Diploma Project

DO WE NEED A NEW EVALUATION METHODOLOGY FOR MULTIPROGRAM SIMULATIONS?

Ioannis Constantinou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2022

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Do we need a new evaluation methodology for multiprogram simulations?

Ioannis Constantinou

Professor
Yiannakis Sazeides

The personal dissertation is submitted in partial fulfillment of requested obligations for receiving the degree of computer science from the department of Computer Science of the University of Cyprus

May 2022

Acknowledgments

At this point I would like to express my special thanks of gratitude to my Professor Mr. Yiannakis Sazeides, for the great opportunity to work with him on this topic. His continuous faith and trust in me, as well as his constructive criticism, helped me overcome every challenge that arose during this project. Most importantly, I want to thank him for the limitless amount of time he spent coaching me how to work properly and assisting me in understanding how I should handle any problem that comes my way. His supervision through this project matured me both academically and personally.

Abstract

Most high-performance processors today have multiple cores that can execute multiple programs simultaneously. Cores share processor resources, like the Last-Level Cache and Main Memory, which may be sources of performance variability because multiple cores may want to use them at the same time. In many research studies in Computer Architecture, a multicore processor is evaluated using microarchitecture simulators based on multi-program workload mixes. Specifically, for a multicore with C cores each mix used in a multi-program evaluation will contain a unique combination of C programs from a suite of benchmarks. The evaluation methodology in all previous multi-program studies considers each mix once, whereby each benchmark is assigned to a specific core. However, one can change the order that the C benchmarks in a mix are assigned to the C cores in multiple ways, to be exact, C! ways. The first question this project tries to answer is whether the order we run a benchmark mix affects the performance of a multicore simulation? In this work, we show, for a specific simulator and benchmark mixes, that changing the order of how benchmark's in a mix are assigned to cores changes the performance of the programs. We evaluate the range of performance (minimum to maximum) of a multicore processor has when changing the order for many mixes and show that it can vary considerably (up to 1.26% the maximum over the minimum). The second question we answer, is whether when comparing the performance of two different multicore processors using the legacy multi-program evaluation methodology can lead to wrong conclusions as far as which of the two multicores is best? We compare two Last-Level Cache replacement policies and show that the range of performance of the two policies overlap which means, legacy multi-program evaluation can lead to wrong conclusions for which policy is best. We propose a new evaluation methodology for multicore simulations that consider the range of performance for the processors we are studying so it would guide us better to understand which of the processors is best. Another important question we asked is, whether the performance variation due to the benchmark order in a mix appear on real machines also? We show that real machines may have variation on performance due to changing the order of a mix, but more data and analysis is needed to establish this clearly.

Contents

Chapter 1

Introduction

1.1 Introduction

With multicore processors nowadays the performance of a program can be affected by programs running simultaneously on the other cores, mostly because they all share resources (Last-Level Cache, Main Memory…). This impact varies depending on which are the other programs running and which core they are executing on.

The study of multicore performance on benchmark mixes requires detailed simulations. The most common evaluation methodology for this kind of study is to consider a set of single-thread benchmarks and randomly define a fixed set of benchmark mixes, simulate them and quantify the performance with a combined metric, i.e., Weighted Speedup [1, 2, 3, 4, 5, 6]. What this evaluation methodology does not take into consideration is on which core each benchmark of the mix is assigned, they run each mix once and each benchmark is assigned to a specific core. However, one can change the order that the C benchmarks of a mix are assigned to the C cores in various ways, to be exact C! ways.

So, one question that comes up is, does the order we run a benchmark mix affects the performance of a multicore simulation? We investigate this by exploring all the possible permutations each benchmark mix has considering a set of benchmark mixes. We performed simulations with Champsim Simulator [7], a simple trace-based simulator for microarchitecture studies that measures the IPC as a performance metric. We show that, changing the order of a benchmark mix affects the IPCs of the benchmarks that is consisted of. We also calculated the Weighted Speedups of the C! experiments for all the simulated benchmark mixes and show that changing the order of a benchmark mix affects also the Weighted Speedup of the benchmark mixes(Will be explained in detail in later sections) and evaluate the range of overall performance the multicore has (minimum to maximum) and it can vary significantly (up to 1.26% the maximum over the minimum). Hence, it is important to take into consideration the order the benchmarks are running. With all been said, the follow-up question is, when comparing two multicore processors performance using the Order-Aware multiprogram evaluation methodology, can lead us to wrong conclusions on which of the two is best?

We examine that using as a case study a comparison of two last-level cache replacement policies, SHIP and SHIP++, which we know clearly which is the best from previous work [1, 3]. We show that, the range of performance of these 2 policies overlap, which means that the Order-Aware multiprogram evaluation can lead us to wrong conclusions on which policy is best. So, it is not safe to simulate just one fixed order of the benchmark mixes, as done in the previous studies.

We propose a new multiprogram evaluation methodology, the Order-Aware Multiprogram evaluation methodology, in which we simulate all the potential orders for each benchmark mix and estimate the extremes (minimum and maximum) of the performance range for the multicore we study. That way, we'll have a better sense of a multicore processor's performance, which will aid us in making a more accurate conclusion regarding which processor is best in a comparison case study.

Finally, we also evaluate how changing the order affects the real machine performance, we run for each benchmark mix all the possible orders like we did for the simulation experiments with only difference that we run every experiment multiple times because a machine is not deterministic like a simulation. We show that also real machine may have variation on performance due to changing the order, however more data and analysis is needed to establish this clearly.

1.2 Outline

This dissertation consists of six chapters and is organized as follows. Chapter 1 was the introduction which presents what this work is, what problems this may have and why it's important to evaluate them. Chapter 2 presents the background someone needs to have to understand this work. Chapter 3 presents the current practices, describes in detail the problem of order sensitivity and propose the new evaluation methodology to evaluate the order sensitivity. Chapter 4 describes the experimental details for the simulator and real machine we used. We evaluate order sensitivity in Chapter 5 for simulator and real machine results. Finally, Chapter 6 concludes this work and discusses future work.

Chapter 2

Background

2.1 Multicore CPUs

Multicore processors have been in existence over two decades ago but have gained more importance off late due to the limits that single-core processors face today, such as high throughput and energy efficiency. A multicore processor is a single processor which contains two or more cores on a chip. The cores are functional units containing computation units and caches like any other single-core processor. These cores work together in order to achieve the performance of the more complex faster single-core processor. The individual cores on a multicore processor doesn’t necessarily run as fast as the highest performing single-core processors, but they improve throughput by executing more tasks simultaneously. Multicore processors commonly have private level caches and a level of a shared cache. The Figure 2.1 shows a 4-core multicore processor with L1 and L2 private caches and each core is connected via an interconnection network to a shared L3 cache and this cache is connected also via an interconnection network to the main memory.

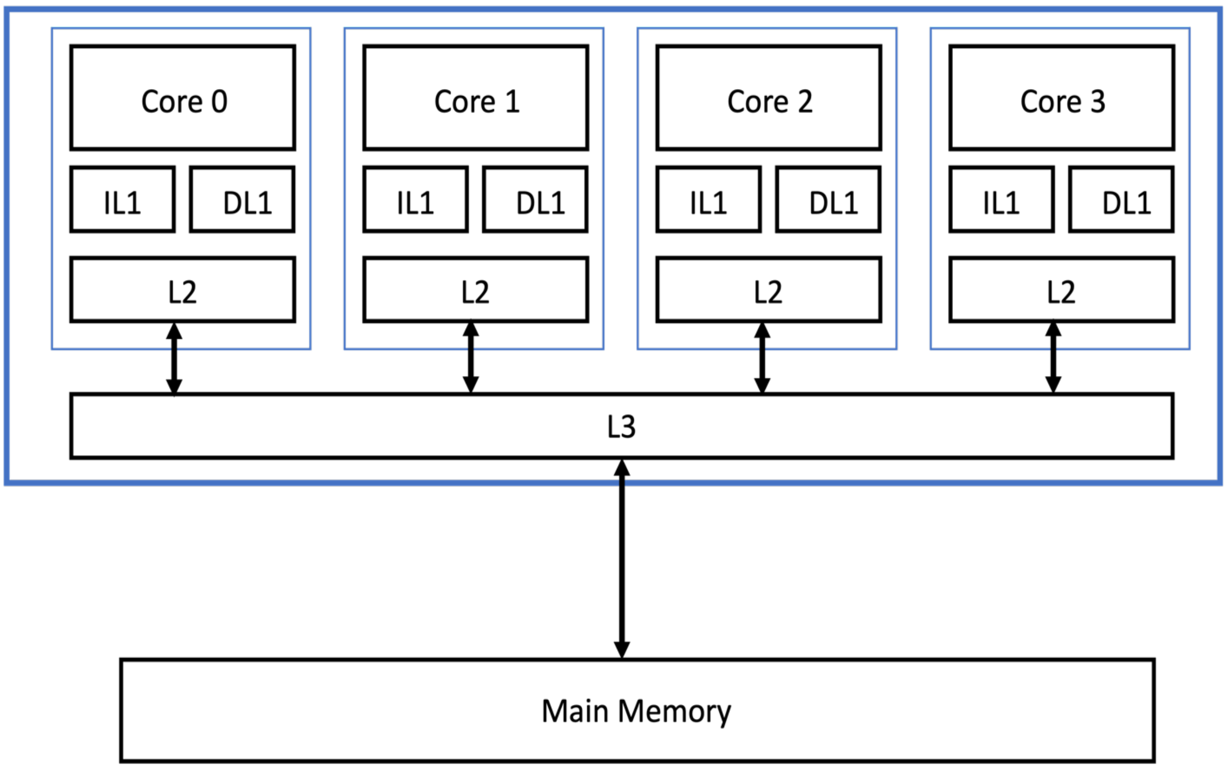


Figure 2.1 Multicore CPU Architecture

On these 4 cores on Figure 2.1 we can run 4 programs simultaneously. These programs throughout their execution are using the resources of the core they are assigned to like the core's Arithmetic Logic Unit (ALU), Instruction-Cache (IL1), Data-Cache (DL1) and maybe the L2 cache. As far as they use the private resources of the core they are assigned to there is no contention between the 4 programs. The problem of contention and performance variability comes when the 4 programs want to use the L3 cache and the Main Memory. So, an important challenge of multicore processors is the fair use of the shared resources because multiple cores access shared resources simultaneously and this may lead to variabilities on the performance of the programs that are running. For example, if a memory-intensive program executes on a multicore processor along with other 3 programs the same time it may use the shared resources a lot more time than the others, this will result to slow down the other 3 programs due to memory delays. Also, if we change the core on which the memory-intensive program executes, we might see a different result because the other 3 programs now may request access to the shared resources earlier than the memory-intensive program and get the data they want faster.

2.2 Simulators

In general, simulators are programs that simulate a behavior we want to study. A computer architecture simulator is a program that simulates the execution of a microarchitecture we want to study. Simulators are often used by researchers in the field of computer architecture to develop and assess new ideas, as well as to evaluate a pre-existing microarchitecture. It's important that the simulators are reliable and representative of pre-existing real machines and machines we are prototyping, also it's important that we use the right simulator for the study we want to do. Another objective to do simulation for a computer architect is to compare two multicore microarchitectures on execution time, multicore throughput, power consumption, etc. In this work, we look at how to evaluate better multicore throughput and how it varies under different conditions.

A benefit of simulations is that we can change any parameter and model any processor we want without any cost. For example, we can run a single-core run or multicore run, we can change how many levels of caches it has, the sizes of the caches, replacement policies of some caches and many more parameters. A downside is that detailed simulations are very slow, so we must be selective on the number of benchmarks we are going to use and how many combinations of these benchmarks we are going to create for multicore simulations. The selection process must be done carefully because we want to use a representative set of benchmarks to evaluate our model [2].

Chapter 3

Order Sensitivity

3.1 Current Multicore Evaluation Methodology

Simulations are widely used and very important for computer architects. One of the goals of simulations, especially with the emergence of multicore processors, is to evaluate the throughput of multicore processors, or the amount of work done by a machine while multiple independent applications are running at the same time. The most common evaluation methodology for multicore evaluation is to take a set of benchmarks and build random combinations of these benchmarks, called benchmark mixes, on which the processor will be evaluated. We call benchmark mix a combination of N benchmarks, N being the number of cores. Each benchmark mix consists of N unique benchmarks from a suite of benchmarks. Researchers consider a fixed sample of B benchmark mixes where B is just a few tens, for example 50 benchmark mixes, or sometimes 100 benchmark mixes, because detailed simulations are slow. The microarchitecture to be evaluated is simulated on all B benchmark mixes and they obtain a total of B x N IPC values. Then to determine the performance of each mix they use a combined metric, the most common metric is Weighted Speedup (WS). Equation 3.1 shows how the Weighted Speedup is calculated for each benchmark mix.

$$Weighted\ Speedup_{mix[k]} = \sum_{i=0}^{n} \frac{IPC_{k,i}^{MP}}{IPC_{i}^{SP}}$$

Equation 3.1 Weighted Speedup Formula

Weighted Speedup$_{mix[k]}$ is the summation of the multiprogram IPC of benchmark “i” and benchmark mix “k” over the single-program IPC of benchmark “i”, the experiments that give us the single-program IPC are using a single-core baseline configuration. In Equation 1 n represents the number of benchmarks the benchmark mix “k” contains. After they calculate the WS of all the benchmark mixes and for all the configurations they simulated, by configurations I refer to the multicore processor they want to evaluate and a reference multicore processor, they are doing the ratio of benchmark mix “k” WS of the current processor to the same benchmark mix “k” WS of a reference processor. Equation 3.2 show how they calculate the Ratio of each benchmark mix “k”. Basically, the Ratio represents how much better or worse the performance of each benchmark mix is compared to the reference multicore processor.

$$Ratio_{mix[k]} = \frac{WS^{curr}_{mix[k]}}{WS^{ref}_{mix[k]}}$$

Equation 3.2 Ratio Formula

After Equation 3.2 they end up with B Ratio values, where B as we said earlier is the number of benchmark mixes. The next and final step for the current evaluation methodology is to aggregate these B Ratio values in order to end up with a value that represents the overall performance of the processor they are studying. The aggregate metric that is most used is the geometric mean. Equation 3.3 show the formula of the geometric mean, which is the Bth root

$$Geometric\ Mean = \sqrt[B]{R_{mix[0]} * R_{mix[1]} * \cdots * R_{mix[B]}}$$

Equation 3.3 Formula of Geometric Mean of the products of the Ratio's, where $R_{mix[i]}$ is the Ratio of benchmark mix "i" and B is the number of benchmark mixes simulated. This value now quantifies the overall performance of a multicore processor, and it can be used to compare two or more microarchitectures. The one with the highest Geometric Mean is deemed to be the best. This multiprogram evaluation method was used by a lot of studies to evaluate or compare multicore processors [1, 2, 3, 4, 5, 6].

3.2 Order Sensitivity

The issue we intend to present in this dissertation is that the current multiprogram evaluation methodology does not account for which core each benchmark in the mix executes on. They consider each benchmark mix once, whereby each benchmark executes on a specific core. However, one can change the order that the 4 benchmarks in a mix are assigned to the 4 cores in multiple ways, to be exact, 4! ways. Figure 3.1 shows a scenario where we have a 4-core processor model for a simulator and we assign 4 different programs on each core. Suppose we use the programs cactuBBSN, Lbm, Blender and Xalancbmk from the SPEC CPU 2017 suite [8]. For the first experiment we are assigning the programs in the order shown on Figure 3.1 on the left side processor and for the second experiment using the exact same programs we assign them in a different order on the cores shown on Figure 3.1 on the right side processor. These two experiments run the exact same 4 programs, on the same multicore processor, however the results may differ. By results I refer to the individual IPC values of the programs. Figure 3.2 shows an example of what the 24 IPC values of each benchmark look like when changing the order they are assigned on the cores all the possible ways.

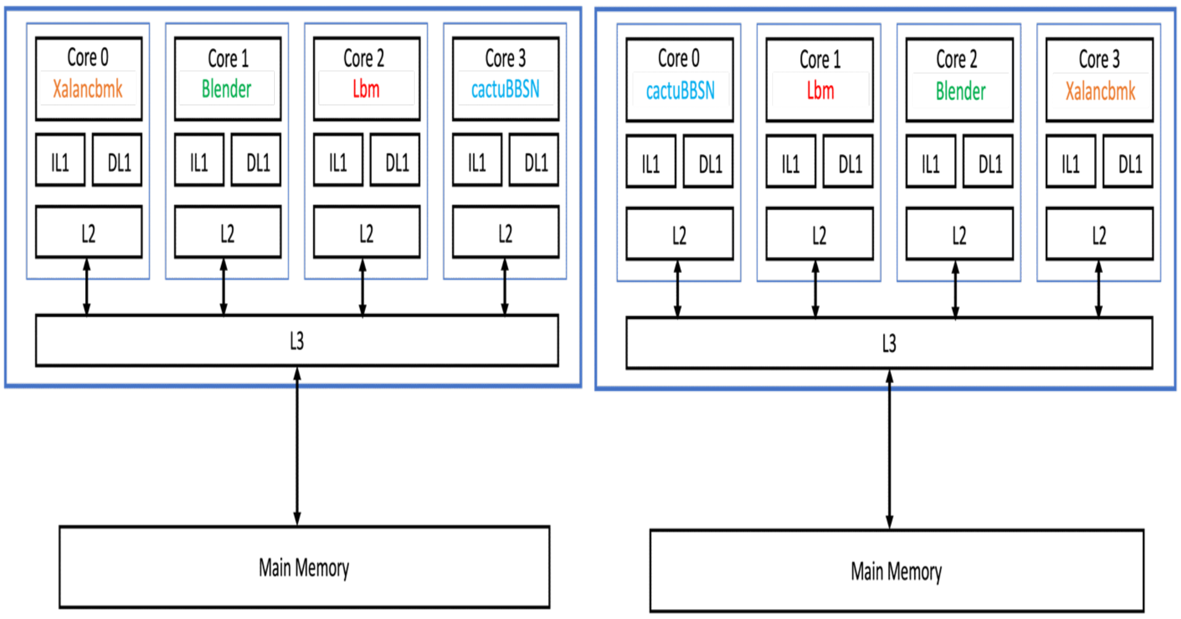


Figure 3.1 Two Multicore CPUs with different orders of benchmarks assigned

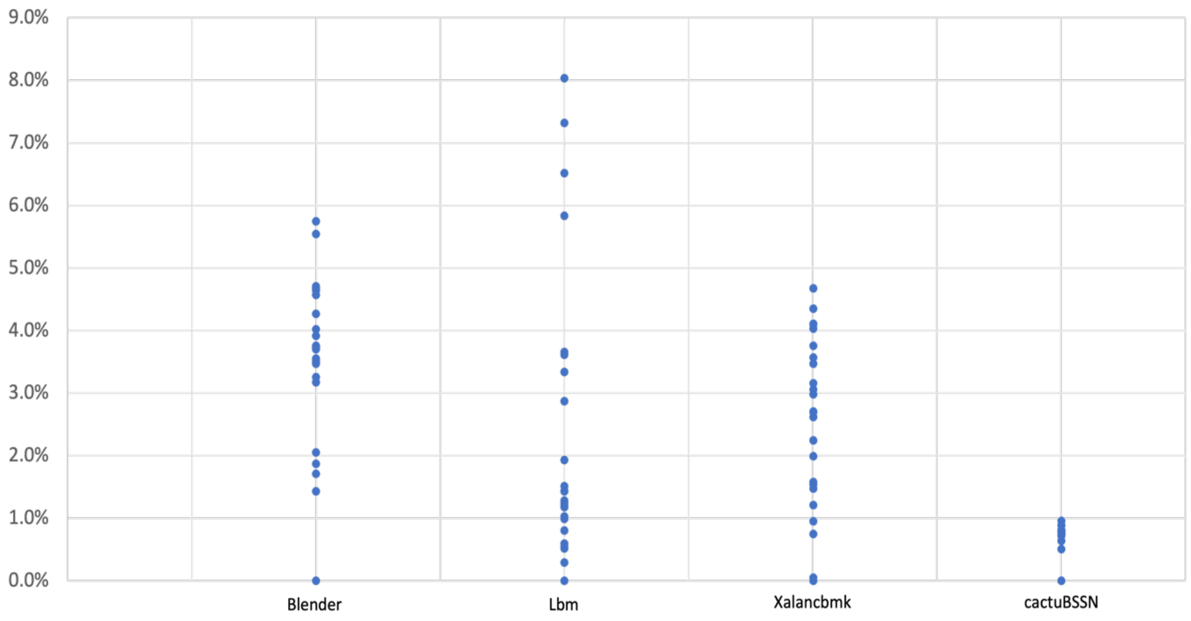


Figure 3.2 Ratio of 24 IPCs over the minimum IPC of the 24 orders for each benchmark

In Figure 3.2 we have ratio of the 24 IPCs over the minimum IPC of the 24 orders. Equation 3.4 show the formula I used to calculate the values shown in Figure 3.2. We can observe that each benchmark varies differently in performance. For example, cactuBBSN is affected the least from changing the order but we can see that Lbm has an IPC that is higher 8% over the minimum IPC which is significant.

$$Ratio_{IPC_i} = \frac{IPC_i}{min(IPC)} - 1$$

Equation 3.4 IPC Ratio over minimum IPC

In general, for a C-core processor we have C! experiments for each benchmark mix we must simulate and for a set of N benchmark mixes we have C!^{N} experiments to evaluate, in other words C!^{N} different Geometric Mean values. This means we have a difficult problem to solve due to the large number of experiments we would have.

3.3 Order-Aware Multicore Evaluation Methodology

Due to the observation in Figure 3.2 we decided to propose a new multiprogram evaluation methodology, the Order-Aware Multicore Evaluation Methodology, that takes into consideration the orders of each benchmark mix and evaluates the range of performance a processor may have. So, we use a set of B benchmark mixes and for each benchmark mix we run the C! orders. For each processor we want to evaluate, we obtain B x N x C! IPC values. Then to determine the performance of each mix and order we use the combined metric Weighted Speedup as they use in the current methodology with the difference that we will end up with B x C! Weighted Speedups. Equation 3.5 show the formula of the WS, where $WS_{mix[k,m]}$, is the WS of benchmark mix "k" and order "m",

$$WS_{mix[k,m]} = \sum_{i=0}^{n} \frac{IPC_{k,m,i}^{MP}}{IPC_i^{SP}}$$

Equation 3.5 Weighted Speedup Formula (With Orders)

and $IPC_{k,m,i}^{MP}$, where is the multiprogram IPC of benchmark mix "k", order "m" and benchmark "i", the "n" is the number of benchmarks the benchmark mix "k" consists of. IPC_i^{SP}, is the single program IPC of benchmark "i" where the benchmark is simulated on a single-core baseline processor. After we calculate the WS of all the benchmark mixes and their orders for the multicore processors we simulate, including a baseline multicore processor, we calculate the ratio of the WS, Equation 3.6 shows the Ratio formula, where $Ratio_{mix[k,m]}$, is the Ratio

$$Ratio_{mix[k,m]} = \frac{WS^{curr}_{mix[k,m]}}{WS^{ref}_{mix[k,m]}}$$

Equation 3.6 Ratio Formula (With Orders)

of benchmark mix "k" and order "m", $WS^{curr}_{mix[k,m]}$ is the WS of the multicore processor to be evaluated of the mix "k" and order "m" over the $WS^{ref}_{mix[k,m]}$, where is the WS of the reference multicore processor of the mix "k" and order "m". The ratio represents how much better or worse is a benchmark mix's WS for a specific order on the current multicore processor compared to the WS of the same benchmark mix and order on the reference multicore processor. After the ratio calculation we end up with 24 ratios for each benchmark mix.

The next step of our methodology is to determine the extremes (minimum and maximum) of the performance range for the set of B mixes and C! orders. We do this using the Equation 3.7 and Equation 3.8. These equations obtain for each benchmark mix the minimum ratio out of the C! orders ratios and the maximum ratio of each benchmark mix and out of the C! orders ratios.

$$minRatio_{mix[k]} = min\left(Ratios_{mix[k]}\right)$$

Equation 3.7 Minimum Ratio out of C! Ratios of each benchmark mix

$$maxRatio_{mix[k]} = max\left(Ratios_{mix[k]}\right)$$

Equation 3.8 Maximum Ratio out of C! Ratios of each benchmark mix

In Equation 3.7 we obtain $minRatio_{mix[k]}$, where is the minimum Ratio of benchmark mix "k" out of C! $Ratios_{mix[k]}$ of benchmark mix "k". In Equation 3.8 similarly we obtain $maxRatio_{mix[k]}$, where is the maximum Ratio of benchmark "k" out of the same C! $Ratios_{mix[k]}$, of benchmark mix "k". After these 2 equations are done we end up with B minimum ratios and B maximum ratios, where B as we said earlier is the number of benchmark mixes.

Then, to aggregate the B minRatios and maxRatios calculated in Equations 3.7 and Equation 3.8 respectively we use the aggregate metric Geometric Mean. Equation 3.9 and Equation 3.10 show the formula of geometric means that determine the extreme values.

$$GM_{min} = \sqrt[B]{minRatio_{mix[0]} * minRatio_{mix[1]} * \cdots * minRatio_{mix[B]}}$$

Equation 3.9 Formula of Geometric mean of minimum Ratios

$$GM_{max} = \sqrt[B]{maxRatio_{mix[0]} * maxRatio_{mix[1]} * \cdots * maxRatio_{mix[B]}}$$

Equation 3.10 Formula of Geometric mean of maximum Ratios

The GM_{min} , represents the minimum performance of the multicore processor we are studying, which is the Bth root of the products of the $minRatio_{mix[B]}$,and the GM_{max}, represents the maximum performance of the multicore processor we are studying, which is the Bth root of the products of the $maxRatio_{mix[B]}$. These two values as I said determine the extreme values of the performance range of the multicore processor.

Throughout the Order-Aware Multicore Evaluation Methodology, we evaluate for the individual IPCs of each benchmark and the Weighted Speedups (WS) of each benchmark their Coefficient of Variations (CV) in order to show that the performance varies within a benchmark mix in addition to overall performance. Equation 3.11 and Equation 3.12 show how we calculate the CV of the IPCs and the WS of each benchmark mix respectively.

$$CV_{bench[k,i]}^{IPC} = \frac{stdev\left(IPC_{bench[k,i,1\ldots C!]}\right)}{mean\left(IPC_{bench[k,i,1\ldots C!]}\right)}$$

Equation 3.11 CV of IPC of each benchmark in a mix

$$CV_{mix[k]}^{WS} = \frac{stdev\left(WS_{mix[k,1\ldots C!]}\right)}{mean\left(WS_{mix[k,1\ldots C!]}\right)}$$

Equation 3.12 CV of WS of each benchmark mix

In Equation 3.11 it's the calculation of $CV_{bench[k,i]}^{IPC}$, which is the standard deviation of the IPCs of all the C! orders of benchmark "i" included in the benchmark mix "k" over the mean of the same C! IPCs. Equation 3.12 is the calculation of $CV_{mix[k]}^{WS}$, which is the standard deviation of the WS of all the C! orders of benchmark mix "k" over the mean of the same C! WS.

So, with our proposed methodology we show only the extreme values (the maximum and minimum performance) of the performance range and not the whole distribution of performance a processor may has. This is far less difficult because we only use C! x N simulations and we don't evaluate all the different $C!^N$ performance values which is a huge number. However, the distribution of the performance would have helped us understand better where most of the geometric mean values rely. Are they closer to the maximum or minimum? Are they distributed in the whole range of performance?

Chapter 4

Experimental Details

4.1 SPEC CPU 2017 Benchmark Suite

The Benchmarks included in the SPEC 2017 package [8] are an industrial standard benchmark suite for CPUs, designed to stress the system processor, memory subsystem and compiler.

The SPEC organization designed this suite to provide a comparative measure of computive-intensive performance across the widest practical range of hardware using workloads developed from real user applications. The benchmarks are provided as source code and require the user to compile the into binaries. In our case, we statically compiled the benchmarks to run on X86 architecture. We compiled them statically in order for them to run as deterministic as possible.

For our study we decided to use 12 of the 23 benchmarks that are included in the SPEC 2017 suite and are the following shown in Table 4.1.

Integer Benchmarks	**Floating Point Benchmarks**
Gcc	cactuBSSN
Mcf	Parest
Omnetpp	Lbm
Xalancbmk	Wrf
	Blender
	Cam4
	Fotonik3d
	Roms

Table 4.1 SPEC CPU 2017 Benchmarks we used divided in Floating Point and Integer

We selected those 12 benchmarks under the criterion of having LLC Misses Per kilo Instructions more than 1 simulated on a baseline single-core configuration. For the real machine tests, we run the compiled binaries until completion of the benchmarks. Using the 12 benchmark above we created a set of 50 benchmark mix randomly.

4.1.1 Traces Description

For simulation tests, we created our own traces for the 12 benchmarks we selected using the PIN Tool [9] champsim_tracer which is included in the Champsim Simulator repository and strongly suggested by the Champsim team. For each benchmark we created the single-program trace of 500 million instructions. We traced the 500 million instructions of the representative region of each benchmark, so that means for each benchmark we skipped different number of instructions. Each trace contains the information we can see on Table 4.2. This information is written in the output file in binary format so that the compression ratio of the trace file its high, that's because the Champsim simulator only accepts trace file that are compressed in gz or xz format, we compressed our traces in ".gz" format.

Instruction pointer (Program counter) value
If the Instruction is branch or not
If branch contains if taken or not taken
Destination registers (output registers)
Source registers (Input registers)
Destination Memory Address (Output memory)
Source Memory Address (Input memory)

Table 4.2 Trace file output of each Instruction

4.2 Champsim Simulator

We use the Champsim simulator which is a trace-based simulator for microarchitecture study. This simulator was used in various contests, such as the 3rd Data Prefetching Championship [10] and the 2nd Cache Replacement Championship [11]. This simulator models a multi-core out-of-order processor.

4.2.1 Champsim Configuration

We modeled a 4-core out-of-order processor with 3 levels of cache. The configuration we used is described in Table 4.3. For LLC replacement policy we used the following policies: LRU,SHIP and SHIP++.

Parameter	Configuration
L1 I-Cache (Private)	32KB, 64B blocks, 8-way 8 MSHRs, 1 cycle latency LRU Replacement Policy
L1 D-Cache (Private)	32KB, 64B blocks, 8-way 8 MSHRs. 4 cycles latency LRU Replacement Policy Next-Line Prefetcher
L2 Cache (Private)	256KB, 64B Blocks, 8-way 16 MSHRs, 8 cycles latency LRU Replacement Policy IP-Based Stride Prefetcher
L3 Cache (Shared)	2MB per core, 64B Blocks, 16-way 32 MSHRs, 20 cycles latency Replacement Policy can be specified
Frequency	4GHz
Fetch, Decode, and retire	4 wide
Execution	6 wide

DRAM	2 channels (1 DIMM per channel) 8 banks (64MB per bank) 8 ranks (512MB per rank) 4GB per DIMM
DRAM I/O Frequency	800MHz
Branch Predictor	Perceptron
Reorder Buffer Size	256

Table 4.3 Champsim Simulator Configuration

4.2.2 Experiments Specifications

In this subsection, we describe how we did our simulations on champsim. It's divided in two parts, the single-core experiments that was used as reference to calculate the Weighted Speedups and the multi-core experiments which we use for our multiprogram evaluation methodology.

For the single-core experiments we used the configuration described in Table 1. The LLC cache baseline size is 2MB and this is multiplied by the number of cores, so for a single-core processor the LLC size is 2MB. Also, we chose the LRU replacement policy for the LLC cache to be our reference. We run each benchmark trace for 500 million instructions with warm-up 100 million instructions.

For the multi-core experiments the configuration is described in Table 1. For LLC replacement policies we used LRU, SHIP and SHIP++. For each replacement policy we run the 50 benchmark mixes we randomly created. For each benchmark mix we run the 24 orders that the Order-Aware multiprogram evaluation methodology suggests. So, for each policy we run 1200 experiments (50 mixes x 24 orders). Each benchmark runs for 500 million instructions with 100 million instructions warm-up. If any benchmark finished faster than the others, the fast one will continue its execution in order to provide pressure to the shared resources. When all the benchmarks complete executing 500 million instructions the simulation is complete.

4.3 Real Machine Specifications

The real machine we use has the Intel Core i5-2400 4-core CPU. The specifications of the machine is described in Table 4.4. We wanted to use 4-core single socket processor so that way we change the orders of the benchmarks within the same socket because this is what we try to evaluate, the variation we may have between 4 cores that share the same socket. Maybe in future work we evaluate the variation when changing where the benchmarks executes between 2 sockets

Parameter	Configuration
CPU	Intel Core i5-2400 @ 3.10 GHz
Cores	4 cores Single socket CPU
Architecture	x86_64
OS	Ubuntu 20.04, Kernel 5.11.0-38
Frequency	3.192GHz
L1 I-Cache	32KB per core
L1 D-Cache	32KB per core
L2 Cache	256KB per core

L3 Cache	6MB
DRAM Size	8GB

Table 4.4 Real Machine Specifications

4.3.1 Experiments Specifications

In this subsection, we describe what experiments we did and how we did them on the real machine. For real machines we did 2 kinds of experiments, single-program and multi-program experiments. We did that because we wanted to check if variation in performance happens on single-program runs and how they compare to the multi-program runs coefficient of variation. We didn't need to evaluate the variation of single-program runs for simulations because the simulator for single-program produces the same results no matter how many times we run it.

For the single-program runs, for every benchmark of the 12 we mentioned before we run the compiled binary until it's fully executed. For each benchmark we run the following experiments, we assign it with taskset on only one core and run it 12 times and we do this for all 4 cores. So, in total we have 48 runs (4 cores x 12 runs) for each benchmark. On simulator we didn't had to evaluate the single program experiments because if we run the same experiment multiple times we would get the exact same result, on the other hand on the real hardware there might be variation between the same experiments run multiple time so we do this in order to evaluate how much variation comes from the core itself.

For the multi-program runs, for 21 benchmark mixes of the 50 we created we run the following experiments, for each benchmark mix we run the 24 orders for 4 times each order. Also on simulator we didn't run multiple times each benchmark mix because as I said for the single-program experiments the same applies to the multicore, if we run the same order of a benchmark mix multiple times we will get the exact same results. The order in which we run the experiments is first we run each of the 24 orders and when they all finish, we continue to the next repetition of the 24 orders until we finish 4 repetitions. The multi-program runs take a long time to complete, taking an average of 24 hours per benchmark mix, so that’s why we were unable to run all the 50 benchmark mixes, also we were unable to complete all of the repetitions we desired. In order to make a more fair comparison, we intended to run 12 repetitions for each benchmark mix to match the single-program runs.For all the experiments we ran, we collect the statistics with the perf stat tool. Also, the statistics perf stat collects contain user and system statistics in our case. The IPC value we use in our work we calculate it using the Instructions and Cycles perf stat counts.

Chapter 5

Experimental Evaluation

5.1 Simulations Evaluation

Our first analysis on the simulation results was to evaluate the coefficient of variation (CV) of the 24 IPCs (From the 24 orders) of each benchmark included in a benchmark mix. So, for each benchmark mix we show 4 CVs of the IPC. Figure 5.1 shows CV of the IPCs of each benchmark in a mix for the all the 50 benchmark mixes we ran for SHIP++. For a benchmark mix each column represents each benchmark of the mix. For example, the first 4 columns represent the 4 CVs of IPCs for the benchmark mix Cam4-Fotonik3d-Lbm-Xalancbmk, the 1st column is the CV of Cam4, the 2nd the CV of Fotonik3d, the 3rd the CV of Lbm and the 4th the CV of Xalancbmk. The goal of this analysis is to show if changing the order of a benchmark mix can affect the performance of the individual benchmarks of the mix. So, based on the figure below we can see that for most of the benchmark mixes there is coefficient of variation which indicates that the performance of the benchmarks is affected by changing on which cores they are assigned.

Figure 5.2 shows the same CV values we showed on Figure 5.1 but for this graph we grouped the CVs by benchmark instead of by benchmark mix. The goal here is to show that every benchmark has variability in performance. This figure is sorted in descending by the average CV of each benchmark. We can observe from Figure 5.2 that every benchmark has variation in performance and this variation is different among the benchmarks but also different within the benchmarks, i.e., Lbm has the highest CV on average and the various CV values of Lbm range from 0.002 to 0.02. This means that every benchmark behaves differently when changing the order of the benchmark mix, also, it means that each benchmark have different variations when running in different benchmark mixes.

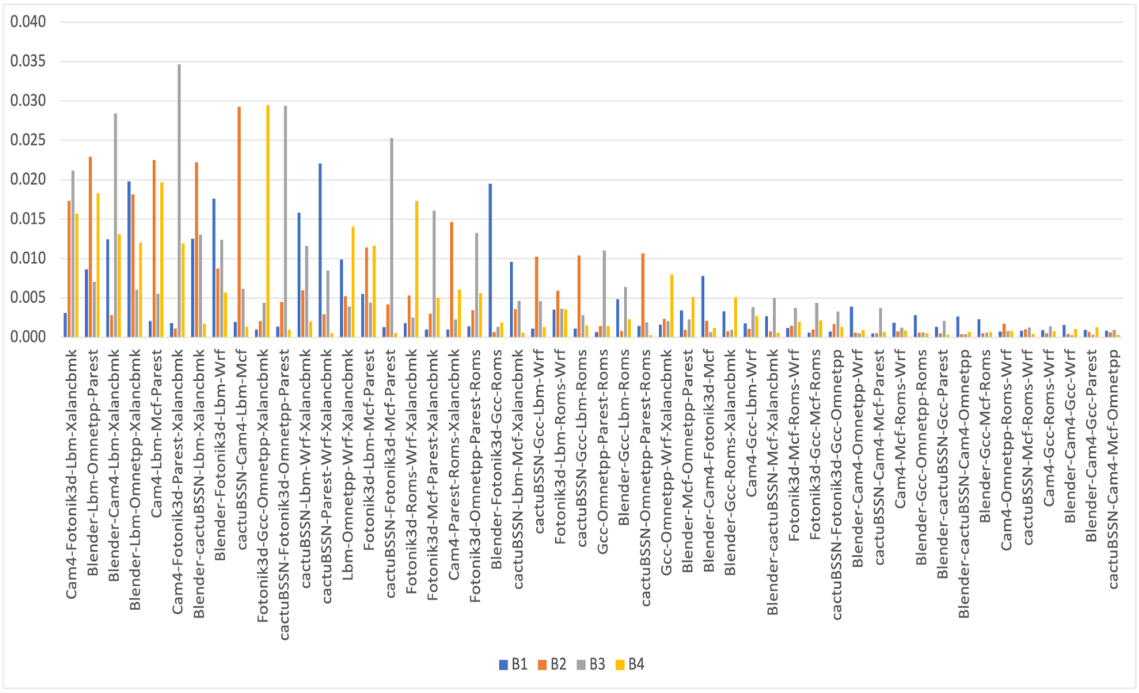


Figure 5.1 CV of the IPCs grouped by benchmark mix for SHIP++

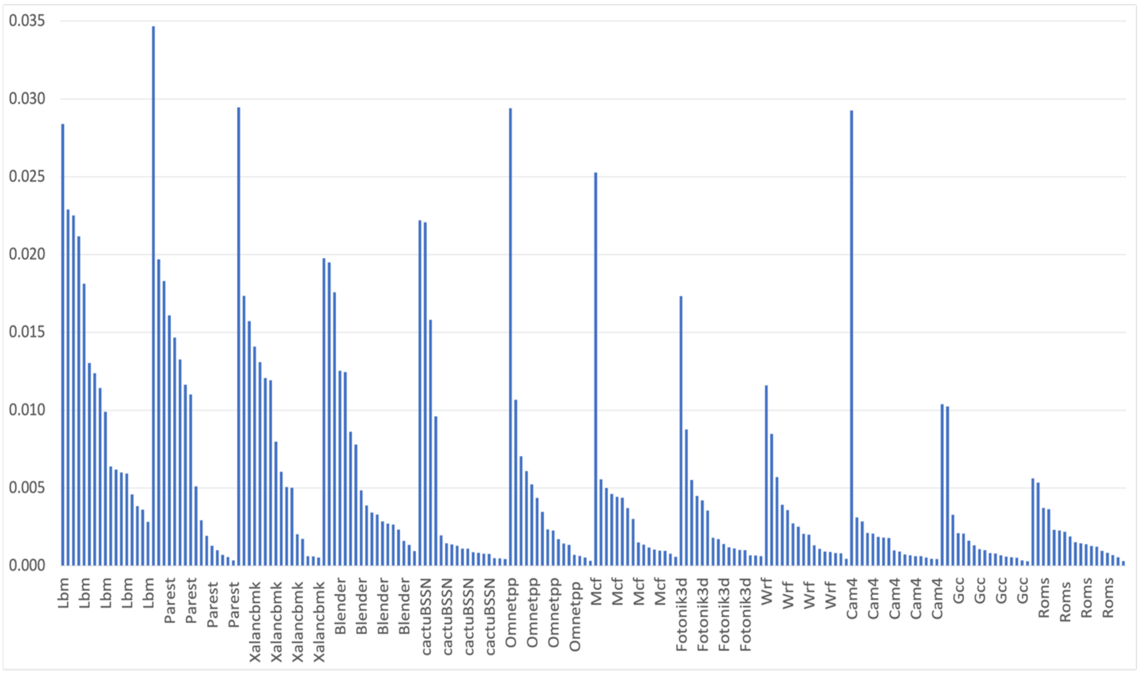


Figure 5.2 CV of the IPCs grouped by Benchmark for SHIP++

We also wanted to evaluate the CV of the Weighted Speedups of the 50 benchmarks in order to show the performance variability a benchmark mix has among the 24 orders. Figure 5.3 shows the CV of the 24 Weighted Speedups of each of the 50 benchmark mixes. We can observe from Figure 5.3 that each benchmark mix has different Coefficient of Variation, which means that each benchmark mix is affected more or less from changing the order of the benchmarks in it. We can also see that the CV of the Weighted Speedup is on the same range as the CV of the individual IPCs.

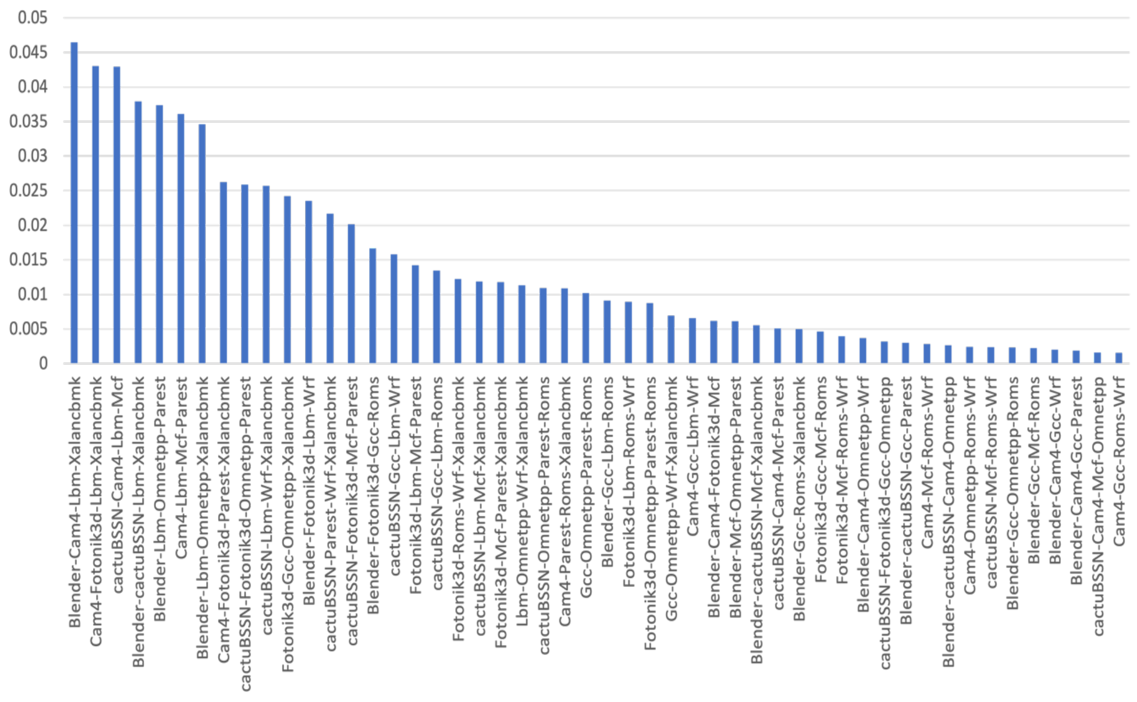


Figure 5.3 CV of 24 Weighted Speedups of each benchmark mix

This next analysis on the simulation results is to answer the second question we asked. For this evaluation we calculated the weighted speedup as we explained in the Chapter 3 for the 50 benchmark mixes for LRU, SHIP and SHIP++. Using the Order-Aware multiprogram evaluation methodology we explained in Chapter 3, with Figure 5.4 we show the GMmin and GMmax as a range for SHIP (The First Range) and SHIP++ (The 3rd Range), the 2nd and 4th performance ranges is the GMmin and GMmax of SHIP++ using the orders we selected as min and max from SHIP and the other range is the GMmin and GMmax of SHIP using the orders we selected as min and max from SHIP++ respectively. The arrows represent the GMmin and GMmax and you can see that the second range is using the same orders as the first range of performance and the GMmin is greater than the GMmax, also the 4th range is using the same orders as the 3rd range of performance. Firstly, we can see that each policy has different range of performance. For SHIP the range is small, from 5.81% to 6.12%, on the other hand, the range of SHIP++ is a lot larger, from 5.99% to 7.57%. Also, we can see that the when using the max and min orders of the other policy the ranges are negligible. After all, as we can see because the ranges of the 2 policies (1st and 3rd Ranges) overlap this means the Order-Aware evaluation methodology can lead us to wrong conclusions on which policy is best. There is a case that SHIP has the performance of 6.12% and a case that SHIP++ has the performance of 5.99% which means if we had these data, we would say that SHIP is better and there are other cases where we would say that SHIP++ is better.

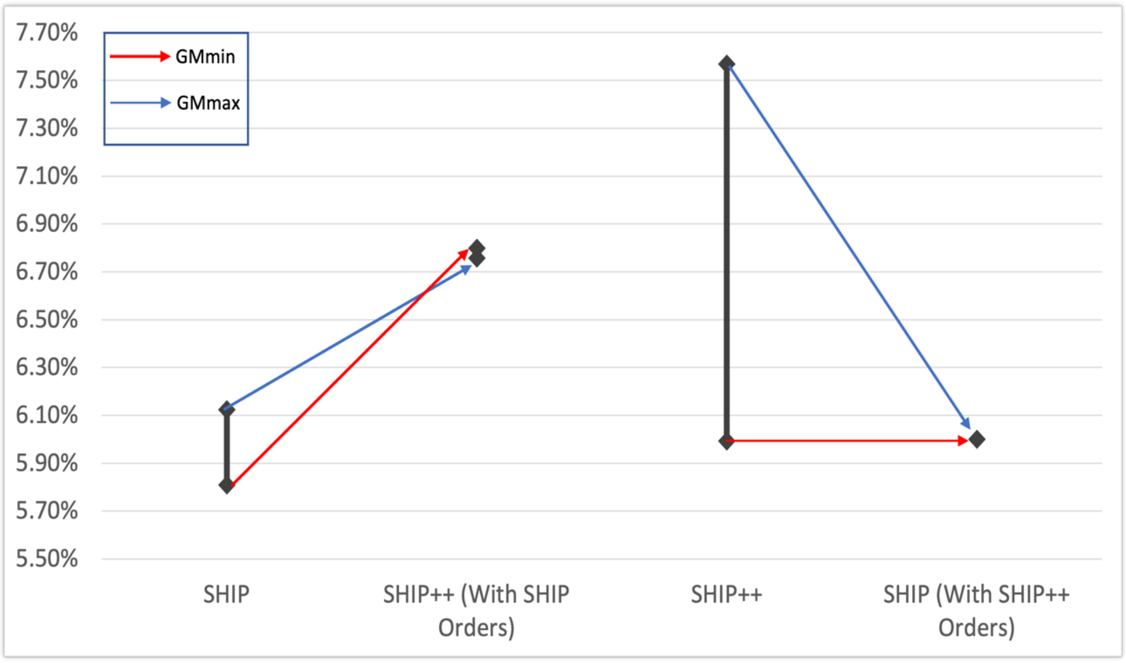
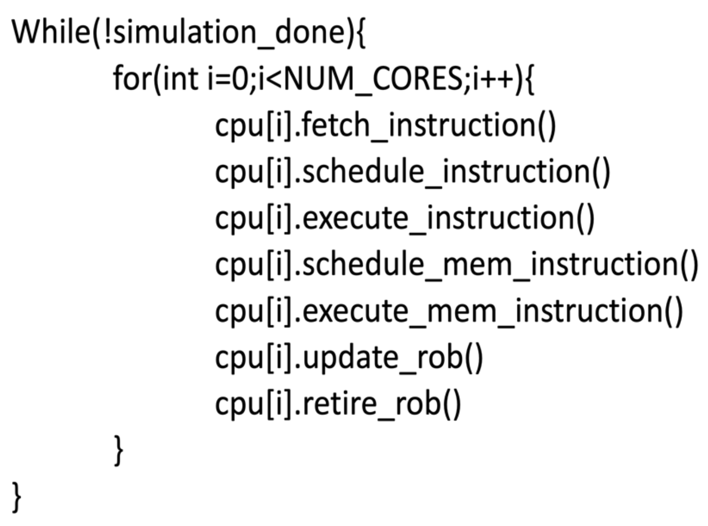


Figure 5.4 Range of Performance for SHIP and SHIP++

For our next analysis we wanted to figure out what causes this variation. Our first guess was that this variation happens because of the simulator’s implementation. The Champsim simulator’s main loop that iterates through all the pipeline stages for each core, iterates through all the cores in a fixed order, the physical order, from core 0 to 3 each cycle. This may cause the variation because if a memory-intensive benchmark is assigned to core 0, every cycle it will request data from the shared resources first, in result most of the LLC requests queue will be core’s 0 requests, this will slow down the fulfillment of the other cores requests. Now if the order of the benchmark mix changes and the memory-intensive benchmark is assigned on core 1 it will request data from the LLC second, this will give a chance to the benchmark that is assigned to core 0 to request data from LLC prior to the memory-intensive benchmark. Figure 5.5 show how the main loop is implemented in the Champsim simulator. Figure 5.6 shows how we changed the implementation of the main loop. We made a list that contains the numbers from 0 to 3 (for our case that we have 4 cores) and each cycle we shuffle this list randomly and the core we are going to serve is selected from this list. This way, each cycle we serve the cores in a different way than the previous cycle.



```
While(!simulation_done){
        for(int i=0;i<NUM_CORES;i++){
                cpu[i].fetch_instruction()
                cpu[i].schedule_instruction()
                cpu[i].execute_instruction()
                cpu[i].schedule_mem_instruction()
                cpu[i].execute_mem_instruction()
                cpu[i].update_rob()
                cpu[i].retire_rob()
        }
}
```

Figure 5.5 Implementation of Champsim's main loop

```
cpu_list = [0,1,2,3]
While(!simulation_done){
        Random_shuffle(cpu_list)
        for(int j=0;j<NUM_CORES;j++){
                i = cpu_list[j]
                cpu[i].fetch_instruction()
                cpu[i].schedule_instruction()
                cpu[i].execute_instruction()
                cpu[i].schedule_mem_instruction()
                cpu[i].execute_mem_instruction()
                cpu[i].update_rob()
                cpu[i].retire_rob()
                }
}
```

Figure 5.6 Our implementation of Champsim's main loop (Our code is highlighted in red)

After we implemented this, we ran again all the experiments that our evaluation methodology suggests for SHIP and SHIP++ in order to check if the variation in performance is less than before. Figure 5.7 and Figure 5.8 shows the average CV of the 4 benchmarks of each mix for the Fixed Order implementation (Champsim's implementation) and for the Random Order implementation (Our implementation) for SHIP and SHIP++ respectively. These graphs are presented as sigma curves graph, in other words, is in ascending order based on the Fixed Order values. As we can observe our implementation did not help reduce the variation of performance for both policies, there are as many degradations as improvements.

We wanted to ensure that our implementation is correct. So, we run an experiment and we counted how many times each order the cores were served occurred. Turned out that the random function we implemented was not biased, each order occurred the same times the randomness of our implementation works correctly.

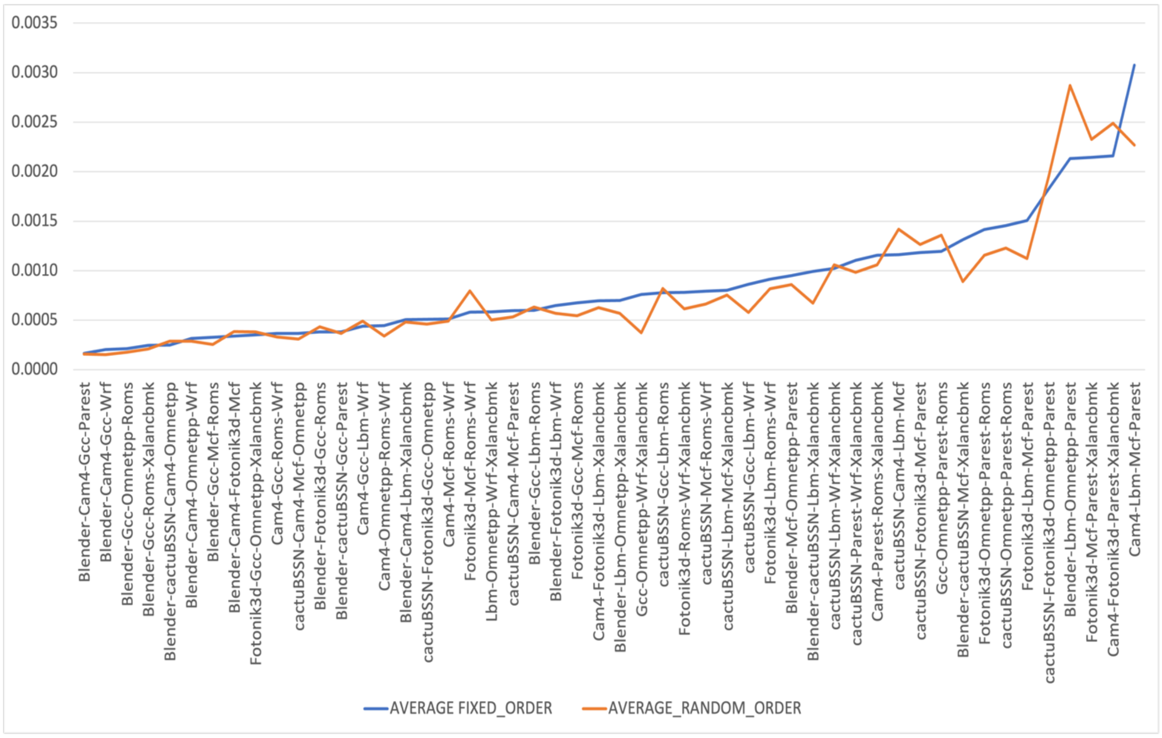


Figure 5.7 Sigma Curve Graph showing the average CV of benchmark mixes for Fixed and Random Implementation for SHIP

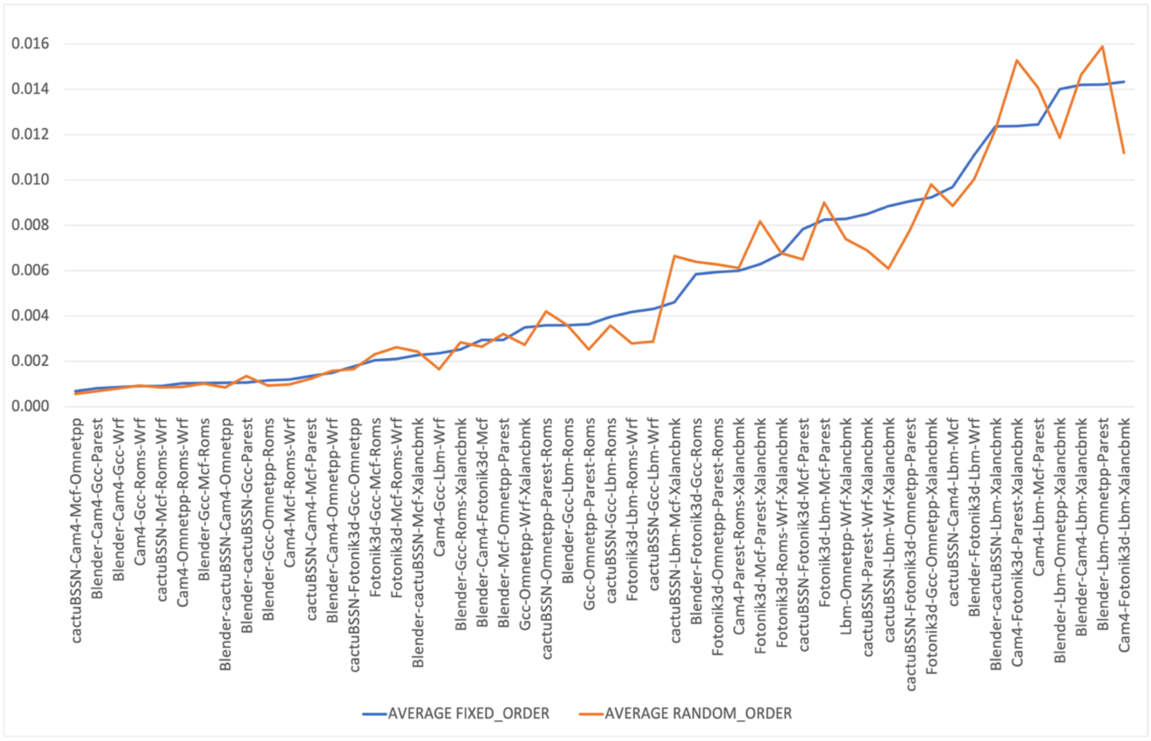


Figure 5.8 Sigma Curve Graph showing the average CV of benchmark mixes for Fixed and Random Implementation for SHIP++

We also compare the range of performance of SHIP with Fixed Order versus SHIP with Random Order and for SHIP++ also. For reference to calculate the weighted speedups for the Random Order results we used the LRU with Fixed Order implementation. We used the same reference for Fixed and Random order implementations because we wanted them to be comparable. From Figure 5.9 and Figure 5.10 we can observe that the range of performance remains the same for both of the policies as we expected because as we saw on Figure 5.7 and Figure 5.8 neither the individual IPCs changes significantly.

In order to conclude the simulations evaluation, we wanted to give an explanation on the variation of performance using the simulator statistics such as LLC misses and LLC hits. We didn't managed to have an explanation yet but it's in our roadmap on doing it in the future.

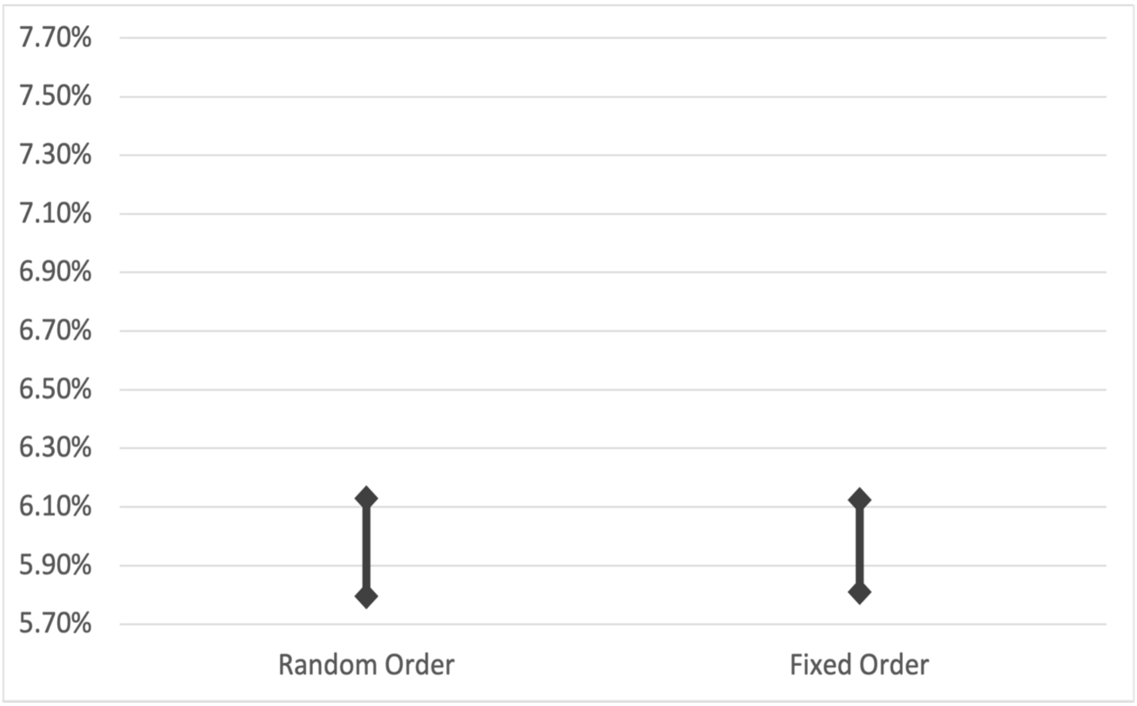


Figure 5.9 Comparison of Range of Performance for Fixed and Random Implementation for SHIP

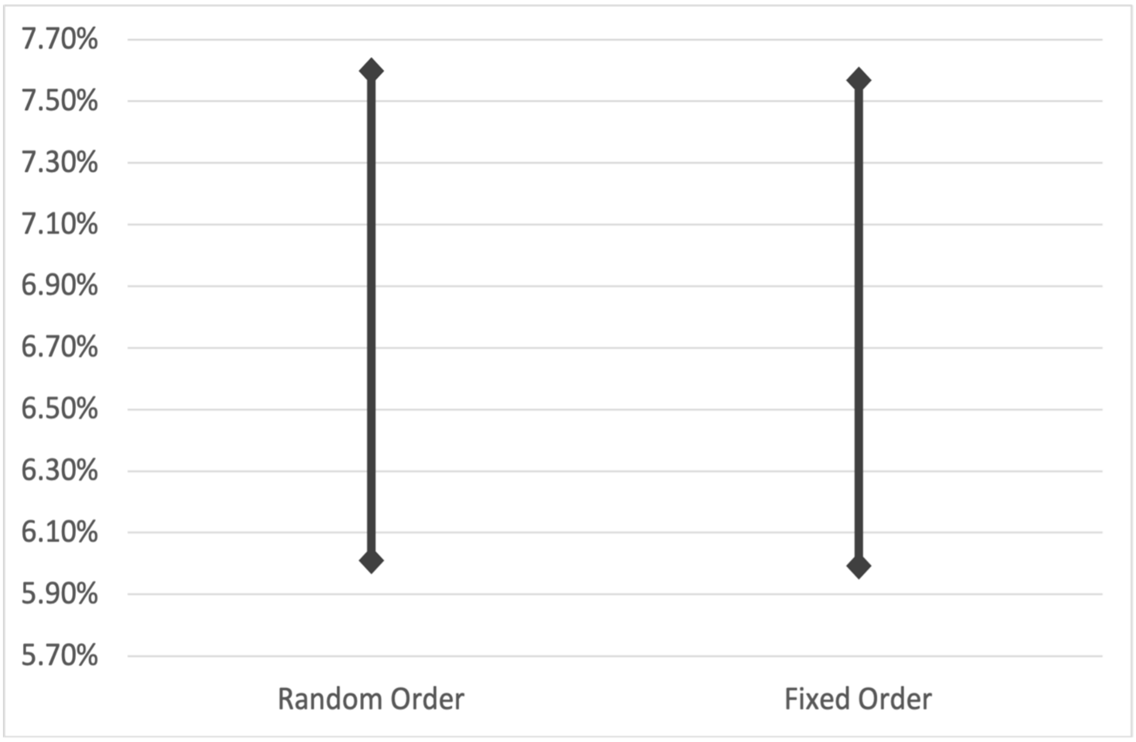


Figure 5.10 Comparison of Range of Performance for Fixed and Random Implementation for SHIP++

5.2 Real Machine Evaluation

Our first analysis on the real machine multiprogram results was to evaluate the coefficient of variation of the Total time, total time being the user time plus system time of each benchmark included in a benchmark mix. For the real machine, we run each order of every benchmark mix 4 times. So, for each benchmark mix we have 96 Total time values (24 orders x 4 iterations). So, we calculate the CV of these 96 values and we show in Figure 5.11 the CV of the benchmarks for the 21 benchmark mixes we run on the machine. We can observe that there is CV on Total time which means that changing the order may affect the performance of the benchmarks included in a mix running on a real machine. To tell for sure that this variation comes from changing the order we have to investigate it more. Throughout this evaluation we will investigate it.

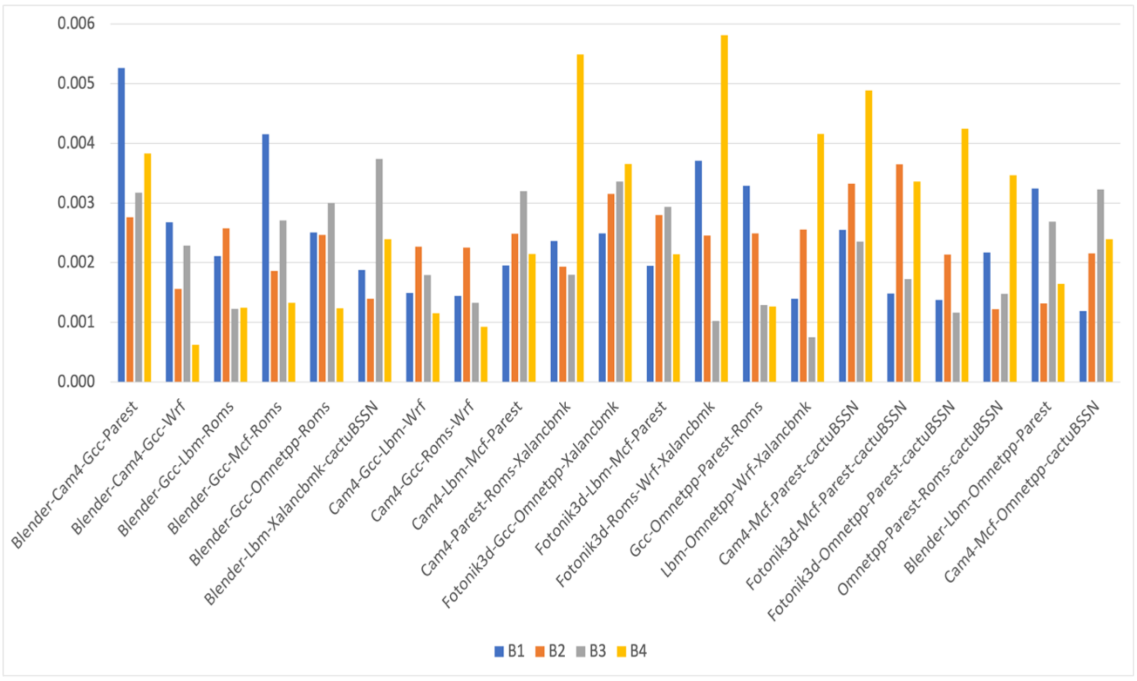


Figure 5.11 CV of Total Time for Real Machine runs (Total of 21 benchmark mixes) grouped by benchmark mix

Figure 5.12 shows the same CV as before but grouped by benchmark. This is the same analysis as we did for the simulator evaluation. The graph is sorted in descending by the average CV of each benchmark. The goal here is to see if on real machine we can observe if among the benchmarks we have different CV and if within the same benchmark we observe different CVs. From the Figure we can see that this is factual, for example Xalancbmk has the highest average CV and Wrf has the lowest average CV, Blender's CV values range from 0.0019 to 0.0053. So, we can say that the variation of the benchmarks is affected by the order and the variation is different when a benchmark is running in different mixes.

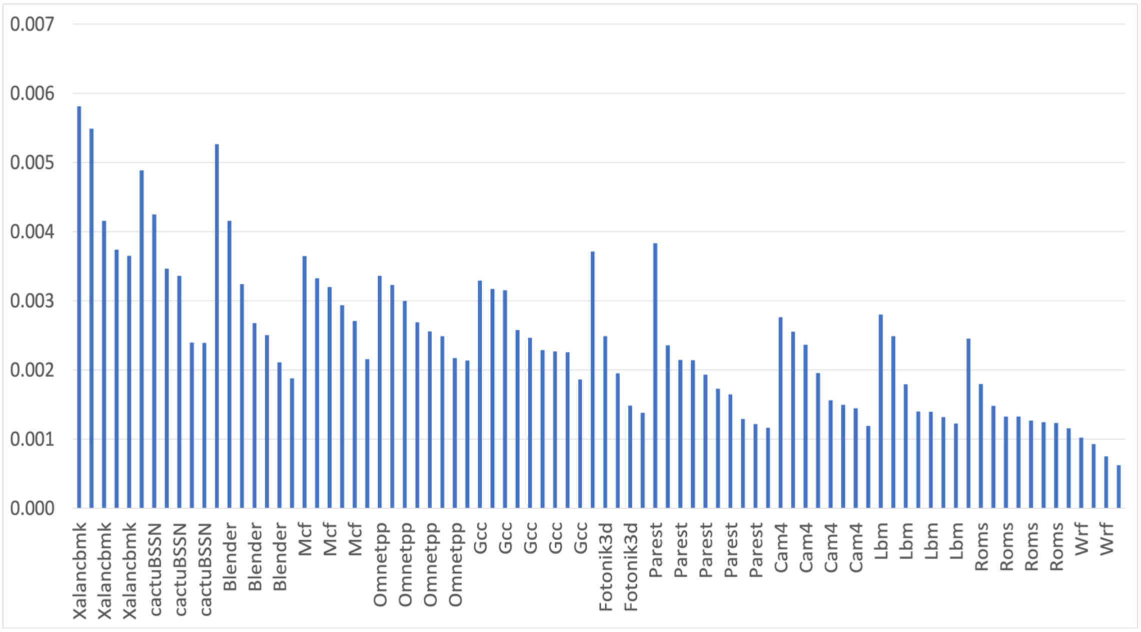


Figure 5.12 CV of Total Time for real Machine runs grouped by Benchmark

For our next analysis, we evaluate the CV of the Total time for the single program results on the real machine for the 12 benchmarks that were used to create the benchmark mixes versus the multiprogram CV of total time. For the single program analysis we ran each benchmark on the 4 different cores for 12 times. So in total for each benchmark we have 48 runs. We calculated the CV of each benchmark removing the minimum and maximum value from each 12 runs of each core, this means we removed 8 values in total. For the multicore we took the average CV of each benchmark in order to compare it to the corresponding benchmark single core CV. For the multicore results we didn’t removed the minimum and maximum of each order because we only have 4 runs per order so we thought that there would not be sufficient data to make comparisons. Figure 5.13 shows the CV of multicore runs vs single runs for each benchmark. We can see that for 8 of the 12 benchmarks the multiprogram CV is more than the single-program CV. With that we can say that some variation comes from just because the single-program runs have CV but some variation comes from changing the order of the benchmark mix in multiprogram experiment.

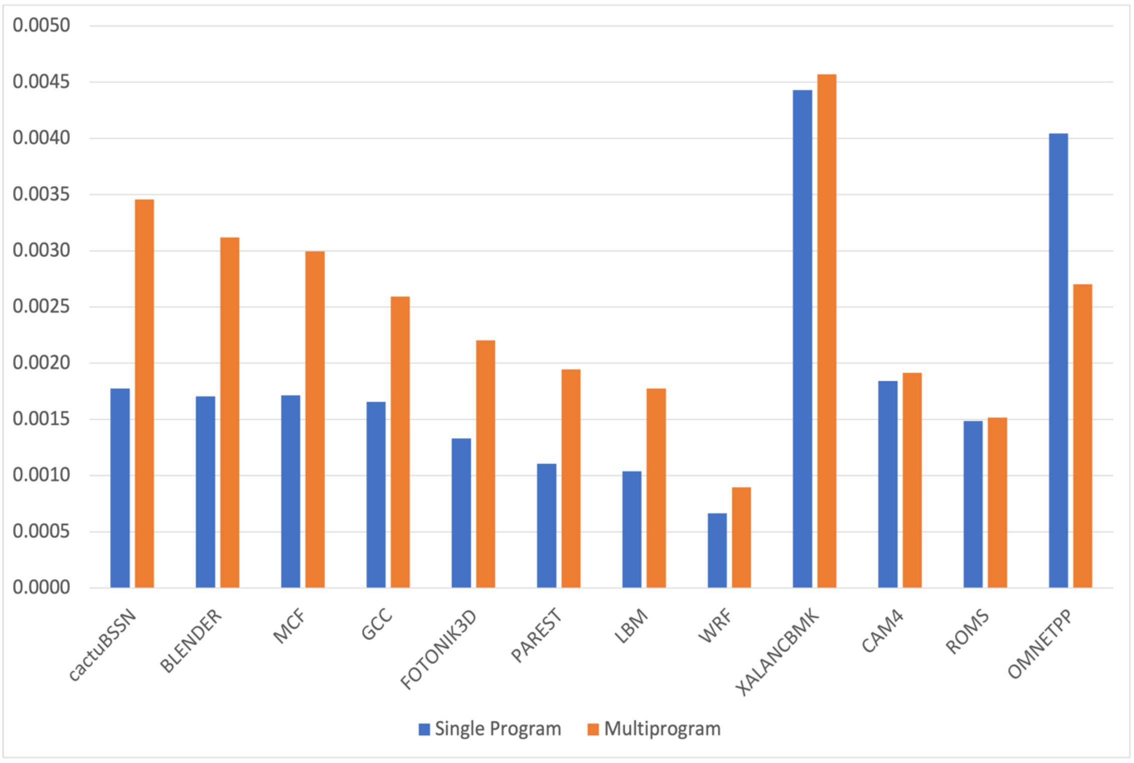


Figure 5.13 Total Time CV Comparison of Single Program vs Multi Program runs

The following analysis aims to prove that the variation of total time we observe in single core experiments comes from the variation of the IPC and the Cycles and not from the variation of the Instructions. We collected these statistics with from the perf stat output and calculated the CV of every one statistic. Figure 5.14 shows for each benchmark the CV of the IPC, Cycles, Total Time and Instructions. As we can observe the CV of IPC, Cycles and Total Time are the same for every benchmark, on the other hand, the CV of Instructions is zero in comparison with the other CVs. So, this observation tells us that the variation we see in Total time comes from the change of IPC and cycles while the instructions executed remains the same. The 3 benchmarks Gcc, Cam4 and Wrf has a small variation in Instructions but it's because they use the system more than the other benchmarks, so this variation in instructions come from system instructions. For example, Gcc is the c language compiler which opens a lot of large files and process them so it uses a lot of I/O. The next step is to investigate this more by analyzing the other statistics like the LLC cache misses in order to understand why this variation in Total time happens.

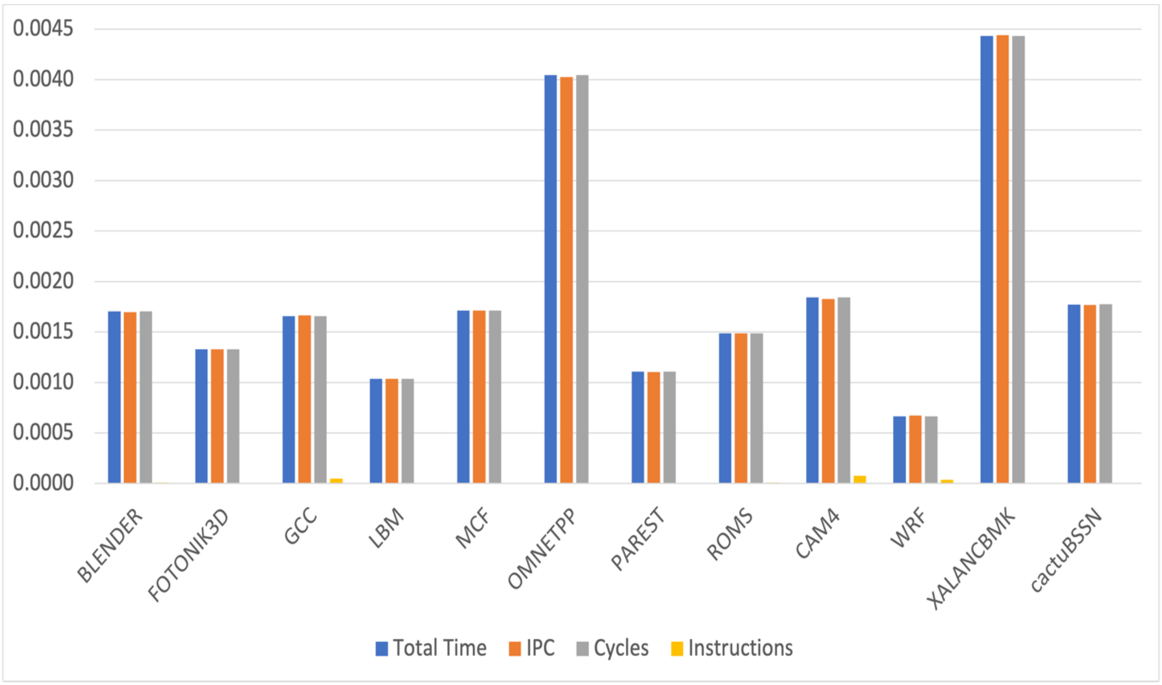


Figure 5.14 CV of Total Time, IPC, Cycles and Instructions for Single Program runs

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In the present thesis, by studying a multicore processor, we were able to evaluate how the performance of a benchmark or benchmark mix gets affected by changing on which core each benchmark will execute on. Using the Order-Aware multiprogram evaluation methodology appeared that the order affects a lot the overall performance of a multicore CPU giving us a range of performance that can vary significantly, up to 1.26% the maximum over the minimum for SHIP++.

Hence, we think that it's important one that studies multiprogram simulations to use our evaluation methodology in order to come to conclusions with more accuracy than before now that will have a range of performance and not a singular performance value of a processor configuration. We think it is not safe to make a decision about which processor is best when only simulating a benchmark mix only once and not consider the other C! ways that it can be simulated.

As a case study, we compared two LLC replacement policies, SHIP and SHIP++ and we showed that our multiprogram evaluation methodology can lead us to wrong conclusions about which policy is best even if we known from previous work that SHIP++ outperforms SHIP.

Then, we tried to implement a random version of how the simulator serves the cores each cycle. We run again all the experiments with the new implementation to check whether the variation gets lower, but we showed that the variation does not change. That means that the variation we saw on the simulator was not caused by the biased implementation of the order they serve the cores but because the order we run a benchmark mix affects the performance.

After showing the variation of performance on simulators, we tried to use the Order-Aware multiprogram evaluation methodology to show that the same variation happens on real machines also because in the end this is what's important to prove. We showed for one specific real machine that there is variation when changing the order.

To back this up we also compared the variation of multiprogram experiments with single-program experiments variation and showed that the multiprogram variation for the majority of the benchmark were more. This means that the variation doesn't fully come from the single-program variation but also comes from changing the order a benchmark executes in a mix.

It was also important to show for single-program runs that the variation of Total Time correlates with the variation of IPC and Cycles and does not correlate with the variation of instructions. This analysis showed us that we were executing the same instructions each iteration, but the performance changed. However, for this analysis we need more multiprogram runs to establish our conclusions clearly.

6.2 Future Work

Our goal is to continue and evolve the Order-Aware evaluation methodology in order to have an accurate methodology to evaluate or compare two or more multicore processors.

In the future we want to evaluate not only the minimum and maximum performance but all the distribution of the performance. This will help us more to have a better image about the overall performance.

One important point of future work is to analyze the simulators statistics in order to explain why the variation happens. Our guess on why the variation happens is because of different timing of events among the different orders. We want to visualize this difference in timing by calculating for aech load instruction how many cycles it was in the pipeline, in other words, how many cycles each load instruction took to commit for different orders. To isolate this, we will break down the load instructions to load that was miss in L2 or hit in L2 for example, to check if there is a pattern that explains why the order the benchmarks run affects the performance. This understanding of why the order the benchmarks run affects performance will maybe help us to understand some characteristics of some benchmarks that will then benefit us assigning them on a specific order to achieve better performance.

Another way to understand why the variation happens is to create a micro-benchmark of our own that targets to pressure resources like the LLC or the Main Memory. Then we will run this micro-benchmark with other benchmarks and simulate all the possible orders. This way we have more control of what the micro-benchmark do and if something changes in the statistics we will isolate it easier and explain why this variation happens.

Another point of future work is to run all the necessary experiments to complete the multiprogram runs in order to conclude if the real machine has variation or not. Also, for real machines we want to compare two different machines that is known the one outperforms the other, and using our evaluation methodology, evaluate the performance of the two and whether it can change the outcome of which of the 2 machines is best.

We also want in future to evaluate other microarchitecture simulators with the Order-Aware multiprogram evaluation methodology to check if it also happens that the order the benchmarks run affects the performance on these other simulators.

Finally, for future progress of our methodology we want to address the problem that our evaluation methodology is not scalable for more than 4 cores because the permutations are C! which gets huge for 8 cores and more. For example, for 8 cores we will have to run for each benchmark mix around forty thousand permutations which we can understand it's impossible. We want to find a way to reduce or eliminate the factorial of the equation so our evaluation methodology can be scalable. One idea is to build a genetic algorithm that it’s goal is to search for orders that maximize the variation of performance. This way we won’t have to do all the experiments our evaluation methodology requires but only run the most impactful benchmark mix orders that will meet the criteria we will require in the genetic algorithm for the maximum and minimum performance.

References

[1] C. -J. Wu, «SHiP: Signature-based Hit Predictor for high performance caching,» σε *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[2] R. A. Velásquez, «Selecting benchmark combinations for the evaluation of multicore throughput,» σε *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[3] Young, «SHiP++ : Enhancing Signature-Based Hit Predictor for Improved Cache Performance,» 2017.

[4] A. Jaleel, «Adaptive insertion policies for managing shared caches,» σε *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[5] G. H. Loh, «Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy,» σε *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[6] Y. &. L. G. Xie, «PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches,» σε *ISCA '09*, 2009.

[7] «GitHub - ChampSim/ChampSim,» . Available: https://github.com/ChampSim/ChampSim.

[8] «SPEC CPU® 2017.,» 2017. Available: https://www.spec.org/cpu2017/.

[9] «Pin - A Dynamic Binary Instrumentation Tool,» 2012. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[10] «The 3rd Data Prefetching Championship,» 2019. Available: https://dpc3.compas.cs.stonybrook.edu.

[11] «THE 2ND CACHE REPLACEMENT CHAMPIONSHIP – Co-located with ISCA June 2017,» 2017. Available: https://crc2.ece.tamu.edu.