that bubble exerts the total pressure of (160 * 2)Kb on the LLC cache. Thus, the final memory footprint of our bubble constitutes number of shared ways and the bubble level. Along with the memory footprint, bubble should be intense in the terms of the number of memory accesses per unit time. For the design of the bubble, we tried 2 different kernels. The basic common layout of both the kernels remains the same as shown in Figure 4-1. It defines region in the memory using mmap relative to the amount of bubble pressure that is needed and it continuously does the memcpy operation in that array copying one part of the memory to the another for a given amount of time specified by the user.



```
//BUBBLE_PRESSURE = Size of the bubble's memory footprint in bytes
block = (char*)mmap(NULL, BUBBLE_PRESSURE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
while (time_spent < usr_timer) {
    begin = clock();
    memcpy(block, block+BUBBLE_PRESSURE/2, BUBBLE_PRESSURE/2);
    memcpy(block+BUBBLE_PRESSURE/2, block, BUBBLE_PRESSURE/2);
    end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
}
```

Figure 4-1 Basic layout of the Bubble's kernel

After defining the basic structure, we experimented with the various implementation of the **memcpy** function. The first kernel uses the in-built memcpy function defined in the gcc library. We wanted the bubble to be able to consistently put pressure on LLC. We designed an implementation of the memcpy function, which performs cache line accesses and just copies a byte from each cacheline. This way, it could generate more accesses to different cache lines, instead of accessing the consecutive elements within the same cache line. Figure 4-2 shows the modified memcpy function used in the second kernel. We ran both the kernel along with a LLC sensitive synthetic workload in order to measure the effect of Bubble on it. Figure 4-3 shows the latency of the LLC sensitive synthetic workload when run with the two kernels. All the experiments are done by running LLC sensitive workload attached to a fixed core and the Bubble running on the separate core. The number of ways of LLC that are shared by both are configured using CAT. Kernel 1 is not at all effective in exerting the pressure on the LLC cache. It has a minor effect on the latency of the co-runner. On the other hand, kernel 2 shows the larger variation in the latency w.r.t the increase in the bubble pressure. The variation in latency is not quite visible when there are lesser number of ways that are shared with the co-runner. We can infer from the graphs that our bubble exerting the correct pressure in terms of the memory footprint, but it still doesn't have much effect on the latency of the synthetic workload. There is a possibility that our bubble is not intense enough in terms of number of memory accesses per unit time. In order to test this theory, we performed a set of experiments by increasing the intensity of the bubble. Bubble of size 1600Kb, sharing 1 way with the co-runner and intensity of 10 means that there are 10 threads running Bubble on separate core with each of them exerting 160Kb of pressure. Similarly, a bubble of size 640Kb, sharing 5 ways and intensity of 10 means that there are 10 threads running bubble on separate core with each of them exerting 320Kb of pressure. Figure 4-4 shows the results of the experiments of running synthetic workload along with kernel 2 of varying intensity of the bubble.



```
void *memcpy (void *dest, const void *src, size_t len)
{
        char *d = dest;
        const char *s = src;
        while (len > 0){
            *d=*s;
            d += 64;
            s += 64;
            len-=64;
        }
        return dest;
}
```

Figure 4-2 Modified memcpy function performing cacheline accesses instead of the consecutive access

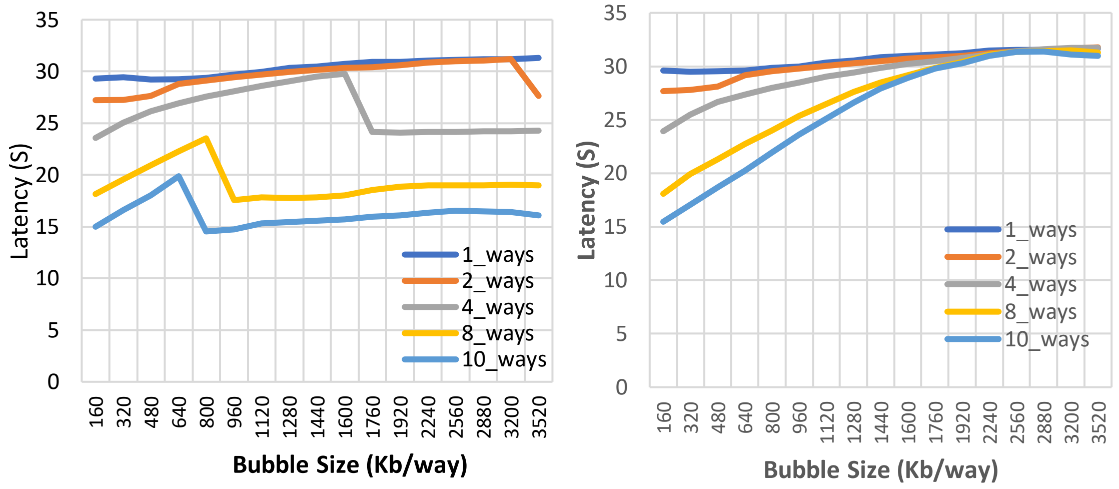


Figure 4-3 Effect of varying Bubble pressure on the latency using a) inbuilt-memcpy (on the left) b) custom memcpy making cacheline accesses (on the right)

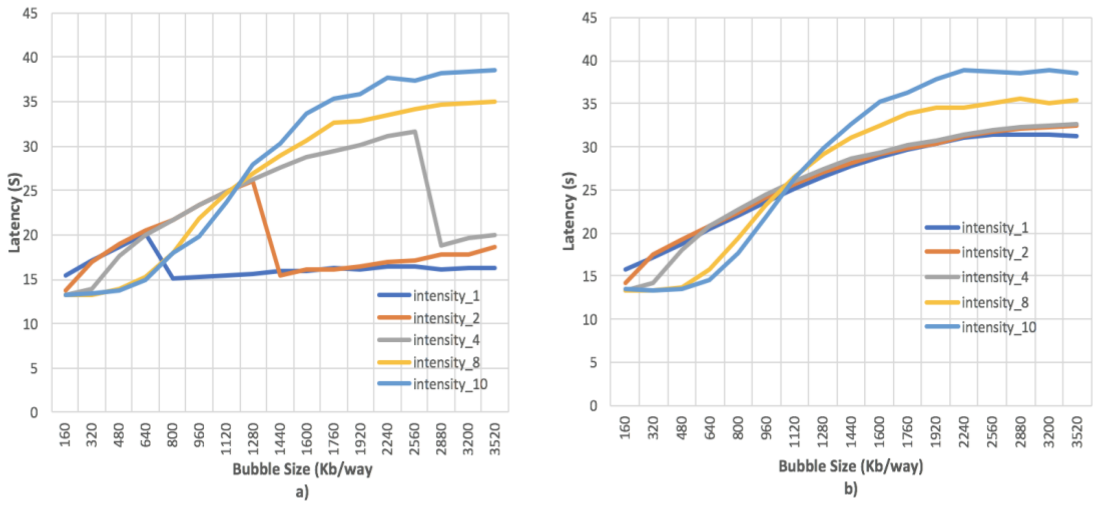
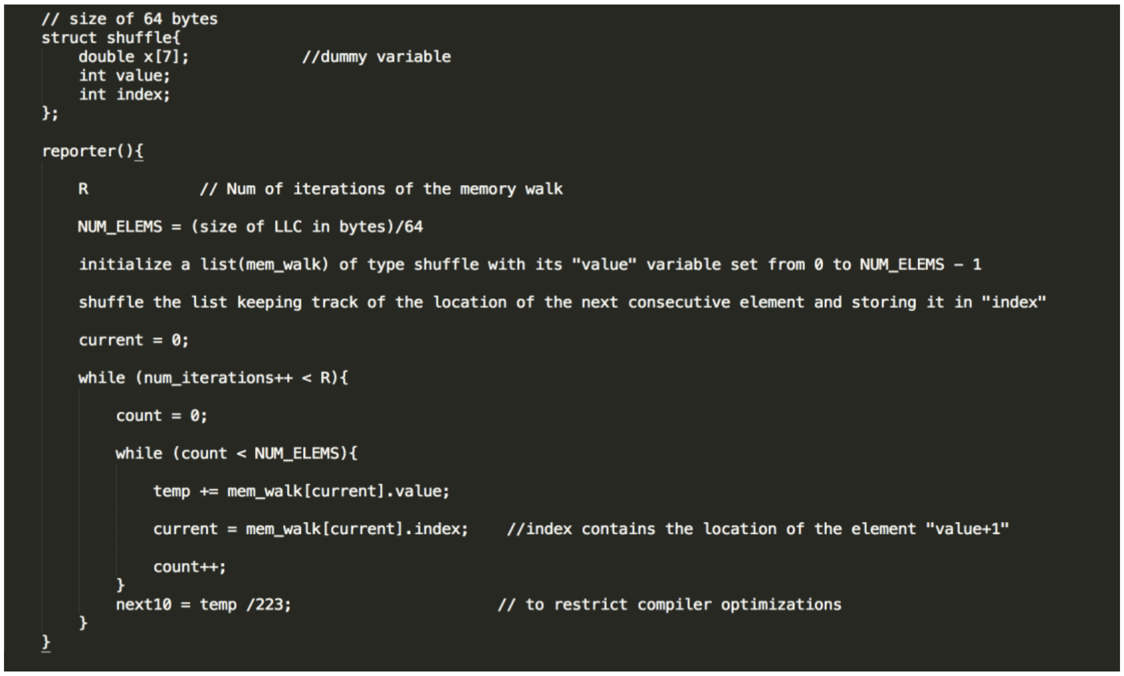


Figure 4-4 Effect of Bubble pressure with varying intensity on the latency in a 10-way shared environment using a) inbuilt-memcpy b) custom memcpy making cacheline accesses

From Figure4-4 we can see that our custom memcpy function gives a relatively smoother curve and a larger variation in the execution times as the intensity of the bubble increases. Larger variation in the latency denotes the effectiveness of the bubble in causing interference to the co-runners. Finally, we have selected our bubble as a custom memcpy kernel as shown in Figure 4-2 with intensity of 10. It means that bubble will run the memcpy kernel on 10 separate cores with each core generating the pressure of one-tenth of the total Bubble pressure. In all the future sections, total bubble pressure would amount to the product of size_of_the_memory_footprint and the number of shared ways with the co-runner. Ten such threads of bubble would be responsible for exerting the total bubble pressure on the LLC. We will be using 11 different levels of bubble with varying memory footprint ranging from 160Kb/way to 1760Kb/way in the increments of 160Kb. These bubble levels are predefined and would be used in characterization of applications in the shared LLC's scenarios.

4.2.2 Reporter

Reporter is the synthetic workload which is sensitive to the contention in the LLC. It is used to measure the interference caused by an application to other co-runners. Each application is mapped to the corresponding bubble level in order to get an estimate of the contentiousness of the application. The idea is to first run the reporter along with the different levels of the bubble to get the sensitivity curve of the reporter w.r.t the bubble. It gives an idea about how sensitive is reporter w.r.t varying level of pressure in LLC. Subsequently, we run the reporter along with our benchmarks individually and measure the latency of the reporter in each individual situation. We use the reporter's sensitivity curve to translate the degradation in the performance by the co-running application to a corresponding bubble value. The sensitivity curve of the reporter needs to be calculated only once and can be used in future for all the applications to find their contentiousness level. Reporter doesn't need to have any fixed design. However, it should follow some basic characteristics. The latency of reporter should be inversely related to the amount of LLC that is allocated to it. reporter should be designed in such a way that it is susceptible to latency variation by the amount of LLC available to the kernel and also by the interference caused by the co-running application. It should be sensitive enough to get affected by the varying memory footprint size of co-runners in LLC. Our reporter function initializes the array equivalent to the size of the LLC. Each element of the array is of size 64 bytes which is the size of the cache line on our machine. So, the number of elements in the array is equivalent to the (size of the LLC in bytes)/64. We perform a simple memory walk using the indexes in the array. The detailed pseudo-code for the reporter function is mentioned in the Figure 4-5 below. Reporter is a single threaded kernel as opposed to the bubble function. It defines an array whose memory footprint is equivalent to the size of LLC and performs memory access through all the portions of LLC by performing a memory walk in the array.



```
// size of 64 bytes
struct shuffle{
    double x[7];            //dummy variable
    int value;
    int index;
};

reporter(){

    R           // Num of iterations of the memory walk

    NUM_ELEMS = (size of LLC in bytes)/64

    initialize a list(mem_walk) of type shuffle with its "value" variable set from 0 to NUM_ELEMS - 1

    shuffle the list keeping track of the location of the next consecutive element and storing it in "index"

    current = 0;

    while (num_iterations++ < R){

        count = 0;

        while (count < NUM_ELEMS){

            temp += mem_walk[current].value;

            current = mem_walk[current].index;    //index contains the location of the element "value+1"

            count++;
        }
        next10 = temp /223;                    // to restrict compiler optimizations
    }
}
```

Figure 4-5 Pseudo code for the reporter function

4.2.3 Experiment Setup

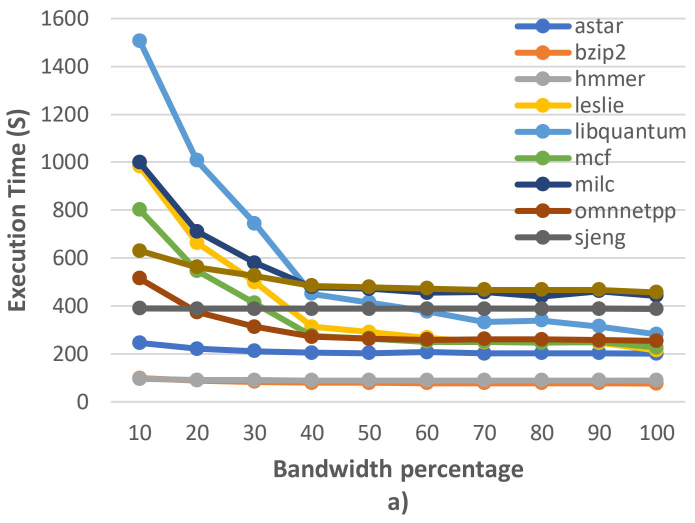
This section describes in detail about the design for various iterations of characterization in detail. All the benchmarks described above were run in multiple scenarios of varying LLC and bandwidth allocations along with varying interference from the co-runners simulated by bubble. All the iterations were run on Intel(R) Xeon(R) Gold 6126 processor. The processor had total of 19712KB of shared l3 cache which were shared by 12 physical cores on the processor. L3 cache was 11-way associative accounting for ~1.75MB per way. Linux kernel 4.18.7-041807-generic was used and perf monitoring tool was used to gather performance related metrics of each application. Intel CAT was used through its kernel interface called resctrl which is available in linux kernel versions above 4.10. All the kernel and other user processes were restricted to use one way in order to restrict the pollution from other co-running processes. Other 10 ways were used in order to characterize the applications in different combination of private and shared ways. Each way could be either be allocated exclusively to the application or can be shared with a bubble of different intensity. During characterization phase, each application was tied to a core and all the other user processes were moved to a single core. Remaining 10 cores were used to simulate a co-runner using bubble as described in the previous section with each core exerting one-tenth of the total bubble pressure. The characterization ran multiple iterations of each application under different configuration of shared and private ways allocated to the application and measured the latency in each configuration. In the remaining thesis, the following notation would be used in order to denote the respective allocation of the LLC ways. A <N_p, N_s, B> where A is the application, N_p denoted the number of exclusive cache ways, N_s denotes the number of shared cache ways and B refers to the bubble level of the co-runner running in the shared cache ways. For ex, A <0, 2, 160> means that application A is running with 0 private ways and 2 cache ways shared with a bubble of size 160KB per cache way. Our experiments performs an exhaustive characterization of an application by various different possible values of N_p, N_s and B. The value of N_p, N_s ranges from [0,10] as there are 10 possible caches ways and value of B ranges from 160KB to 1760KB with increment of 160KB, thus resulting in 11 different bubble levels. The values of N_p, N_s should be allocated such that their summation should not be greater than the total number of available cache ways ($N_p + N_s <= 10$). This yields 715 different possible allocation patterns of LLC cache ways along with co-runner per application. Along with LLC, we have done detailed characterization of memory bandwidth allocation. The execution time of each application also depends on the memory bandwidth allocated to that particular resource. Some applications may value LLC more, while other applications may value memory bandwidth; on the other hand some applications may be sensitive to both the resources. Thus, it is important to include memory bandwidth allocation as the part of characterization. Thus, each possible allocation <N_p, N_s, B> were executed multiple times with varying memory bandwidth allocated to the application each time. The range of values of memory bandwidth(MB) varied from 10% to 100% with increment of 10%. We only performed characterization upto MB ranging from 10% to 40% due to the reasons mentioned in the next section.

4.2.4 Analysis of Characterization

In this section, we will analyze the data obtained during the characterization experiments in details. Our characterization produced an exhaustive performance metric with four variable parameters, namely number of exclusive cache ways, number of shared cache ways, bubble pressure of the shared cache ways, memory bandwidth percentage. Due to the exhaustive iterations of the experiments, there are lot of insights that can be obtained from those performance metrics which could help us in reducing the search space in the auction algorithms. Auction algorithms are usually variations of combinatorial optimization algorithms which have exponential runtime in the order of the number of bidders and number of items in the system. Thus, analysis of the characterization might give us a detailed insight about the performance of applications in various situations and will be helpful to us in developing heuristics to restrict the search space in the auction algorithms.

4.2.4.1 Effect of Memory Bandwidth

In this section, we will have a look at the effect of varying memory bandwidth allocated to the application. In these experiments, all the applications have been given exclusive access to all the ways in the LLC with the only variable parameter being the percentage of memory bandwidth allocated to the process. Figure 4-6 shows the execution time of the applications under varying memory bandwidth allocations.



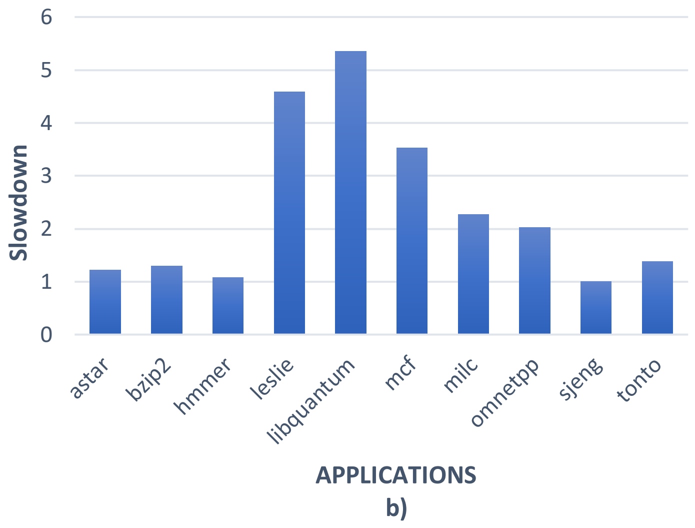


Figure 4-6 a) Effect of Memory Bandwidth allocation on the execution time b) Worst case slowdown experienced by each application

We can infer from the graph, that after 40% of memory bandwidth most of the application reaches a saturated state. This is the reason for restricting our characterization of bandwidth till 40%. Also, in applications like libquantum and leslie, restricted memory bandwidth access leads to a significant slowdown of up to 4-5x in the execution time. On the other hand, for applications like astar, hmmer and sjeng, memory bandwidth plays an insignificant role in their execution time.

4.2.4.2 Last-Level Cache-way Analysis

Applications shared last-level caches in the many core systems. Their performance could be impacted by the co-runners executing on other cores. The applications sensitivity to the co-runner also depends on how much of the LLC does the application actually need to execute efficiently. In order to understand the requirement of the applications, Figure 4-7 shows the execution time of each application w.r.t the number of allotted private ways and the worst-case slowdown experienced by the application. The memory bandwidth is set to 100% and the only variable parameter here is the number of exclusive cache ways.

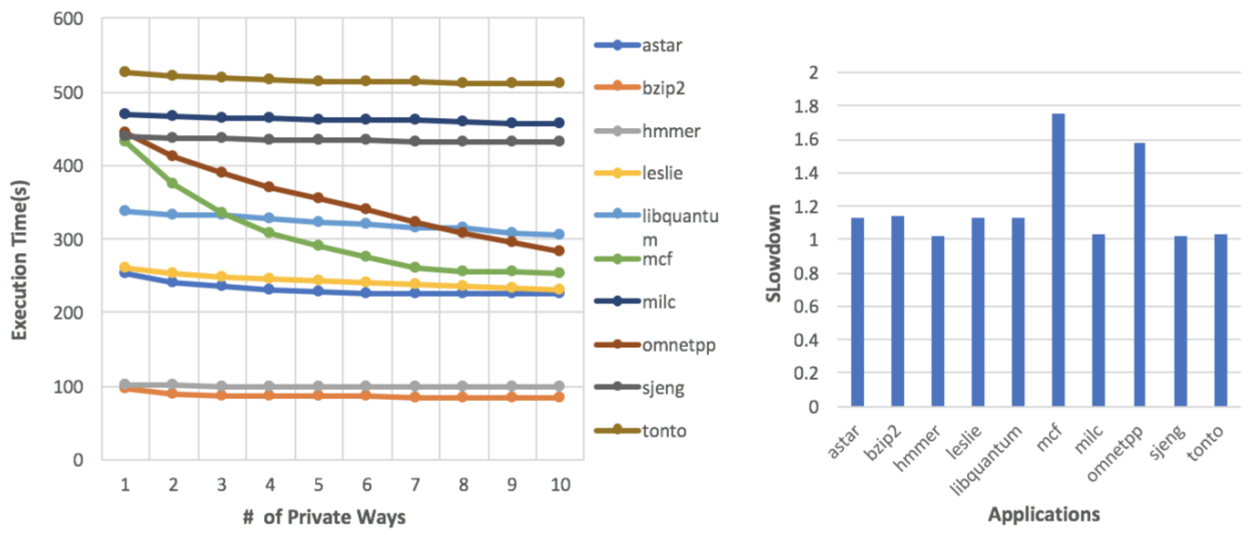


Figure 4-7 Effect of varying exclusive cache on the execution time

We have seen the worst-case slowdown experienced by the applications when only exclusive caches were provided. But, in the real world scenario, multiple co-runners cause interference in the cache, thus evicting applications' data from the cache and increasing the reuse distance. Figure 4-8 shows the worst-case slowdown when an application is subjected to varying <Np,Ns, B> environment. The memory bandwidth is set to 100% and the variable parameters are the number of private cache ways, number of shared cache ways and the bubble pressure of the co-runner. From the graph, we can see that some applications like tonto didn't have any effect of the size of the LLC cache, but they are highly sensitive to the interference. On the other hand, applications like hmmer and sjeng doesn't show any sensitivity to either the size of the LLC or the interference by the co-runner.

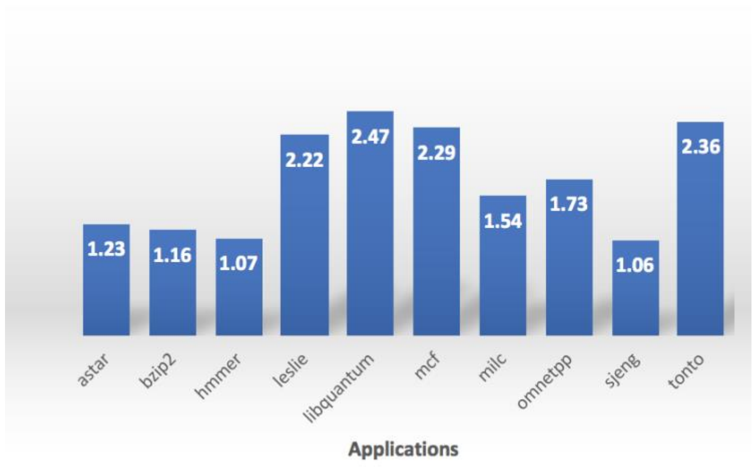


Figure 4-8 Worst-case slowdown with varying private and shared cache ways

4.2.4.3 Effect of Bubble Intensity

Along with the number of private and shared caches allotted to an application, the execution time of an application can be affected by the interference of the co-runner. In order to understand the behavior of the applications in such scenario, we varied the intensity of the co-runners using the bubble function. In this section, we will analyze the effect of varying bubble intensity on the execution time of the applications. Heterogeneous applications may have different sensitivity to the co-runners. In order to understand the effect of bubble levels, we will fix the number of exclusive and shared cache ways (bubble only uses shared cache ways). Figure 4-9 shows the effect of varying levels of bubble on the execution times of some of the applications. You can see that sjeng doesn't have any effect on the execution times. It is extremely insensitive to the interference in the cache. On the other hand, applications like mcf and libquantum can show upto 2-3x in performance degradation.

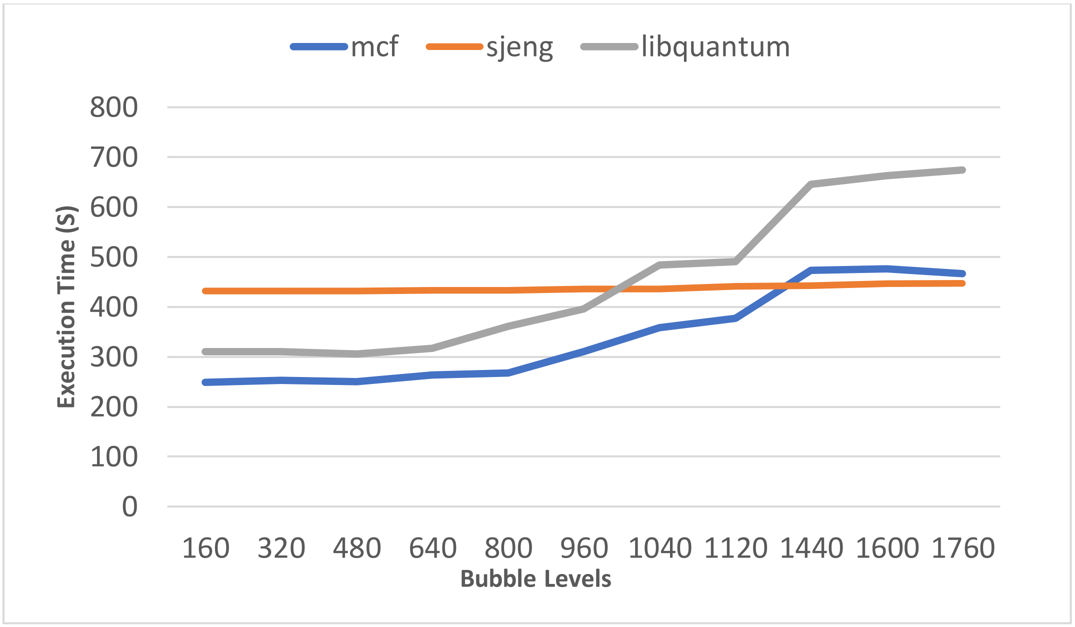


Figure 4-9 Effect of interference on execution time of applications

Chapter 5

Auction

Cloud providers wants to allocate the resources among different applications in order to maximize the client' benefits. Applications are like black boxes to the cloud providers. It is very difficult for the cloud providers to evaluate the effect of a particular allocation on the performance of the application. Also, applications are oblivious to the co-runners on the system. All the applications know their own valuation of various resources from the performance metrics calculated during the off-line profiling(characterization). Thus, it is in best interest of the cloud providers to let the applications decide their own valuation of a particular set of resources and their willingness-to-pay for it. Applications would bid their true valuation under various resource constraints and cloud providers would try to optimize the allocation in order to maximize the aggregated benefits of all the clients. Seller usually try to sell the resources either in order to maximize the social welfare or to maximize the revenue. In our scenario, provider's wants to improve the aggregated benefit of the clients, the improving the efficiency of the system as the whole. In order to achieve this, we will use a well-known auction algorithm called VCG auction algorithm, which is explained in detail in the next section. VCG auction tries to maximize the social welfare of the system.

5.1. Vickrey-Clarke-Groves Auction

In this section, we will briefly describe the working of Vickrey-Clarke-Groves(VCG) auction mechanism. VCG mechanism tries to optimize the efficiency in the system. It does so by choosing the efficient outcome as allocation from the given bids and charging each player to pay for their welfare cost. It is a multi-unit auction and a type of sealed-bid auction. The bidders do not know the bids of the other players. Each bidder doesn't know the bids of the other players.

They only have an estimate about their valuations for the items. Valuation of the items can also be considered as their willingness-to-pay for the items. Allocations of items done by the auction mechanism tries to do it in the socially optimal manner. In VCG auction, the bidders can place multiple bids for the different combinations of the items being sold. Their willingness-to-pay for the item may vary along with the number of items they receive. Auction is closed after receiving the bids from all the bidders. All the auction mechanisms are usually exponential in complexity as they consider all the possible combination of the bids, before choosing the combination that maximizes the aggregated value of all the bids. VCG is a second-price auction where each bidder does not pay the value equivalent to their own bid. Instead, they pay for only the marginal harm caused to the bidders. VCG does not aim at maximizing the revenue of the seller. VCG is a truthful bidding algorithm where the optimal strategy for the bidder is to bid the true valuation of the combination of the items without lying about it. The final price to be paid by the bidder is calculated as the difference between the second maximum combination of the bid and the value that other's have bid in the winning combination. Each auction needs to define the bidders, the items to be sold, bidder's valuation for each item and the pricing scheme that each bidder need to pay if they win the items in the auction. We will define each of these in the next section.

5.2. LLC and memory bandwidth Auction

In each auction round, each application bids exclusive or shared access of the cache ways and the percentage of memory bandwidth. Each application places multiple bids for various combination of items evaluated from the off-line profiling. The valuation of an application is calculated in \$/s for a combination of resources. It is calculated by the formula given below:-

$$\text{Valuation}(Np, Ns, B) = S * \text{Perf}(Np, Ns, B)$$

$$\text{Valuation}(MB) = S * \text{Perf}(MB)$$

where Valuation (Np, Ns, B) refers to the valuation of an application for Np private cache ways, Ns shared ways, bubble intensity B and memory bandwidth MB; S is the scaling factor (which refers to the flexibility of the customers to pay) and Perf is the performance of the application for the respective allocation normalized from 0 to 1. Some applications may want to pay extra to reserve the resources for them, while other applications may be fine with whatever they get. This can be adjusted by setting the value of S. By default, S = 1; where each application bids their own valuation. S is measured in ($/performance unit) and Perf is measured in (Performance unit/s).

In each auction round, a single best combination of the bids is selected and the resources are allocated accordingly to the applications. In our case, the auctioneer announces only 9 cache-ways and 80% of the memory bandwidth for the auction. Remaining 2 cache-ways and 20% of the memory, namely Reserve is shared by all the applications which were not allocated anything in the auction. Those applications would only pay the fixed cost of using Reserve and would not pay for anything else. The payment rule according to the VCG auction, guarantees that the applications won't pay anything extra than the original bid places by that application.

Chapter 6

Methodology

This section covers in details about the experimental setup and the methodology for comparing our mechanism with alternative cache and memory bandwidth allocation methods.

6.1. Machine Setup

The machine used had a Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz processor with 19712K L3 cache shared among 11ways. There were 12 cpu cores. All the applications were tied to run separate cores. All the existing user processes were moved to run on a single core. The machine ran Ubuntu-16.04 with kernel version 4.18.7-041807-generic #201809090930. Dynamic cache allocation and memory bandwidth allocation were implemented through kernel interface using Resctrl file system. The number of CLOS available on this machine was limited to 8. All the processes were mapped to CLOS0 by default.

6.2. Alternative Resource Allocation and Auctioning Strategies

We compared our auction mechanism with the following two strategies for resource allocation. Both these strategies were used in comparing our mechanism for bandwidth as well as LLC cache allocation.

Shared-Resource: All the applications shared the resources without any isolation from one other. They shared entire LLC and memory bandwidth among themselves without any user-defined restrictions.

Equal-Partitioning: All the applications were allotted equal partition of the resources. The resources were equally allotted to each application, until the number of applications were lower than 8. If the number of applications exceed 8, then the remaining applications were simply assigned to the Reserve.

Performance-optimizing: In this method, resources were allocated using the auction mechanism which aim to maximize the aggregated performance of the system. Each application submitted there respective valuation of the resources as mentioned in the previous section. The scaling factor component of the valuation was set to 1 for all the applications.

As we have seen during the characterization, execution time of an application is a combined function of the amount of LLC and memory bandwidth allocated to it. Each of these allocation methods were compared for the following three auction scenarios-

Memory Bandwidth:- In this case, only memory bandwidth was auctioned with all the applications sharing the LLC without any isolation.

LLC Cache:- In this case**,** only the cache-ways in LLC were auctioned with no restrictions being placed on the memory bandwidth.

6.3. Experimental Setup

All the applications were divided into three types, based on their level of sensitivity to the resource being auctioned. Random mix of applications were generated using one of the following distributions:- **(4,1,0)**, **(2,1,2)**, **(0,1,4).** The distributions are denoted by (# of low sensitive applications, # of medium sensitive applications, # of high sensitive applications). Each application was spawned on the different core. There were only 8 CLOS available in the machine. Due to this limitation, if the number of applications winning a resource in auction exceeded by 7, they were removed from the auction and assigned automatically to the pit. The experiments were run for 50 iterations for each distribution and auction scenario mentioned in the previous section.

Chapter 7

Results

In this chapter, we will discuss and compare various allocation strategies and their overhead.

7.1. Memory Bandwidth Allocation

We compared our optimized strategy with equal partitioned and shared strategy. Figure 7-1 compares the slowdown suffered by each strategy. Each strategy has been compared by varying combination of 5 applications of (low sensitivity, medium sensitivity, high sensitivity). “All” section in the graph compares all the strategies while executing all the 10 applications in our benchmark as our co-runners. The slowdown in the graph is calculated as the average of slowdown of each application executing in the distribution. They are normalized to the “baseline” execution.

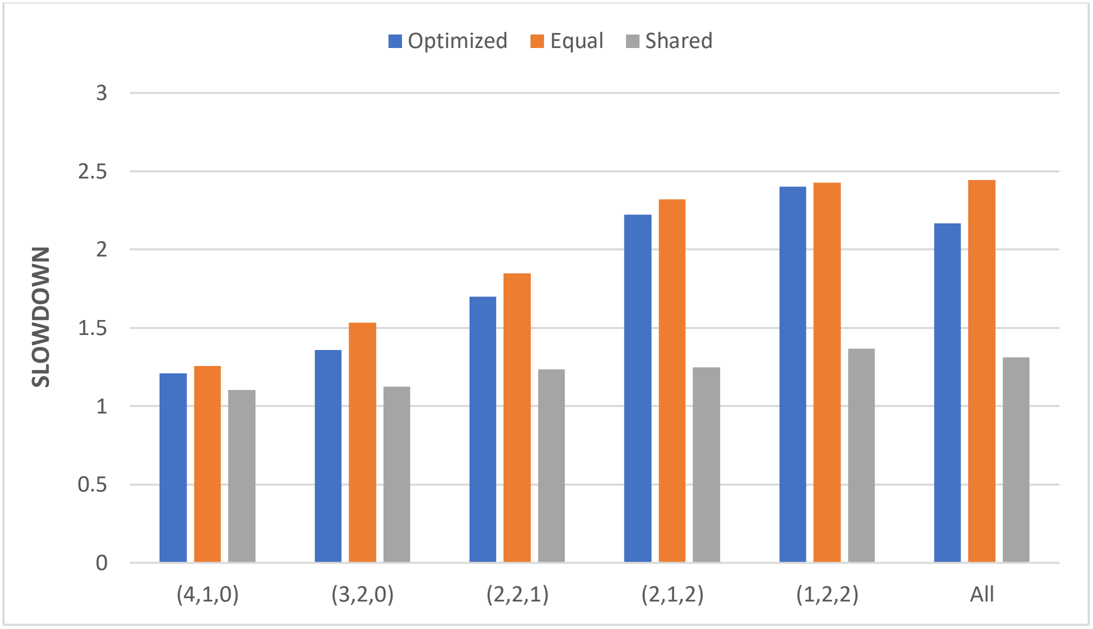


Figure 7-1 Comparison of various Memory Bandwidth allocation strategies

"Baseline" performance is the execution of an application when 100% of the memory bandwidth is allocated along with all the LLC ways. You can see from the graph that optimized allocation always performed better than the equal partitioned. But, when all the resources are shared among all the applications, slowdown observed is much lower. This is due to the fact that even though the applications in the mix are sensitive to the interference in the bandwidth, but the applications in the mix are not causing any interference. In the graph, as we go from left to right, the overall intensity of the combination increases; resulting in higher slowdowns.

7.2. LLC Allocation

We compare various allocation strategies with our optimized allocation strategy. We compared each strategy normalized to the baseline. "Baseline" performance is the execution time of an application when it is allocated all the cache ways along with 100% memory bandwidth.

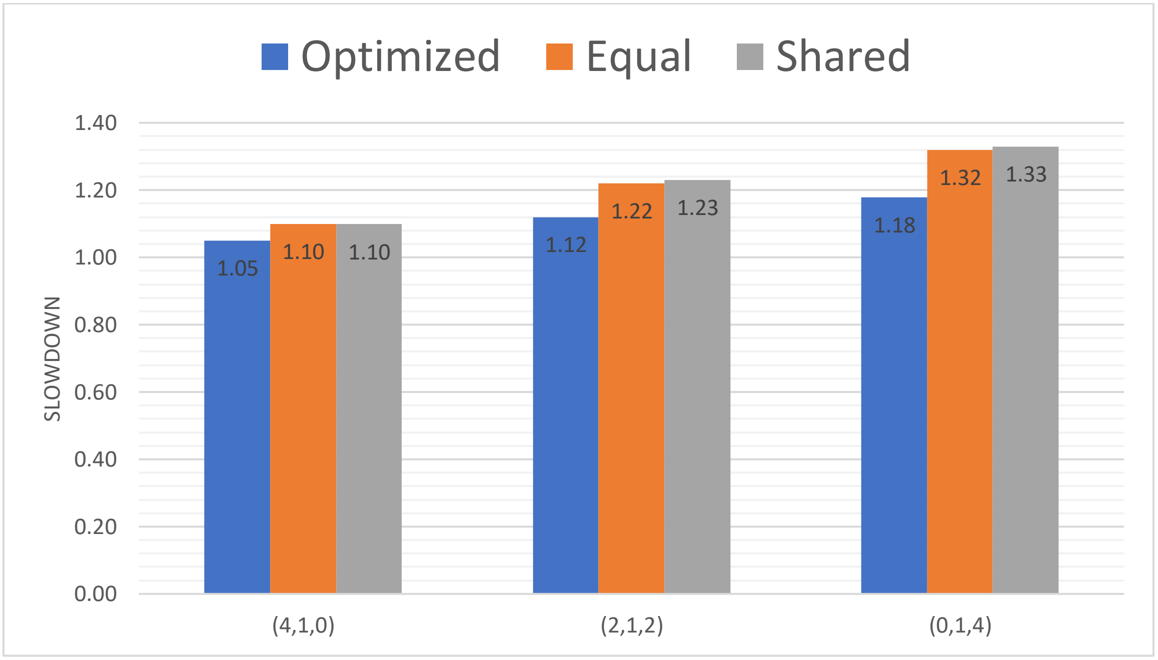


Figure 7-2 Comparison of various LLC allocation strategies

Figure 7-2 shows the comparison of various allocation strategy across the various distributions. From, the graph we can see that slowdown is similar in the case of both equally partitioned and shared cache caching scenario. It is due to the fact that there are 5 applications and each application would receive almost equivalent cache ways in both the scenarios.

7.3. Cache Leakage

Dynamically reassigning the cache ways to an application, may lead to loss of the data in the cache. It leads to cache leakage and those data is evicted and needs to be fetched into cache again on the next access. In order to study the effect of the cache leakage on the performance of an application, we executed the highest sensitive application *mcf* and reassigning it’s cache way allocation after every time quanta. Table 2 shows the execution time of mcf application with varying time quantum. We can see that it shows worst case performance degradation of around 3-4%, which is negligible.

Table 2: Effect of dynamic reassignment of LLC cache ways on the application

Time Quantum (s)	**Execution time (s)**
10	287.8
20	282.7
30	280
60	275
120	278

Chapter 8

Conclusion

Interference in shared resources like LLC and memory bandwidth could be a major cause of performance degradation in many-core systems. Depending upon the sensitivity of an application on the interference, we showed that it can have varying impact on the slowdown of an application by upto 5x. In these situations, it becomes extremely important to distribute the resources among the application and provide isolation in order to maximize the aggregated benefits of all the clients. Varying utility of allocated resources, helps in allocating resources at finer granularity where they can be shared by the co-runners. Thus, we present an auction mechanism to maximize the social welfare of the system and demonstrate it to incur a lower slowdown when compared with state-of-the art allocation algorithms.

REFERENCES

[1] AGMON BEN-YEHUDA, O., POSENER, E., BEN-YEHUDA, M., SCHUSTER, A., AND MU’ALEM, A. Ginseng: Market-driven memory allocation. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE) (2014), ACM, pp. 41–52.

[2] http://www.intel.com/content/www/us/en/communications/cacheallocation-technology-white-paper.html, 2015. Accessed May, 2016.

[3] INTEL OPEN SOURCE.ORG. Cache monitoring technology, memory bandwidth monitoring, cache allocation technology & code and data prioritization.

[4] Henning, John L. "SPEC CPU2006 benchmark descriptions." *ACM SIGARCH Computer Architecture News* 34.4 (2006): 1-17.

[5] AssafSchuster, LiranFunaro OrnaAgmonBen-Yehuda. "Ginseng: Market-driven llc allocation." *2016 USENIX Annual Technical Conference*. 2016.

[6] Agmon Ben-Yehuda, Orna, et al. "Ginseng: Market-driven memory allocation." *ACM SIGPLAN Notices*. Vol. 49. No. 7. ACM, 2014.

[7] GONZALEZ ´ , A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In ACM International Conference on Supercomputing 25th Anniversary Volume (2014), ACM, pp. 217–226.

[8] JAIN, P., DEVADAS, S., AND RUDOLPH, L. Controlling cache pollution in prefetching with software-assisted cache replacement. Comptation Structures Group, Laboratory for Computer Science CSG Memo 462 (2001).

[9] KASERIDIS, D., STUECHELI, J., AND JOHN, L. K. Bank-aware dynamic cache partitioning for multicore architectures. In International Conference on Parallel Processing (ICPP) (2009), IEEE, pp. 18–25.

[10] LEE, H., CHO, S., AND CHILDERS, B. R. Cloudcache: Expanding and shrinking private caches. In IEEE 17th International Symposium on High Performance Computer Architecture (HPCA) (2011), IEEE, pp. 219–230.

[11] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In Proceedings of the 42nd Annual International Symposium on Computer Architecture (2015), ACM, pp. 450–462.

[12] LAZAR, A. A., AND SEMRET, N. Design and analysis of the progressive second price auction for network bandwidth sharing. Telecommunication Systems—Special issue on Network Economics (1999)

[13] Seyed Majid Zahedi, Songchun Fan, Matthew Faw, Elijah Cole, Benjamin Lee. "Computational sprinting: Architecture dynamics, and strategies." ACM Transactions on Computer Systems (TOCS), 34(4):12.1-12.26, January 2017.

[14] Marisabel Guevara, Benjamin Lubin, Benjamin C. Lee. "Market mechanisms for managing datacenters with heterogeneous microarchitectures," ACM Transactions on Computer Systems (TOCS), 32(1):3.1-3.31, February 2014.

[15] Seyed Majid Zahedi, Songchun Fan, Benjamin C. Lee. "Managing heterogeneous datacenters with tokens." ACM Transactions on Architecture and Code Optimization (TACO), 15(2):18:1–18:23, June 2018.

[16] JAIN, P., DEVADAS, S., AND RUDOLPH, L. Controlling cache pollution in prefetching with software-assisted cache replacement. Comptation Structures Group, Laboratory for Computer Science CSG Memo 462 (2001).

[17] DYBDAHL, H., AND STENSTROM¨ , P. Enhancing last-level cache performance by block bypassing and early miss determination. In Advances in Computer Systems Architecture, C. Jesshope and C. Egan, Eds., vol. 4186 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 52–66.

[18] GONZALEZ ´ , A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In ACM International Conference on Supercomputing 25th Anniversary Volume (2014), ACM, pp. 217–226.

[19] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In Proc. ASPLOS-18, 2013

[20] J. Mars, L. Tang, R. Hundt, et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In Proc. MICRO-44, 2011.

[21] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In Proc. ISCA-38, 2011.

[22] TAYLOR, G., DAVIES, P., AND FARMWALD, M. The TLB slice a low-cost high-speed address translation mechanism. In Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA) (1990), ACM, pp. 355–363.