Enhancing Performance Evaluation and Characterization Techniques to Identify Performance Changes in High-Performance Computing Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Yectli Adolfo Huerta

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

ADVISOR: Prof. David J. Lilja

November, 2021

Acknowledgements

El apoyo y comprensión que mis padres y hermanos me brindaron durante mis estudios fue indispensable. Fue un sacrificio muy grande de su parte y por eso les estoy eternamente agradecidos. Gilberto Vázquez Valle y el profesor Guillermo Rojas fueron fuentes de inspiración durante los años que tuve el placer de convivir con ellos. Gracias por su amistad y apoyo. *Requiescant in pace.*

Professor David Lilja is one of the kindest professors I have ever work with. I first met him as an undergraduate student when he was assigned to be my faculty mentor. He invited me to eat lunch several times. I enjoyed the experience very much. After I was accepted to the Scientific Computation graduate program, professor Lilja invited me to join his research group. After learning about the research his group was doing, I did not hesitate in taking up his offer to be part of his group. His infinite patience, insight and steadfast nature made it possible for me to complete my PhD. I am thankful for his support and guidance. Additionally, I want to thank current and former co-workers of the Minnesota Supercomputing Institute, especially Brent Swartz, Evan Bollig, Andrew Gustafson, Jeff McDonald, and Yuk Sham. Working full-time while completing my degree was no simple task. Their encouragement, insights and depth of knowledge provided a beacon that made it possible to continue to make progress through the years.

My grammar skills have improved greatly with the efforts of Ms. Anna-Marie Byrne. She is patient and kind when reviewing the multiple drafts of this thesis. Last, but not least, I am thankful to the many people that I had the pleasure of meeting along the way at the University of Minnesota. I was able to forge many long lasting friendships that brought joy and hopefulness to me during stressful times. ¡Muchas gracias!

Dedication

A mis padres, gracias por su apoyo en cada etapa de mi vida.

Abstract

High Performance Computing systems are complex and require a lot of effort to tune the system to achieve peak performance. Performance analysis is a time consuming process. The goal of this thesis is to understand the effects changes to the system or compiler configuration had on performance and how it is reflected in CPU performance metrics. This thesis presents two approaches to enhance the evaluation process of HPC systems. First, a process that makes it possible to systematically and efficiently search the parameter space to find an optimal configuration of a benchmark with a large number of tunable parameters is introduced. The search for an optimal combination of parameters can be daunting, especially when it involves high dimensionality of mixed type categorical and continuous variables. This thesis shows that through the use of statistical techniques, a systematic and efficient search of the parameter space can be conducted. These techniques can be applied to variables that are categorical or continuous in nature and do not rely on the standard assumptions of linear models, namely that the response variable can be described as a linear combination of the regression coefficients. Second, a normalization technique that will make it possible to identify relative differences between performance metrics to better understand the effects changes had on the underlying system is presented. The use of Top-Down microarchitecture analysis method makes it possible to understand how pipeline bottlenecks were affected by changes in the system configuration, or compiler version. Bottleneck analysis makes it possible to better understand how different hardware resources are being utilized, highlighting portions of the CPU's pipeline where possible improvements could be achieved. The Top-Down analysis method is complemented with the use a normalization technique from the field of economics, purchasing power parity (PPP), to better understand the relative difference between changes. This thesis showed that system changes had effects that sometimes were not reflected on the corresponding Top-Down metrics. The use of the PPP normalization technique made it possible to highlight differences and trends in bottleneck metrics, differences that standard techniques based in absolute, non-normalized, metrics failed to highlight.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Recent trends in CPU designs have increased the number of cores to increase CPU performance. These multi-threaded and multicore CPUs allow for better hardware utilization [8]. Parallel programming frameworks, such as OpenMP, have made it possible to use highly complex computational resources efficiently. But compilers and programming frameworks have a wide variety of features and options that can make it challenging to obtain optimal performance without some significant effort. It is essential to have analysis tools and techniques that can provide users with precise and detailed information on the compiler's performance to assess whether or not the generated code uses all of the architectural features of the CPU efficiently, as measured in overall runtime. The user could select different compiler flags, modify the code, or select a different compiler version that might further enhance performance. When designing compilers, developers are interested in in-depth profiling to quantify the impact of new features on performance. This entails the use of statistical tools to explore the different possible parameter combinations in an efficient and systematic approach. An efficient search of the parameter space can provide improvements in performance in a more expedient manner since a properly design experiment will narrow the number of possibilities to test.

Performance analysis can be a challenging and time-consuming endeavor. Modern CPUs have complex microarchitectures that improve overall code performance. The

use of deep pipelines, buffers and prefetchers makes it possible to hide and mitigate stalls – delays in the processing of an operation. Stalls can occur when an operation has to wait for needed resources to become available before it can be completed. This microarchitectural complexity makes it difficult to analyze bottlenecks – portions of the code where stalls occur, and that should be examined for a possible change to improve performance, once they are properly identified, and their effects quantified. Features that minimize stalls and improve overall performance will tend to make it difficult to pinpoint such bottlenecks by hiding latencies. For example, a CPU will reduce its data retrieval time by guessing which data value an instruction will need next, or by storing frequently used instructions or memory contents in its caches.

There are three commonly used performance metrics that are used to characterize HPC applications: Flops-Per-Cycle (FPC), algebraic computation rate, Instructions-Per-Cycle (IPC), quantifies useful work is being done for each cycle, Loads-Per-Cycle (LPC), rates memory efficiency in terms of cycles [9]. Care must be taken not to misinterpret metrics values. For instance, it was found that good CPI rates, the inverse of IPC, were not indicative of good performance. It was understood that the reason is that CPI rates are biased toward setups with low core frequencies. At lower frequencies, a smaller number of core cycles are spent in memory access operations, resulting in improved CPI rates [10]. In the same study, the authors demonstrate that improved CPI rates do not translate into better or worse system performance. Focusing only on certain ratios can provide an incomplete picture of what is actually happening in a system. Focusing exclusively in L1 miss ratios can be misleading. An application can have good L1 performance, while at the same time, subpar performance in other levels of the memory hierarchy. L1 and L2 miss rates were not predictive of L3 or DRAM miss rates [9].

Our first contribution makes it possible to optimize performance of an application through better a better search of the parameter space. Users have the need to test multiple options to enhance performance of an application. A rigorous approach to effectively explore different setting configurations includes the use of Design of Experiments [11] to be able to systematically check as many possible configurations as the experimenter's resources will allow. For instance, Plackett and Burman were successfully used to identify key processor parameters, identify the effect benchmarks had on

the processor [12]. But standard experimental designs are not flexible enough to allow for the use of parameters with varying number of settings, such as is the case of categorical variables with multiple levels. We showed that Orthogonal Arrays [13] are a viable alternative to regular experimental designs when multiple categorical and quantitative variables of varying value levels. The HPL [14] benchmark was successfully optimized in a hybrid CPU and GPU environment with multiple computational nodes [15]. An approach that regular standard practices would have had a more difficult time to match.

Our second proposed technique can be used to better understand and discern the information performance metrics provide [16], [17], [18]. The research presented in this thesis highlights subtle differences between performance rates, especially when a change in the code, the underlying system or compiler options were made. Our approach is an extension of best practices. The current approach, first introduced in [1] and later adopted by a CPU vendor [19], involves the comparison of performance metrics after a change has been made to the system's configuration, compiler options or the code itself. The user needs to focus in portion of the code, the so called *Hot Spots*, where the CPU spends a large amount of time. Performance metrics for that portion of the code are computed and then compared to a new set of performance metrics after changes to improve performance are made. This differential analysis approach [20] could potentially highlight the effect a change had on performance through the analysis of different performance metrics. But there are times when performance metric values might be relatively close in value, even when the changes made to the system or the program had an effect on performance. The exclusive use of absolute rates in performance metrics might provide an incomplete picture of the change between the before and after the change was made rates. For instance, a suggested approach for ratio comparison, such as cycles-per-instructions (CPI), is to only compare when *"one part of the ratio is changing"* [21]. This approach is not feasible when changes, such as the number of threads used to run a program, affects both parts of the ratio. In the case of the CPI ratio, the number of CPU cycles and the number of retired instructions – architecturally executed instructions – change when the number of threads is varied. A more practical approach is to normalize rates with a reference baseline to quantify the relative performance metric changes as the number of threads is increased.

Relative changes, normalized with the purchasing power parity (PPP) technique

[22], can provide additional information on the metric drift. There are other techniques such as the Roofline model [23] that gives users an idea of how their code is performing when compared to memory and floating-point peak performance. Other approaches include the use of statistical methods to model performance based on metrics such as cache hit rates and memory latencies [24]. Both of these approaches give users insight into how performance is affected as different settings or features are changed, but they are specific to the parameters that are used in the statistical model, or the few metrics that are used to determine peak performance.

The work presented in this thesis is based on a number of publications. Chapter 3 presents a technique based on Orthogonal Arrays to improve the performance of the HPL benchmark in a hybrid environment made up of CPU and GPUs [15]. Chapter 4 introduces the use of the Purchasing Power Technique as it applies to the Top-Down bottleneck analysis when comparing compiler drift between different versions of the GCC compiler suite [17]. Chapters 5 and 6 discuss the effects scaling an application has on Top-Down metrics and regular performance metrics on the aarch64 architecture when the performance rates are normalized with the PPP technique [18]. Chapter 7 discusses the effects of the Spectre/Meltdown security fixes on a system as analyzed through the use of the Top-Down bottleneck analysis and the PPP normalization technique [16].

We summarize the main contributions of this thesis as follows:

- We demonstrated the use of an asymmetric statistical design to explore the parameter space where other methods might not suffice or required significantly more experiments. Regular designs allow for the use of multiple parameters with the same number of levels. Our approach made it possible to test categorical parameters with multiple settings along side quantitative variables. Something not easily done in commonly used regular experimental design.

- Through the use of PPP normalized rates, we were able to highlight trends and differences in performance metrics that might have otherwise gone unnoticed by standard evaluation practices. This made it possible to better ascertain the effects that changes to a program, compiler options or system configuration had on the performance metric when compared to a baseline value.

Chapter 2

Background

2.0.1 Purchasing Power Parity

One of our goals in the research presented in this thesis was to better understand the effect changes made to a program or a system's configuration have in performance metrics. There might be times when the changes reflected in the values of performance metrics don't fully reflect the underlying values that generated that metric ratio. We use a technique by which performance metric rates could be normalized by a baseline value. We adapted a normalization technique from the field of economics called Purchasing Power Parity (PPP), which is used to compare the purchasing power of different currencies, to the field of performance computing. Instead of comparing currencies, we compare performance metric ratios. This makes it possible to better understand the relative difference between performance metrics, such as the IPC rate, when normalizing with the values of a reference metric, such as CPU cycles, when changes between runs of the same program are made. PPP states that *the exchange rate is proportional to the ratio of price levels in two countries.* [22]. Essentially, it allows for the comparison of the same good in different countries to see if a country's currency is overvalued (when the good is more expensive), or undervalued (when the good is cheaper) relative to another currency.

There are two versions of the PPP theory, absolute and relative. The absolute version of PPP theory is the focus of this work which does not take into account transportation and other barriers to trade in its assessment. Absolute PPP also excludes all non-traded

goods – goods which are not traded in the international market. Clements goes on to describe absolute PPP using Equations 2.1, 2.2, 2.3, and 2.4. The domestic price of good i is represented by P_i. The price of a similar product in a foreign country is represented by P_i^*, priced in terms of the foreign country's currency. S stands for the spot exchange rate. Assuming that there are zero transactions costs and no barriers of trade between the countries, the cost of the similar good, in terms of a common currency, can be expressed as shown in Equation 2.1.

$$P_i = SP_i^* \tag{2.1}$$

The previous equation can be further developed to encompass a country's entire economy. Let w_i and w_i^* represent the portion of good i in the home and foreign countries. The total sum of individual goods will be representative of a country's entire economy, in this case $\sum_{i=1}^{n} w_i = \sum_{i=1}^{n} w_i^* = 1$. These sums can be multiplied to both sides of Equation 2.1, resulting in Equation 2.2. Its left hand side of the equation represents the share weighted average of the n prices at home, which is also known as the price index. The right-hand side of the equation is applying domestic weights to foreign prices.

$$\sum_{i=1}^{n} w_i P_i = S \sum_{i=1}^{n} w_i P_i^* \tag{2.2}$$

The equation can be further simplified by making the assumption that foreign and domestic weights coincide, resulting in $\sum_{i=1}^{n} w_i P_i^* = \sum_{i=1}^{n} w_i^* P_i^* = P^*$, which gives us Equation 2.3, an econonywide interpretation of Equation 2.1. P represents the domestic currency cost of the basket of goods, $\sum_{i=1}^{n} w_i$, while P^* represents the cost of a similar basket of goods in the foreign country in question.

$$P = SP^* \tag{2.3}$$

Equation 2.4 represents the absolute version of PPP. It shows that the exchange rate is the ratio of domestic to foreign prices.

$$S = P/P^* \tag{2.4}$$

The work presented in this thesis focuses on the PPP implications as applied to a specific case: the foreign exchange rate based in terms of the cost of Big Mac burgers in their respective country's currencies as specified by absolute PPP theory, as shown in Equation 2.5.

$$\begin{aligned} Exchange\ Rate = Price\ of\ Big\ Mac\ in\ foreign\ currency \div \\ Price\ of\ Big\ Mac\ in\ local\ currency \end{aligned} \tag{2.5}$$

The Big Mac Index [25] is the most famous application of absolute PPP theory, and it compares the value of Big Mac burgers across many countries. The two conditions required for absolute parity to hold are exhibited in the Big Mac Index. These conditions are the absence of barriers to trade for the ingredients used to make the burger, and the use of a similar basket of goods. The Big Mac Index comparison across countries is possible because the burger itself is viewed as a basket of tradeable ingredients, that include beef patties, cheese, lettuce, onions and bread, that are common across countries. Other non-tradeable goods, such as rent and labor, are reflected in the overall price of the Big Mac, the Big Mac Index makes it possible to just focus on the basket of tradeable ingredients used to make the burger. Equations 2.6 and 2.7 show an example of how to compute the Big Mac Index. Assume that the cost of a Big Mac in the US is $3.57. The same burger costs 7.50 reales in Brazil. The PPP exchange rate is shown in Equation 2.6, which is 2.10. The currency exchange rate at that time was of 1.58 reales for $1. In Equation 2.7, we compute the Big Mac Index by subtracting 1.58 from the PPP exchange rate and then dividing by 1.58. According to the index, the real is overvalued almost 33% as compared to the US dollar [22].

$$7.50/3.57 = 2.10 \tag{2.6}$$

$$(2.10 - 1.58) * 100/1.58 = 32.91\% \tag{2.7}$$

We applied the absolute PPP approach, as implemented by the Big Mac Index, to computer performance metrics, to better understand relative changes in performance as reported by performance metrics of the same program after changes to either the program's code, compiler versions, or system configuration were made. Computer programs, like burgers, can be conceptualized as a collection of operations, ingredients, each

with different costs. These operations can include arithmetic or comparison functions to apply mathematical formulas on datasets. Others can be operations that require the retrieval and storage of data in different components that make up the memory hierarchy of a system. There are also the operations that require communication across CPU cores. All of these operations are translated into machine instructions by compilers that a CPU system can then execute. Each of the operations that make up the program can be assigned a cost in a variety of ways: the number of instructions it took to decompose the program by the compiler, the number of cycles it took to run the program to completion, the number of memory data fetches the program required to run to completion, the number of cache misses the program incurred when fetching data from memory, among many others. These performance measurements are used in performance metrics, ratios of measurements, to determine the efficiency of use, as reflected by the performance measurements, of the different components that make up a CPU. Differences in the performance metrics results would indicate that the change made to the system or the program had a direct effect on the component the performance metric is describing. We used the Big Mac Index to quantify, in terms of a common measurement, the relative difference in the performance metric rate that changes to the system or the program generated.

Equations 2.6 and 2.7 were adapted to use performance metrics to compute the PPP normalized rate. The first step is to identify the baseline performance metric ratio that will be used as the baseline for comparison. The baseline setup or configuration before a change is made. The ratio of the resulting performance metric ratio after a change is made is divided by the baseline performance metric result, Equation 2.8. The ratio of the denominator of the performance metric ratios is also taken, Equation 2.9. Finally, the two resulting ratios are then used in Equation 2.10 to compute the relative difference between the two performance metric ratios in terms of the metric used as the denominator of the performance metric where PM = *Performance Metric*, PMB = *Performance Metric Baseline*, $MPPP$ = *Performance Metric PPP Rate*, ER = *Exchange Rate*, M = *Metric*, MB = *Metric Baseline*, and PNR = *PPP Normalized Rate*.

$$MPPP = PM/PMB \tag{2.8}$$

$$ER = M/MB \tag{2.9}$$

$$PNR = (MPPP - ER) * 100/ER \tag{2.10}$$

A generic example that shows the use of PPP normalized rates to compare performance metrics is described next. In this example, bottleneck metrics as defined by the Top-Down micro-architectural method [1] are compared. The *currency* used to compare the cost is the number of cycles it took to run the program to completion. The *product* being compared are the Top-Down metrics for any given benchmark. The goal is to show that a metric value can differ, or be similar to another, as defined by the Top-Down formulas, while its true cost might be relatively higher or lower, when compared to a baseline run. PPP normalized rates close to 0% imply parity in the performance metrics between the baseline run and the run with that implemented a change. It takes about the same number of CPU_clk cycles for a similar number of pipeline slots to achieve similar Top-Down metric rates. For positive PPP rates, it implies that the CPU_clk cycles are overvalued. The new run requires less CPU_clk cycles to achieve same amount cost when compared to its baseline run. Negative PPP rates imply that the CPU_clk cycles for the run are undervalued. It requires more CPU_clk cycles to complete the same cost when compared to its baseline run. For instance, the cost, in terms of Top-Down metric rates, can be the number of stall cycles it took to store data in memory, or the number of uops that were retired.

An example of how the PPP normalized rates were used to compare bottleneck drift, when comparing different versions of the GCC compiler suite [17], is described next. In the study, the products we compared were benchmark metrics, and the currency that we are comparing are the cycles that it took to complete those benchmarks. PPP theory made it possible to compare different compiler generated Top-Down metrics to determine if the metrics were undervalued or overvalued relative to the same metrics generated by the GCC4 compiler. An undervalued metric signifies that a bottleneck had a relatively lesser effect when compared with the baseline GCC4 metric. Overvalued metrics translate into bottlenecks that had a relatively greater effect than the baseline. We computed the PPP exchange rate, Equation 2.11, by dividing the *cpu_clk_unhalted.thread* event value of a given compiler by the same performance event value that was collected when the same benchmark ran with a binary generated with

the GCC4 compiler. We used *cpu_clk_unhalted.thread* for all computed metrics, except for the FrontEnd Bandwidth metric. This metric uses the *cpu_clk_unhalted.thread_any* performance event, which is also used to calculate its corresponding metric PPP exchange rate. Our choice of *cpu_clk_unhalted.thread* and *cpu_clk_unhalted.thread_any* as normalization metrics was because they account for cycles when the thread is not in a halt state [26], as opposed to using execution time, which is a less granular metric.

$$ER = CPU\_clk/CPU\_clk_{gcc4} \tag{2.11}$$

The Metric variable represents all of the Top-Down Method categories described earlier. As Equation 2.12 shows, we divided a Metric by its GCC4 generated counterpart before using the *ER* variable to compute the PPP normalized percentage.

$$PNR = 100 * ((PM/PM_{gcc4}) - ER)/ER \tag{2.12}$$

For a metric to be overvalued, the performance metric rate, $PM/PMgcc4$, needs to be positive and bigger than *ER*. This occurs when the GCC4 baseline metric is smaller than the number of events of the other compiler. Undervalued metrics occur when the metric rate has a smaller magnitude than the *ER*. We have parity when the PPP normalized rate is zero or very close to zero.

Equations 2.13 and 2.14 show an example of the PPP normalization index for the 350.md benchmark for GCC6 using GCC4 as the baseline. The PPP exchange rate is 0.92. The Bad Speculation metric values were 0.0204 for GCC6, and 0.0206 for GCC4. In this, case we saw that there is a 7.64% relative increase of Bad Speculation for GCC6 over GCC4. The metric is overvalued.

$$270956056194992/294654034404202 = 0.92 \tag{2.13}$$

$$((0.0204/0.0206) - 0.92) * 100/0.92 = 7.64\% \tag{2.14}$$

There are other methods that can be used to identify differences in performance metrics when changes are made to the code or the system configuration is changed. We next present two very popular techniques to identify the effects changes made to a program or a system have on performance metrics.

2.0.2 Other Approaches Used to Identify Differences

The Roofline Model

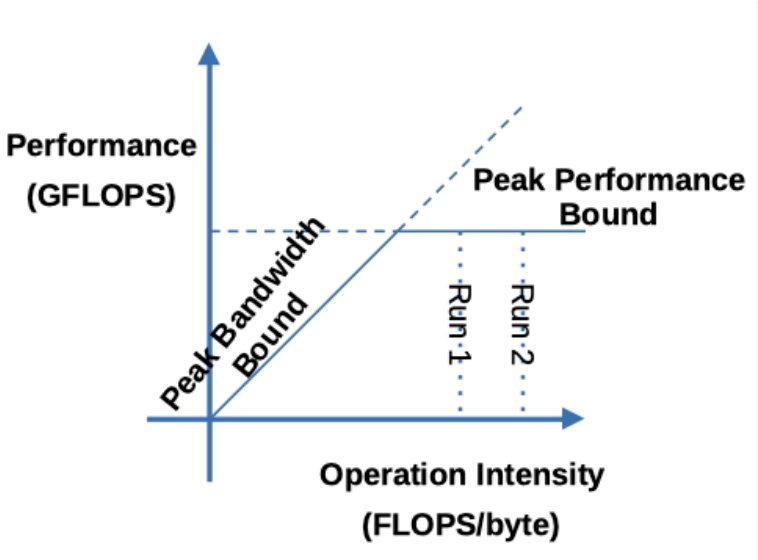


Figure 2.1: Roofline model

The Roofline model [23] is a visual performance tool that can provide information to programmers on how to improve their applications in terms of floating point operations. Figure 2.1 shows an example of the Roofline model. The X axis depicts the operation intensity, the number of operations of Flops per bytes of DRAM traffic. The total byte count is meant to account for the traffic between the different caches and memory instead of the traffic between the processor and the caches. Peak floating-point performance can be obtained from the vendor's specifications or through testing of the system with micro-benchmarks. Peak memory performance results are obtained through the use of the STREAM benchmarks [27].

Peak floating-point performance is plotted using a horizontal line. This will be used as the upper bound since no computational kernel can go higher than the system's hardware limits. Peak memory performance is plotted as a 45 degree angle line. Equation 2.15 shows the formula that is used to compute the maximum floating-point performance bound for the given memory system. Where the two lines intersect, the peak

computational and memory bandwidth performance can be found. Which each successive change to the code, a programmer can compare multiple runs to see the effect each of the changes had in terms of peak memory and computational performance.

$$\begin{aligned} Attainable\ GFlops/sec = Min(&Peak\ Floating\ Point\ Performance,\\ &Peak\ Memory\ Bandwidth \times Operational\ Intensity) \end{aligned} \tag{2.15}$$

The Roofline model has been modified to account for other system properties. Some examples include the following. An extension of the Roofline model that has been used to explain the relationships between time, energy and power costs of a program [28]. To provide more granularity, the Roofline model was modified to account for the different levels of a system's cache hierarchy [29]. The Instruction Roofline for GPUs extended the original Roofline concept to use instructions instead of floating-point operations. This approach made it possible to account for fetch-decode-issue bottlenecks for GPU accelerated applications [30].

Statistical Models

The Roofline Model originally focused on just two performance characteristics, arithmetic intensity and memory bandwidth. Further enhancements allowed for the use of more and varied metrics. There are other approaches that can generalize other characteristics on a system, such as instruction throughput, cache misses, and LLC misses, to find any possible interactions among these characteristics. Statistical models can be used to specify the relationship between one or more independent variables and a dependent variable. A number of techniques are used to systematically select independent variables of significance to a model. Some approaches are more efficient than others.

Experimental design is used to create different combinations of factors –system configuration changes, compiler options, or code changes – with the goal of creating a set of experimental runs that are both balanced, where all factors have the same number of observations, and orthogonal, where effects of each of the factors can be determined independently of other factors [11], [31]. Experimental designs that involve factors that can be assigned two values, high and low, are well understood. Among the many options

Table 2.1: Full Factorial Design for 3 Two Level Parameters

	A	B	C
1	1	-1	-1
2	1	1	1
3	-1	1	1
4	1	1	-1
5	-1	-1	-1
6	1	-1	1
7	-1	-1	1
8	-1	1	-1

available for two-level designs are Full Factorial, Fractional Factorial, Orthogonal Arrays, and Plackett-Burman designs. Researchers select an appropriate design for their needs based on the number of runs needed to complete the design and the ability of the design to differentiate the parameters with the most influence on the response variable from interactions with other factors.

Full Factorial designs can estimate all effects and high order interactions of all of its factors because it searches the complete parameter space by trying out all possible parameter level combinations. Table 2.1 depicts a Full Factorial design for 3 two-level parameters, which requires 8 runs. Full Factorial designs are a resource expensive approach since the number of required runs to implement it grows at an exponential base 2 rate as can be seen in Table 2.2. There are alternative designs that can provide sufficient process optimization information with fewer experimental runs while still being able to estimate the effects individual factors have on the results, and in some cases, interactions among the factors.

The second step of process optimization is the statistical analysis of the collected experimental data obtained from the experimental runs [11]. Generally, linear regression based models are used to describe relationships between response and predictor variables whenever the data to be modeled follows a normal distribution with constant variance. If a linear regression based model is to be applied, analysis of variance, ANOVA, can be used to determine the significance of the different parameters. Plots can be used

Table 2.2: Number of Runs for Full Factorial, Fractional Factorial, and Plackett-Burman Designs

Number of Factors	Full Factorial	Fractional Factorial			Plackett-Burman Design
		Resolution III	Resolution IV	Resolution V	
3	8	4	8	8	4
5	32	8	16	16	
6	64	8	16	32	
7	128	8	16	64	8
8	256	16	16	64	
9	512	16	32	128	
10	1024	16	32	128	
11	2048	16	32	128	12
12	4096	16	32	256	
13	8192	16	32	256	
14	16384	16	32	256	
15	32768	16	32	256	16

to verify if the distribution of the data to be analyzed is normally distributed. Normal distributions have a bell curve shape and quantile-quantile (QQ) plots will depict normality if the plotted data points are close to the straight line. If the points don't fall close enough to the straight line, transformation techniques could be used to try to force the data closer to a normal distribution.

For nonlinear models, Classification And Regression Trees (CART) can be used as an alternative to linear regression [32]. CART is a method by which data is partitioned recursively into a set of rectangles, and in each of the rectangles a model is fitted. The recursive algorithm is as follows:

1. A split of a predictor variable is selected producing two partitions. The partition that minimizes the residual sums of squares (RSS) is chosen, Equation 2.16.

$$RSS(partition) = RSS(part_1) + RSS(part_2) \tag{2.16}$$

2. The previous rule is applied recursively to the child partitions.

3. The splitting stops once there is no further gain or a pre-stopping condition is

met.

The regression tree technique can be applied to categorical and continuous predictor variables without the need to encode categorical values. In linear regression, variable encoding is a requirement because numeric values are used to assign slopes to each of the predictors. Additionally, linear regression requires an experimenter to specify a model and then manually add or subtract parameters or a combination of parameters to improve the validity of the model. At each step, the validity of the model needs to be checked with residual plots. These are plots where the observed error is shown to be consistent with stochastic error if the model is valid. If it is not valid, a different model needs to be tested. Regression trees do not require the specification of a model.

Regression trees are easy to interpret, there are no plots to check the validity of the results. At each partition, the mean value for the response variable for that partition is shown as well as the number of elements that make up that partition, N, and the percentage of the total number elements that N represents. If the node is not a leaf node, then the variable and the value at which the partition was made is shown. Higher values always go to the right. In this document the right most leaf node will represent the partition with the highest GFLOPs mean value.

2.0.3 Conclusion

The Roofline model made it possible to compare the performance of an application in terms of a system's theoretical peak arithmetic intensity and memory bandwidth bounds. The initial Roofline model lacked granularity of the different levels that make up the cache hierarchy when peak memory bandwidth was computed. Subsequent models were proposed that took into account more complex features and properties of a system such as the memory hierarchy, time and instructions. The Roofline model's simplicity lies on the fact that it quantifies performance in terms of two theoretical bounds, and subsequent extended models to include more reference metrics. The main issue is that restricting the use of the number of metrics to use has its limitation. There are many more performance metrics that can give different perspective on how efficiently computational resources are being used by an application. A more complete picture on how resources are being used can make it possible to identify sources of inefficiency that

if addressed could potentially reap performance gains.

Developing statistical models can be time consuming. Multiple runs must be made to generate enough data. Care must be taken to account for orthogonality, and that successive experiments are executed under similar conditions. The resulting model will be restricted by the number of independent variables that were used in the experiments. There is always a chance that an omission of a variable might have skewed the resulting model. No model is perfect, but an understanding of the system will make it possible to make informed decisions when selecting the appropriate variables.

PPP normalized metrics make it possible to compare the relative difference between runs of any performance metric ratio. It is easy to compute, a baseline value is selected, and then it is compared to another value using a common unit as the PPP exchange rate. Any performance metric can be compared using this approach. In the following chapters, will show how different performance metrics were compared using the PPP technique. We were able to identify patterns that would have gone otherwise unnoticed, if regular non-normalized rates were used when comparing results.

Chapter 3

Improving Performance Through the Use of Design of Experiments Techniques

Performance optimization of the HPL benchmark requires finding an optimal combination of settings. This requires multiple runs, each with different settings to try to identify an optimal pattern. Current practices are not as efficient or appropriate for parameter space searches that include qualitative parameters of more than two possible settings. Standard high and low experimental designs might not be appropriate for this case. In this chapter, we showed how we can use Orthogonal Arrays to systematically search the parameter space made up of a mixture of quantitative and qualitative variables in a efficient manner. This approach was presented in [15].

3.1 Introduction

One form of heterogeneous computing is the collaborative computing between GPUs and CPUs, which has the benefit of using all available computational resources to provide better resource utilization and overall performance [33]. The challenge is to find an optimal division of labor between the two very distinct architectures. Each device has its own strenghts and weaknesses, making it difficult to identify overall settings that

will make it possible to operate them both at peak performance.

To this end, we propose the use of asymmetric orthogonal arrays, OA, and regression trees to setup experimental runs that can identify parameters and the levels at which they have an effect in the performance of the application, in this case HPL. Design of experiments approaches have been previously proposed. Fractional factorials were used to analyze the effects of the different architectural features of the ARMv8 architecture on HPC applications [34]. Regular orthogonal arrays were used to identify compiler options that had the greatest effect on an application [35]. Plackett and Burman designs, a form of orthogonal arrays, were used to identify key parameters and the effects these had on the performance of processor enhancements [36].

We use the HPL benchmark [14], a well known and studied benchmark. It is used to rank HPC clusters based on the number of FLOPS [37]. It's been used to exercise CPUs and validate power measurements [38], as a reference in a power consumption analysis of supercomputer workloads [39], and as a validation test for an in-memory checkpointing method [40]. It will be beneficial to the community to have a process to optimize HPL for the different setups for which it is used. For the purposes of our study, there are other advantages to using HPL. HPL has a large number of parameters of different types to tune, making it a challenging experiment to define and interpret. Additionally the HPL version we studied can be used in GPU and CPU collaboration mode, something we are keen to explore. This paper makes the following contributions:

1. We demonstrate the use of an asymmetric statistical design to explore the parameter space where other methods might not suffice or required significantly more experiments. Regular designs allow for the use of multiple parameters with the same number of levels. Some designs can be modified to take a mixture of levels, say 2, 3 and 4, but parameters with more levels are difficult to accomodate in many statistical designs.

2. We show that previously reported HPL parameter combination levels don't provide peak performance in a setup with more computational resources, be it in the form of CPU cores, amount of total memory, number of GPUs, etc. Parameter settings are not universal, resulting in the need to search out new parameter combinations and levels. New technological advances bring new features and better performance.

Such new capabilities make it necessary to test previous assumptions to verify their applicability to the new technologies.

3. We show that regression trees can be used to analyze and select optimal settings for a nonlinear distribution. After each iteration, a regression tree will be used to select the levels at which factors have the most significant effect on the response variable. It results in a fast, efficient and simple way to analyze and select parameters and their optimal values.

In the following sections, we provide background information on the HPL benchmark, the optimization process, design of experiments, and subsequent statistical analysis. First, we discuss the experiment and the results of each set of experimental runs, then we discuss our results and then compare them with previously published results and optimization techniques. Finally, we conclude with some remarks regarding the effectiveness of our approach.

3.2 Background

3.2.1 HPL

[14] is benchmark that solves a dense linear system NxN of linear equations $Ax = b$. HPL performs an iterative LU decomposition with partial pivoting. The matrix is logically partitioned into $NBxNB$ blocks that are distributed onto a PxQ grid of processors. HPL results are used to rank HPC systems in the TOP500 list [37] because it gives an approximation of a system's peak performance. [41] ported HPL to a heterogeneous GPU and CPU cluster, where it was shown that a combined GPU and CPU collaborative computing approach outperformed a GPU or CPU only HPL configuration. GPUs are high performance accelerators for parallel applications, and are made up of hundreds of processing units that support single and double-precision arithmetic. We use this NVIDIA provided HPL version to test our work partition optimization process. The HPL parameters to be tuned and its descriptions are shown in Table 3.1.

Table 3.1: Parameters to be optimized

Variable Name	Variable Category	Component	Description
DGEMM Split	Continuous	HPL	Division of work between GPUs and CPUs
Equilibration	Nominal	HPL	Enables/Disables Equilibration phase
NBMINs	Ordinal	HPL	Recursion stopping criteria
BCASTs	Nominal	HPL	Broadcast algorithms for panels whose factorisation has been computed
DEPTHs	Nominal	HPL	The lookahead depth for trailing sub-matrices
Swap	Nominal	HPL	Specification of swapping algorithm
PFACTs	Nominal	HPL	Panel factorisation
NDIVs	Ordinal	HPL	Subdivision of the panel factorisation
RFACTs	Nominal	HPL	Recursive factorisation
Swapping Threshold	Ordinal	HPL	Number of columns for which the binary exchange swapping algorithm will be applied
NBs	Ordinal	HPL	Number of blocks
BIND	Nominal	OpenMPI	Affinity of the MPI process and its threads
OpenIB Eager Limit	Continuous	OpenMPI	Sets the maximum size of the first fragment
GPU 1 Frequency	Ordinal	GPU	GPU clock frequency (MHz)
GPU 2 Frequency	Ordinal	GPU	GPU clock frequency (MHz)

3.2.2 Process Optimization

Process optimization is an iterative sequence of steps in which an optimal set of factors and combinations are found to have an improved response on a process being analyzed [42]. The first steps involve the design of experiment based on well known statistical techniques that will provide information on the effect certain combinations of factors at different levels have on the process being optimized. Statistical analysis will then be used to quantify the effect each factor has on the process and based on that information,

the experimenter can select the most important factors and their levels or values for the next iteration. The idea is to reduce the number of factors and update the levels at which the factors are being used at each iteration in order to obtain an improved result with each iteration. The process is concluded once the researcher has reached the desired performance gains, there is no further time or resources to allow for further iterations, or there is little performance being gained in preceding iterations.

3.2.3 Design of Experiments

Design of experiments are statistical techniques by which multiple factors are varied at different levels or values in order to gain information on the effect such changes have in a response variable [11],[31]. Design of experiment techniques can be useful in identifying most important factors, interactions between factors, and in the minimization of experimental error or noise. Multiple variables are changed simultaneously and not on an individual basis. One factor at a time experimental approaches fail to identify interactions between the factors. An interaction between factors exists when the presence of one factor has an effect, positive or negative, on another factor. This is important in our experimentation because factors such as GPU clock frequency and percentage of work partition between GPUs and CPUs can have an effect on each other. Interactions will be clearly seen in the following sections.

Experimental designs that involve factors that can be assigned two values are well understood. Factors are assigned high and low values. The design will then create different combinations with the goal of creating a design that is both balanced, where all factors have the same number of observations, and orthogonal, when effects of each of the factors can be determined independently of other factors. Among the many options available for two level designs are full factorial, fractional factorial, orthogonal arrays, and Plackett-Burman designs. Researchers select an appropriate design for the needs based on the number of runs needed to complete the design and the ability of the design to differentiate the parameters with the most influence on the response variable from interactions with other factors.

The challenge arises when the input variables have more than two possible values as it is the case with categorical variables. Regular two level factorial designs will not be suitable for such variables. There are Plackett-Burman and fractional factorial designs

that can be used for a mixture of three and two level factors, but higher factor levels experiments are more difficult to design. Another option is the use of orthogonal arrays. An orthogonal array is a N x M matrix, where $M = M_i + ... + M_z$, representing the predictor variables of the design. Each column M_i can have two or more levels. Any subset of two columns will contain all possible combinations of all existing levels. The number of runs, N, will be divisible by the product of each possible pair of factor levels. Orthogonal arrays provide information of the main effects and some interactions, which is sufficient for our process optimization needs. Asymmetric, or mixed level, OA designs are designs that have a mixture of parameter levels as opposed to regular designs that just have parameters with a similar number of levels, usually two. Asymmetric OA designs were used in for the experimental design presented in this paper.

Table 3.2: Orthogonal Array L12.2.4.3.1

A	B	C	D	E
1	1	1	1	1
2	1	1	1	2
2	1	1	2	3
1	2	1	2	1
1	1	2	2	2
1	1	2	2	3
2	1	2	1	1
1	2	1	1	3
2	2	2	2	1
2	2	2	1	3
1	2	2	1	2
2	2	1	2	2

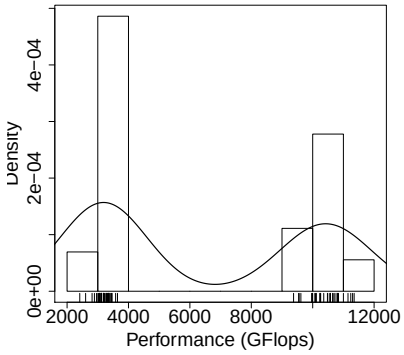
Tables with different OA options are available in [43]. OA are selected based on the number of parameters and the levels of each parameter. There might be different options with varying number of parameters and required number of rows. For instance, for an experiment design using a parameter with three levels and two parameters with two levels each, arrays L12.2.4.3.1, L24.2.16.3.1 and L24.2.13.3.1.4.1 are suitable. Table 3.2 shows array L12.2.4.3.1, which consists of an experimental setup of twelve runs that can be used for experiments with up to four two-level parameters and one three-level parameter. Since the experiment requires only two out of the four two-level parameters,

a matrix consisting of any two two-level and the three-level parameter can be used for the experiment. The strength or accuracy of the experiment is not lost if fewer variables are used. For the aforementioned experiment, columns A, B and E could be used, as well as B, C and E. The efficiency of OA will be clearly shown in the following section, when a design of 72 rows will be used by the first two experimental runs to analyze a one twelve-level, one six-level, seven three-level, and six two-level parameters. The last two experimental runs will require a design with 36 rows to analyze a six-level, six three-level and six two-level parameters. Both OA designs require much less effort than a complete experimental run that will cover all possible combinations.

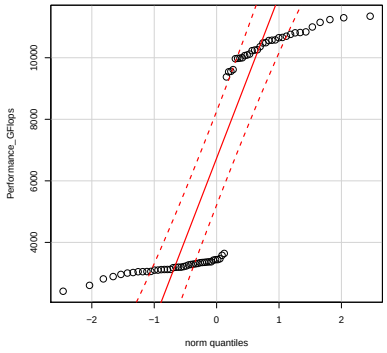
3.2.4 Statistical Analysis

The second step of process optimization is the analysis of collected data [11]. Generally, linear models are used to describe relationships between response and predictor variables whenever the data to be modeled follows a normal distribution with constant variance. Plots can be used to ascertain the distribution of the data to be analyzed. If the data does not satisfy the linear model requirements, other statistical techniques will be required to analyse the data. Figure 3.1 shows the data distribution with histograms, the probability density function as computed by the LOWESS nonparametric regression method, and a QQ plot, which shows normality when the data points are close to the line. Figure 3.1(a) shows that the data distribution is bimodal. Figure 3.1(b) shows that there are two near consistently straight lines, one for each of the two peaks, modes, in Figure 3.1(a). Figure 3.1(c) represents the data distribution of the second run and it continues to show a bimodal distribution that is less pronounced. Figure 3.1(d) shows that a portion of the data distribution falls away from the line. The more pronounced bell shape curve data points are close to the QQ plot line because these points exhibit linearity properties around their local maxima. Figures 3.1(e) and 3.1(f) show that the third run's data distribution is closer to a normal distribution. Transformation techniques could be used analyze the third run but they would not be suitable for the first and second runs because of the noticeable bimodal distribution. Other statistical tools will be needed for at least the first two runs.

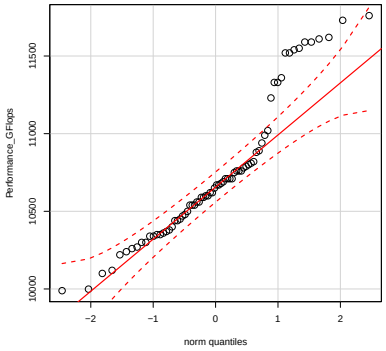
Classification and regression trees (CART) can be used as an alternative to linear models for data modeling. It is a method by which data is partitioned recursively into a



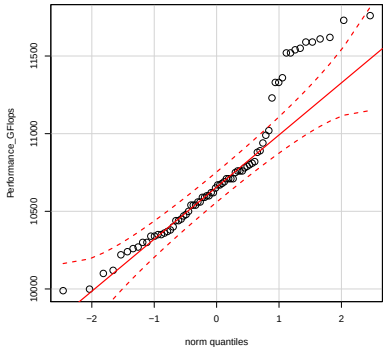
(a) First Run



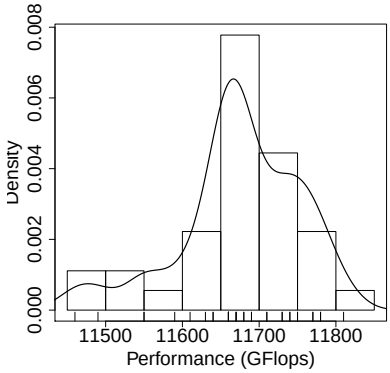
(b) First Run



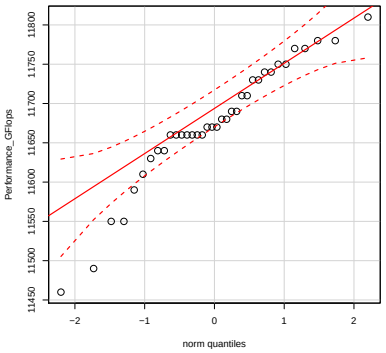
(c) Second Run



(d) Second Run



(e) Third Run



(f) Third Run

Figure 3.1: Probablity density function/histogram and QQ plots for all runs.

set of rectangles, and in each of the rectangles a model is fitted. The recursive algorithm is as follows:

1. A split of a predictor variable is selected producing two partitions. The partition that minimizes the residual sums of squares (RSS) is chosen, where RSS(partition) $= RSS(part_1) + RSS(part_2)$.

2. The previous rule is applied recursively to the child partitions.

3. The splitting stops once there is no further gain or a pre stopping condition is met. [32] provides an in depth description of CART algorithms.

Because partitions take place within partitions and not across partitions, the resulting partitions can be represented as trees. This technique can be applied to categorical and continuous predictor variables. In this paper, regression trees are used to provide information on the optimal parameters that need to be explored.

Regression trees are easy to interpret. At each partition, the mean value for the response variable for that partion is shown as well as the number of elements that make up that partition, N, and the percentage of the total number elements that N represents. If the node is not a leaf node, then the variable and the value at which the partition was made is shown. Higher values go to the right. The right most leaf node will represent the partition with the highest GFLOPS mean value.

3.3 Experimental Setup

The HPL benchmark was run across four nodes of large cluster, each node having 128 GB of RAM and 2 NVIDIA Tesla K40m GPUs. Each K40m GPU has 11 GB of RAM and 2880 CUDA cores. Each node has 24 cores from two Haswell Xeon 2680 V3 CPUs and are connected over a Mellanox Infiniband network. OpenMPI 1.8.4, GCC 4.9.2 and CUDA 7.5 were used in the experiments. The nodes were running Linux CentOS 6.8 with the performance frequency governor enabled.

Table 3.1 shows the parameters to be used in the optimization process. The following variables were hardcoded and their values were set as recommended by [14]. PMAP was set to 0, row column major mapping. L1 was set to 1, for transposed form. U was set

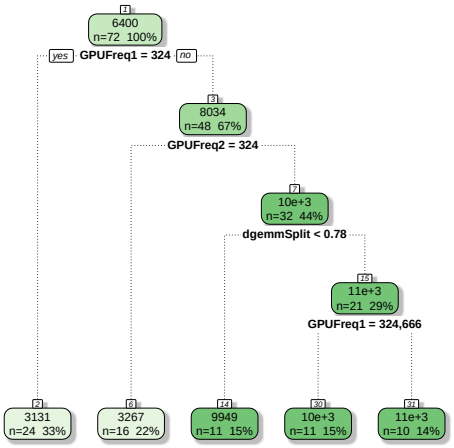
to 0 for non-transposed form. All 12 cores of each socket were assigned to each GPU. N, the problem size, was set to 228608 for all experiments. This size was used given the amount of available memory on each node and the amount of time we had available to run each set of experiments. P and Q, are the values of a PxQ grid of multiple NBxNB blocks. P was set to 4 and Q was set 2 for all experiments because for some values of NBs, HPL failed in its residual check.

3.3.1 First Run

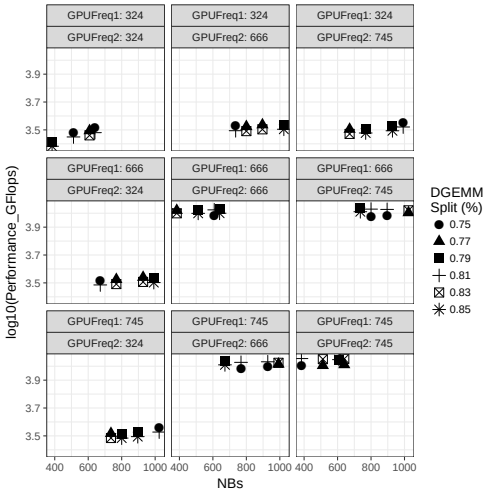
Table 3.3: First Run Values

Variable Name	Values
DGEMM Split	0.75,0.77,0.79,0.81,0.83,0.85
Equilibration	0,1
NBMINs	4,8
BCASTs	4,5
DEPTHs	0,1
Swap	1,2
PFACTs	0,1,2
NDIVs	2,3,4
RFACTs	0,1,2
Swapping Threshold	128,192,384
NBs	384,512,608,640,672,736, 768,800,896,928,992,1024
BIND	core,socket
OpenIB Eager Limit	128000,195000,262000
GPU 1 Frequency	324,666,745
GPU 2 Frequency	324,666,745

The L72.2.27.3.11.6.1.12.1 OA design was used for the first experiment as shown in Table 3.3. Twelve NB values were used along with six DGEMM split values. There were seven additional three-level and six two-level parameters. The first two splits occurred at GPU frequencies of 324. This is to be expected. The K40 GPUs have three base level frequencies and two memory frequency settings. At GPU core frequency 324 MHz, the memory frequency is set to 324 MHz. For GPU core frequencies 666 and 745 MHz, the memory frequency is set to 3004 MHz. So higher clock frequencies will not only have the advantage of faster GPU cores but it will also have access to faster memory. Another



(a) Regression Tree



(b) NBs vs Performance

Figure 3.2: First run plots.

significant parameter is DGEMM split. DGEMM split values range from 0 to 1. At level 1, all work is assigned to the GPUs, while at level 0, all work will be done by the CPUs. In the first run results, values greater than 0.78 reported better performance as shown in Figure 3.3(a) and 3.3(b). The right most leaf reports an average performance value of 10909 GFLOPS for GPU settings of 745 MHz for both GPUs. The peak performance run came in at 11350 GFLOPS with a DGEMM split value of 0.81, NB value of 384

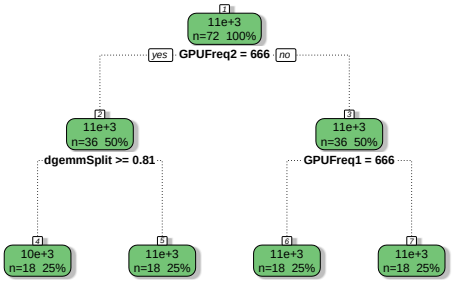
and GPU frequencies set to 745 MHz.

3.3.2 Second Run

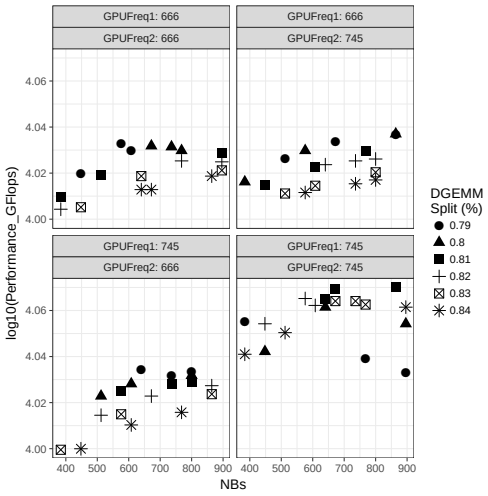
Table 3.4: Second Run Values

Variable Name	Values
DGEMM Split	0.79,0.8,0.81,0.82,0.83,0.84
Equilibration	0,1
NBMINs	4,8
BCASTs	4,5
DEPTHs	0,1
Swap	1,2
PFACTs	0,1,2
NDIVs	2,3,4
RFACTs	0,1,2
Swapping Threshold	128,192,384
NBs	384,448,512,576,608,640, 672,736,768,800,864,896
BIND	core,socket
OpenIB Eager Limit	128000,195000,262000
GPU 1 Frequency	666,745
GPU 2 Frequency	666,745

The L72.2.27.3.11.6.1.12.1 OA design was again used for the second set of experiments. The DGEMM split range was changed to the range 0.79 to 0.84 with 0.01 intervals as indicated Table 3.4. NB values greater than 896 were dropped, and values 864, 576 and 448 were added since the plot for GPU frequencies set to 745 MHz shows that the larger NB values were not optimal. In total, there were six two-level, seven three-level, one six-level and one twelve-level parameters in this experiment. The second run's regression tree, Figure 3.3, shows as expected, that GPU frequencies of 745 MHz outperforms lower frequencies. The NBs vs Performance plot shows that 0.81 DGEMM split has a better performance than other values. Peak performance came in at 1176 GFLOPS with a DGEMM value of 0.81, NB value of 846 and GPU frequencies of 745 MHz.



(a) Regression Tree



(b) NBs vs Performance

Figure 3.3: Second run plots.

3.3.3 Third Run

The L36.2.10.3.8.6.1 OA design was used for the third experimental run. In this set of runs there were six two-level, six three-level and one six-level parameters used. DGEMM split values of 0.81, 0.815 and 0.82 were used based on the data collected from the previous run. Only six NB values were used: 640, 672, 736, 768, 800 and 864. Both GPU frequencies were set to 745 MHz. The NBs vs Performance plot in Figure 3.4(b)

Table 3.5: Third Run Values

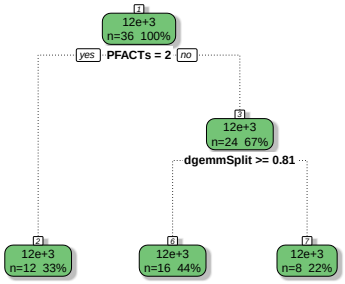
Variable Name	Values
DGEMM Split	0.81,0.815,0.82
Equilibration	0,1
NBMINs	4,8
BCASTs	4,5
DEPTHs	0,1
Swap	1,2
PFACTs	0,1,2
NDIVs	2,3,4
RFACTs	0,1,2
Swapping Threshold	128,192,384
NBs	640,672,736,768,800,864
BIND	core,socket
OpenIB Eager Limit	128000,195000,262000
GPU 1 Frequency	745
GPU 2 Frequency	745

shows that 0.81 is the best performing DGEMM split value when used with NB set to 864. 0.81 also outperforms the other two DGEMM split settings for NB values 736 and 672. The regression tree plot in Figure 3.4(a) shows that PFACTs values have a bigger effect on performance than the DGEMM split variable. Peak performance came in 11810 GFLOPS with DGEMM split value of 0.81, PFACT set to 1 and NB value set to 864.

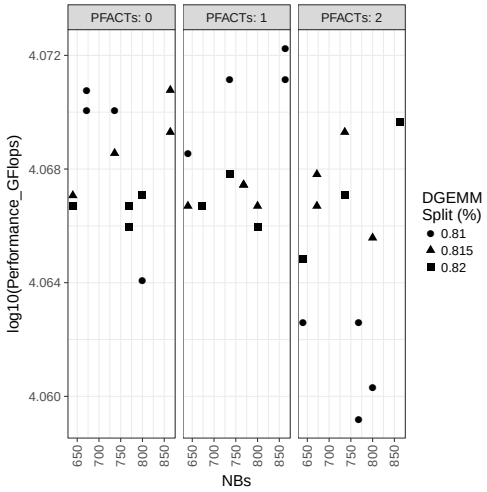
3.4 Results and Discussion

Regression trees in Figures 3.2(a) and 3.3(a) showed that optimal performance was obtained by setting the frequency of both GPUs to 745 MHz. Regression tree in Figure 3.4(a) showed that for PFACTs values less than 2 and DGEMM split values of less than 0.8125, HPL has its peak performance. Figure 3.4(a) shows a DGEMM split value 0.81 instead of the calculated value of 0.8125 due to a lack of space. Figures 3.3(b) and 3.4(b) showed that DGEMM split value of 0.81 outperformed all other values when NB was set to 864.

In order to verify the results of this OA experimental design, an additional set of



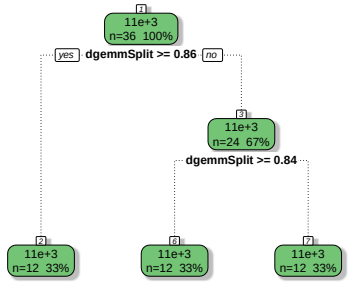
(a) Regression Tree



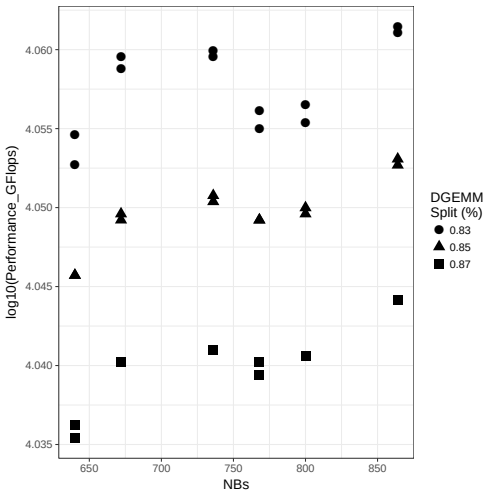
(b) NBs vs Performance

Figure 3.4: Third Run Plots. GPU Frequencies Set to 745 MHz.

runs were made to show the validity of the obtained DGEMM split optimal values. The fourth run parameters and their levels are shown in Table 3.6. This test run is a set of runs that was used to test whether or not DGEMM split values greater than 0.81 performed better. The L36.2.10.3.8.6.1 OA design for DGEMM split values of 0.83,0.85 and 0.87 with NB values of 640, 672, 736, 768, 800, and 864. Figure 3.5 shows the results of this run. As it can be seen in Figure 3.5(b), lower DGEMM split values



(a) Regression Tree



(b) NBs vs Performance

Figure 3.5: Fourth run. GPU frequencies set to 745 MHz.

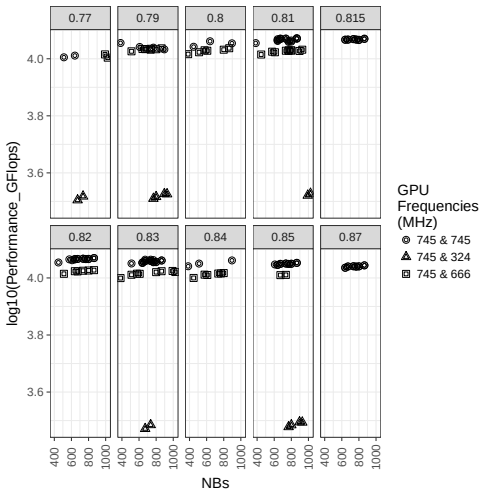
perform better. The regression tree in Figure 3.5(a) confirms that DGEMM split values have the most significant effect out of all the parameters. This confirms the conclusion that there is no performance gain when the DGEMM split value is increased past 0.81 for the given set NB values.

Figure 3.6 shows the combined performance results of all experiments: the original three experiments and the additional experiment. GPU frequencies were combined into

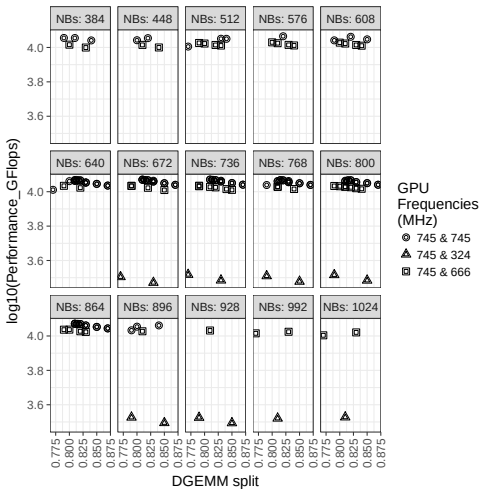
Table 3.6: Fourth Run Values

Variable Name	Values
DGEMM Split	0.83,0.85,0.87
Equilibration	0,1
NBMINs	4,8
BCASTs	4,5
DEPTHs	0,1
Swap	1,2
PFACTs	0,1,2
NDIVs	2,3,4
RFACTs	0,1,2
Swapping Threshold	128,192,384
NBs	896,640,672,736,768,800,864
BIND	core,socket
OpenIB Eager Limit	128000,195000,262000
GPU 1 Frequency	745
GPU 2 Frequency	745

a single categorical entry. For instance, entries marked by frequencies 745 & 324 represent GPU frequency settings GPUFreq1 = 745 and GPUFreq2 = 324 and GPUFreq1 = 324 and GPUFreq2 = 745. The plots show that the performance increases as DGEMM split values are increased starting at 0.77. DGEMM split values 0.81, 0.815 and 0.82 are very similar with 0.81 being the best performing value. Overall performance starts to decrease as the DGEMM split value is increased beyond 0.82 for all NBs. We also show that NB value of 864 is an optimal solution. Figure 3.6(a) shows that there is a gradual performance increase starting from NB 384, it peaks at 864 and then it continues a gradual decrease in performance. Figure 3.6 confirms that the approach proposed in this paper produces an optimal performance solution to the HPL parameter selection problem. [44] recommended an NB setting of 896 and a DGEMM split of 0.95. Our largest tested DGEMM value for that given NB, 896, was 0.87. Its performance came in at 1.107e+04, which is 6.7% lower than peak performance achieved by our method.



(a) Performance vs NBs



(b) Performance vs DGEEM Split

Figure 3.6: DGEMM split, GPU Frequencies and NBs comparisons across all runs.

3.5 Related HPL Works

HPL is a well studied benchmark and there are many published works with different parameter settings, but many of them do not include an HPL version of the algorithm where the GPU works in collaboration with the CPU and hence its proposed parameter settings are not applicable. This is due to the fact that the those setups do not need to use the DGEMM split variable to divide the work between the GPUs and CPUs.

Also, the CPU only HPL setups will require a NB value that is much lower than the one used by GPU only setup to obtain optimal performance as [45] points out. An optimal collaborative HPL setup will require a NB value that is optimal for both kinds of architecture and a division of work that is optimal for the number of CPU cores and GPUs.

[46] provides a guide to run HPL in collaborative and CPU only modes. The setup used an NB value of 512 with DGEMM split values of 0.836 for a two Tesla S2050 with a Intel Hex Core X5670 dual core processor. Another run used DGEMM split of 0.66 and a NB value of 1152. For CPU only runs, an NB value of 224 was used. As our experiments showed, these settings are not condusive to peak performance for our hardware setup.

In similar study to the one presented in this paper, Konstantinos [44], describes the process used to optimize HPL in a four node setup. Each node had two K40 NVIDIA GPUs, 64GB of memory, two Intel Xeon E5-2680V2 CPUs, each with 10 cores each connected over a Mellanox Infiniband network. The author reports that NB set to 896 and a DGEMM split of 0.95 worked best with GPU clock frequencies set to 745 MHz. Figures 3.5 and 3.4(b) show that the combination of DGEMM split set to 0.81, PFACT set to 1, GPU frequencies set to 745 MHz and NB set to 864 provide the best result. A 5% CPU workload is too small for a system that has twice as much memory per node and two more cores per socket than the system used by Konstantinos. The reported peak performance was 10320 GFLOPS for a matrix of size 161280. Our system peaked at 11810 GFLOPS for a matrix of size 228608 after three experimental runs.

Konstantinos uses a simple, one variable at a time, process to optimize the parameter settings. To find an optimal NB value, the author iterated through multiple NB values for a given matrix size N. For the first set of runs, the following values were used 256, 512, 768, 1024, 1280, and 1536 with a matrix of size 153600. In a second set of runs, NB values 840, 896 and 960 were used with a matrix of size 161280. NB value of 896 gave the best performance result out of the different setting combinations. To find the optimal DGEMM split value, the following were tested, 0.95, 0.96, 0.97, 0.98, 0.99, 0.995 and 1. For each DGEMM split value, two different CPU frequencies were tested, 2 GHz and 2.8 GHz. 0.95 gave the best results out of the tested DGEMM split values. There was another set of runs that was used to find an optimal matrix size. In this case,

the following matrix sizes were tested 157696, 159488, 160384, 161280, 162176, 162206, 164864 and 168960. There is an issue with this approach:

The simple one variable at a time search in Konstantinos' study is limited. For instance, when searching for an optimal matrix size, N, the only variable that changed in value was N. No DGEMM split or NBs were changed during this experiment and as a result no information on the effect and interaction of these variables with the size of the matrix could be obtained. As we reported in the previous sections, parameter combinations have a significant influence in the results and they should be included in any parameter search. By not varying the parameter setting in each of the runs, it is difficult to glean information regarding parameter interactions. Parameters interact, and some interactions can have more influence in the performance results than others.

In [47], a fast correlation-based filter solution, a machine learning approach was used to select the most significant HPL parameters in two CPU based HPC clusters. The authors identified parameters NB, Q and N as significant in one cluster, and Q, NB and N in the other cluster. While machine learning techniques are widely used for data analysis, it is not clear if the data being analyzed was generated in a way that reduces the confounding amongst the variables. In the second portion of their paper, the authors vary the parameter levels they identified as most significant, to characterize the power efficiency of their clusters. The issue with their approach is that the experimental runs that were performed were similar to the ones presented in [44]. For their first experiment, the authors held fixed the value of the PxQ grid, while they varied NB and N. In their second set of experiments, the value of NB was held fixed while the PxQ grid was varied. As it was stated earlier, there are shortcomings to the one variable at a time approach, and these shortcomings make it difficult to obtain optimal results.

In [48], data mining techniques are used to identify the most important HPL factors on a CPU based HPC cluster. The authors use regression techniques to obtain general models using one subset to train the data and another to test the data. Parameter selection occurs using feature selection methods: ReliefF, SWMWrapper, M5Wrapper and Correlation-based Feature Selection. All four methods selected N, NB, P and Q as significant. Only three methods selected PMAP as an additional parameter. The authors recognize that if issues arise from the data mining analysis, these issues come from anomalies in the data or the guidelines used as a basis for the analysis were not

suitable for the specific system. These shortcomings can be addressed by generating data suitable for analysis. And this can be done consistently through the use of statistically robust design of experiment techniques.

An alternative nonstatistical approach to NB value selection is proposed by [45]. The authors discuss the NBs issue, the NBs selection dilemma between a GPUs that require a bigger NBs and CPUs that require a lower NB. Their solution to the NB level selection is to use an adaptive NB. For portion of the HPL code that are GPU dominant, a larger NB value is used. For the portions of the code that CPU dominant, a lower NB value will be used. This is an alternative to using hardcoded parameter values for the entirity of the run, but it is not clear from their results if a properly tuned HPL like we proposed in this paper, compares well with a dynamically updated NB HPL. No discussion on the effects of the other HPL variables and their interactions with NB values is provided.

As this paper has shown, an optimization based on a design of process approach can yield an optimal solution because it can systematically search a large parameter space and provide information on parameter interactions and provide information on main effects thanks to the orthogonality of the design. This can minimize the number of experiments required. Other proposed approaches were shown to use analytical techniques that are well established but their data generating approach was not as effective as one based on design of experiment techniques, keeping the system from achieving performance gains.

3.6 Conclusion

Collaborative computing between CPUs and GPUs make it possible to use all available resources. But there is the challenge of trying to optimally divide the amount of work between the CPUs and GPUs, with each having very different architectural characteristics and bottlenecks. This is in addition to the software's own tunable parameters. The search of an optimal parameter combination in a large parameter space can be complicated and long. In this paper we presented a method that can be used to tune an application with different parameters in a collaborative CPU-GPU setting. Performance tuning is not the only possible use of this method, the response variable presented in

this study was performance as measured by FLOPs. A different response variable such as power consumption could be used to find an optimal parameter setting for power efficiency.

We showed that previously published parameter settings were not all suitable since technological advances such as an increase in the number of CPU cores have a direct impact in the amount of work a CPU can do as compared with previous CPU generations. When the number of cores, their frequency, the memory bandwidth and PCI Express bandwidth changes, these parameter settings will no longer be suitable and a new set of experiments will be needed to find the optimal or near optimal settings. Having a process that can be used to systematically search the parameter space will assist researchers in finding an optimal solution for their specific hardware and application setup. As new features and components are added to computing platforms, the number of tunable parameters will increase. With higher complexity, design of experiment techniques will make it possible to systematically search a complex parameter space for optimal parameter combinations, while minimizing the number of required experiments.

Chapter 4

Quantifying Compiler Drift

Programming tools like compilers evolve in order to adapt to new hardware features, to incorporate the latest security fixes, introduce new features and improvements. Previous works have highlighted and quantified performance drift between versions of compilers using different architectures. In this chapter, we show that the standard comparison techniques can be further enhanced through the use of Purchasing Power Parity normalization techniques. This approach highlighted differences and trends that might have gone unnoticed through the use of standard techniques [17].

4.1 Introduction

Parallel programming frameworks, such as OpenMP, have made it possible to use highly complex computational resources efficiently. But compilers and programming frameworks have a wide variety of features and options that can make it challenging to obtain optimal performance without some significant effort. It is essential to have analysis tools and techniques that can provide users with precise and detailed information on the compiler's performance to assess whether or not the generated code uses all of the architectural features of the CPU efficiently, as measured in overall runtime. The user could select different compiler flags, modify the code, or select a different compiler version that might further enhance performance. When designing compilers, developers are interested in in-depth profiling to quantify the impact of new features on performance.

Performance analysis can be a challenging and time-consuming endeavor. Modern

CPUs have complex microarchitectures that improve overall code performance. The use of deep pipelines, buffers and prefetchers makes it possible to hide and mitigate stalls – delays in the processing of an operation. Stalls can occur when an operation has to wait for needed resources to become available before it can be completed. This microarchitectural complexity makes it difficult to analyze bottlenecks – portions of the code where stalls occur, and that should be examined for a possible change to improve performance, once they are properly identified, and their effects quantified. Features that minimize stalls and improve overall performance will tend to make it difficult to pinpoint such bottlenecks by hiding latencies. For example, a CPU will reduce its data retrieval time by guessing which data value an instruction will need next, or by storing frequently used instructions or memory contents in its caches.

To better understand the behavior of the CPU, computer chip manufacturers include performance counter units in their CPUs to track hardware and software events. Performance counters will keep track of the number of times a particular event occurs for a user-defined sampling interval. These events include cache-misses, cycles, page-faults, and branches. Kernel tools like *perf* [49] will interact with the performance counters and report the results. These reports can provide useful information on how the program behaves on the CPU as it was executed, e.g., how a component is being efficiently utilized.

Performance analysis is an iterative process. Users will profile the code to get baseline bottleneck metric values. Based on those values, the user will then make appropriate changes to the code, environment, or compiler flag to reduce the effect of the bottleneck. Then the code will be recompiled, executed, and profiled again. The goal is to understand the effects the change had on performance and how it is reflected in CPU bottlenecks. Further, the goal is to identify and mitigate bottlenecks to improve performance. Bottleneck identification and quantification of their effects are crucial in this optimization process. In this paper, we extend current analysis techniques to compare and quantify how different compilers use CPU features. Our goal is to improve the performance analysis process by making it easier to compare different compilers.

This study shows the following:

- For many programs, newer versions of GNU compilers [50] will not always entail better performance.

- Current analysis techniques show and quantify the difference in reported bottlenecks between programs generated by the same compiler. Our study shows that these bottleneck comparisons, across programs generated by different compilers, can differ greatly, and this variation is not captured using the regular technique.
- The PPP normalization technique can be used to better compare bottlenecks across multiple compiler versions using a specific compiler as a reference. In our study, we use GCC4 compiled programs to compare the magnitude of bottleneck changes across different compiler versions.
- PPP normalized rates make it easy to make bottleneck comparisons show trends and highlight differences.

4.2 Background

4.2.1 The Top-Down Method

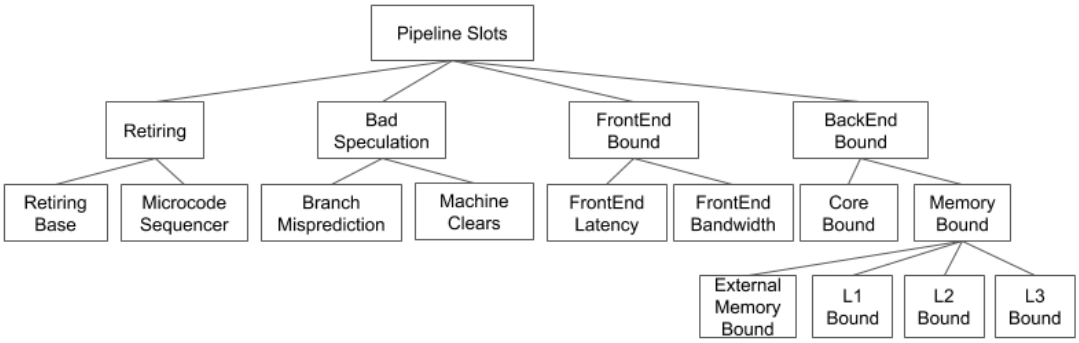


Figure 4.1: Top-Down hierarchy breakdown used in this study. Retiring, Bad Speculation, FrontEnd Bound and BackEnd Bound are the main categories used to classify pipeline slots. Subsequent subcategories give more granular information on specific architectural components.

When optimizing a program, a developer will make changes to the program or compiler options until the desired performance gains are made, or there are no longer resources to continue the process. This technique, called differential analysis [20], can be combined with a systematic approach to bottleneck identification, called the Top-Down

method [1]. The Top-Down method focuses on accounting for the use of pipeline resources, making it possible to highlight micro-architectural components that generate stalls, and narrow down the list of possible components, accounting for the most stalls. This iterative Top-Down method is used by the Intel VTune profiler suite [51] and has been used to analyze different systems [52],[53],[54].

The Top-Down method divides the instruction pipeline into two parts: the Front End and the Back End. The Front End is the portion of the pipeline where instructions are fetched, decoded into *uops* – low-level instructions – and then queued to wait their turn to be executed by the Back End. The Back End will schedule *uops* for execution, for example, integer store/load or floating-point operations. The results will then be committed or retired.

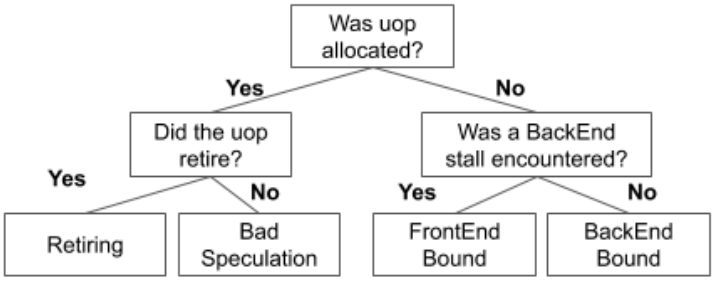


Figure 4.2: Top-Down *uop* classification tree.

The Top-Down method has four categories to track the progress of *uops* as they traverse the pipeline. The categories are FrontEnd Bound, BackEnd Bound, Bad Speculation, and Retiring. Figure 4.1 shows the Top-Down classification hierarchy used in this study. The category assignment occurs as follows – if a *uop* is allocated, then it will be either retired or not retired. If it is retired, it will be assigned to the Retiring classification. If it is not retired, it will be classified as Bad Speculation. Retiring and Bad Speculation account for non-stalled slots. If the *uop* is not allocated, then there are two choices, if there is a Back End stall, then it will be classified as BackEnd Bound. If it is not a Back End stall, then it will be classified as FrontEnd Bound. Figure 4.2 shows the decision tree used in *uop* classifications.

The BackEnd Bound stalls occur when *uops* are not delivered to the execution pipeline due to lack of resources at the Back End. This could be due to a data-cache

miss or a stall due to lack of execution resources. BackEnd Bound stalls are divided into Memory Bound stalls – stalls related to the memory subsystem and Core Bound stalls – all other BackEnd Bound stalls. L1 Bound, L2 Bound, L3 Bound, and External Memory are stalls when accessing data already present in the specific memory subsystem.

The Retiring category focuses on the last stage of the pipeline in which the *uops* are completed and retired. Ideally, this should be the largest category of all Top-Down classifications. Retiring can be further divided into Retiring Base – retired *uops* that do not involve using the microcode sequencer (Microcode Sequencer) when complex operations are broken up into multiple *uop* operations such as sine and cosine. Heavy use of the Microcode Sequencer could imply that some of the extra *uops* could have been avoided, and the use of additional compiler options, such as the flushing to zero of certain subnormal results *-ftz*, could prove beneficial [55].

FrontEnd Bound stalls occur when the Front End portion of the pipeline does not supply enough *uops* to the Back End portion of the pipeline to process when the Back End is ready to receive them. FrontEnd Bound stalls are further divided into two subcategories, FrontEnd Bandwidth – the number of cycles when the number of *uops* is less than the maximum allowable of 4 and FrontEnd Latency – the number of cycles when no *uops* were issued to the available Back End.

The Bad Speculation category involves wasted pipeline resources with operations that are not useful. There might be *uops* that were issued speculatively and will never be retired, thus wasting pipeline slots, or stalls that occur when the pipeline recovers from an earlier misspeculation. The subcategories that make up Bad Speculation are Branch Mispredicts and Machine Clears. The Branch Mispredictions category is used when a bad speculated branch takes place, while *uops* are classified as Machine Clears when the pipeline is flushed to recover from a misspeculation.

Table 4.1 shows the analysis of a matrix multiplication example. The formulas needed to compute the different Top-Down metrics are given in [1], [7]. The first step, *multiply1*, is a simple matrix multiplication that is Memory Bound. The use of loop interchange optimization in the second step, *multiply2*, achieves a significant performance gain by shifting the bottleneck from Memory Bound to Compute Bound. Loop interchange takes advantage of locality through better use of memory access patterns. In the last step, *multiply3* uses vectorization to further improve performance by reducing

Table 4.1: Results of tuning matrix-multiply case [1].

Metric	multiply1	multiply2	multiply3
Speedup	1.0x	11.8x	16.5x
IPC	0.17	1.19	0.80
Frontend Bound	0.00	0.07	0.02
Retiring	0.05	0.41	0.28
Bad Speculation	0.00	0.00	0.00
Backend Bound	0.95	0.52	0.70
Memory Bound	0.84	0.12	0.31
L1 Bound	0.05	0.07	0.03
L2 Bound	0.03	-	0.05
L3 Bound	0.05	-	0.01
Stores Bound	-	-	-
Core Bound	0.15	0.64	0.55
Divider	-	-	-
Ports Utiliz	0.15	0.64	0.55

the use of port utilization [1]. These optimization examples show that different bottlenecks can be identified, and their effects mitigated in order to obtain performance enhancements. The incremental changes between versions of the program generated by the same compiler is manageable. Code changes will potentially affect the specific bottleneck in question and their effects potentially mitigated with each change.

Comparisons between programs generated by different compilers or compiler versions are more difficult. Each compiler could generate different instructions, and the number of cycles required to complete those instructions could vary widely for the same program, resulting in widely different bottlenecks. To make it easier to compare and analyze programs generated by different compilers, we propose the use of the Purchasing Power Parity (PPP) economic theory. PPP will be used to normalize Top-Down metrics with respect to a single compiler, GCC4, to be able to better compare metrics generated by different compilers. We describe this approach in the next subsection.

4.2.2 Purchasing Power Parity

PPP states that *the exchange rate is proportional to the ratio of price levels in two countries.* [22]. Essentially, it allows for the comparison of the same good in different countries to see if a country's currency is overvalued (when the good is more expensive), or undervalued (when the good is cheaper) relative to another currency [56]. The Big Mac Index [25] is the most famous application of the PPP theory, and it compares the value of Big Mac burgers across many countries. Equations 4.1 and 4.2 show an example of how to compute the Big Mac Index. Assume that the cost of a Big Mac in the US is $3.57. The same burger costs 7.50 reales in Brazil. The PPP exchange rate is shown in Equation 4.1, which is 2.10. The currency exchange rate at that time was of 1.58 reales for $1. In Equation 4.2, we compute the Big Mac Index by subtracting 1.58 from the PPP exchange rate and then dividing by 1.58. According to the index, the real is overvalued almost 33% as compared to the US dollar [22].

$$7.50/3.57 = 2.10 \tag{4.1}$$

$$(2.10 - 1.58) * 100/1.58 = 32.91\% \tag{4.2}$$

In our study, the products we compare are benchmarks, and the currency that we are comparing are the cycles that it took to complete those benchmarks. PPP theory allows us to compare different compiler generated Top-Down metrics to determine if the metrics are undervalued or overvalued relative to the same metrics generated by the GCC4 compiler. An undervalued metric signifies that a bottleneck has a relatively lesser effect when compared with the baseline GCC4 metric. Overvalued metrics translate into bottlenecks that have a relatively greater effect than the baseline. We compute the PPP exchange rate, Equation 4.3, by dividing the *cpu_clk_unhalted.thread* event value of a given compiler by the same performance event value that was collected when the same benchmark ran with a binary generated with the GCC4 compiler. We use *cpu_clk_unhalted.thread* for all computed metrics, except for the FrontEnd Bandwidth metric. This metric uses the *cpu_clk_unhalted.thread_any* performance event, which is also used to calculate its corresponding metric PPP exchange rate. Our choice of *cpu_clk_unhalted.thread* and *cpu_clk_unhalted.thread_any* as normalization metrics is because they account for cycles when the thread is not in a halt state [26], as opposed to

using execution time, which is a less granular metric.

$$PPP\_Exchange\_Rate = CPU\_clk/CPU\_clk_{gcc4} \tag{4.3}$$

The Metric variable represents all of the Top-Down Method categories described earlier. As Equation 4.4 shows, we divide a Metric by its GCC4 generated counterpart before using the PPP_Exchange_Rate variable to compute the PPP normalized percentage.

$$PPP = 100 * ((Metric/Metric_{gcc4}) - PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \tag{4.4}$$

For a metric to be overvalued, the metric rate, $Metric/Metric_{gcc4}$, needs to be positive and bigger than $PPP\_Exchange\_Rate$. This occurs when the GCC4 baseline metric is smaller than the number of events of the other compiler. Undervalued metrics occur when the metric rate has a smaller magnitude than the $PPP\_Exchange\_Rate$. We have parity when the PPP normalized rate is zero or very close to zero.

Equations 4.5 and 4.6 show an example of the PPP normalization index for the 350.md benchmark for GCC6 using GCC4 as the baseline. The PPP exchange rate is 0.92. The Bad Speculation metric values are 0.0204 for GCC6, and 0.0206 for GCC4. In this, case we see that there is a 7.64% relative increase of Bad Speculation for GCC6 over GCC4. The metric is overvalued.

$$270956056194992/294654034404202 = 0.92 \tag{4.5}$$

$$((0.0204/0.0206) - 0.92) * 100/0.92 = 7.64\% \tag{4.6}$$

4.3 Experimental setup

In this study, we use the Top-Down method in combination with the SPEC OMP2012 benchmarks [57], [2] to evaluate versions 4.8.5, 6.3.1, and 7.3.1 of the C, C++, and Fortran GCC compiler suite. It is a freely available and widely used suite of compilers that can run in multiple platforms and operating systems. All benchmarks were

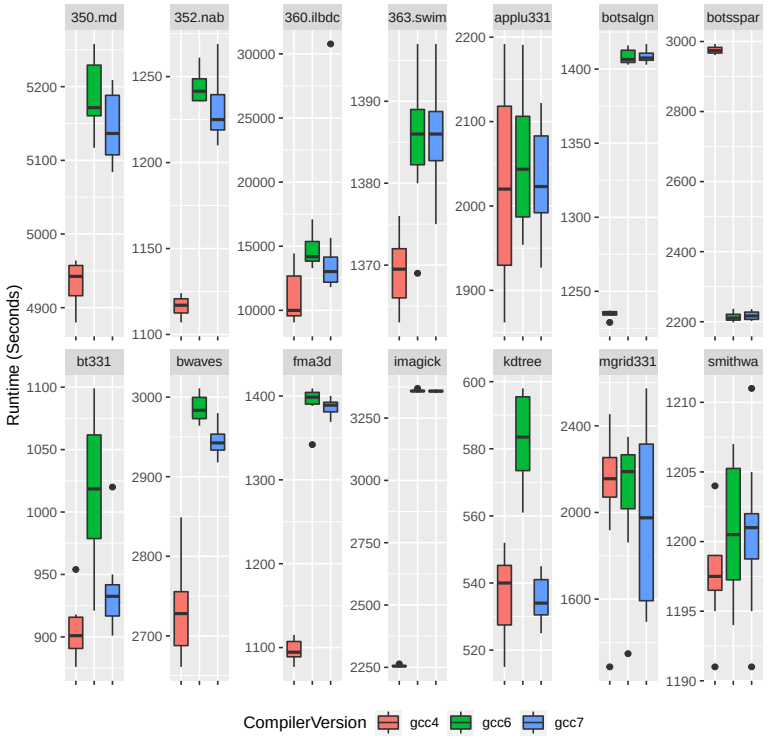


Figure 4.3: Runtimes for SPEC OMP2012 benchmarks using 32 threads with SMT enabled. Lower is better.

compiled using the following flags: *-fopenmp -O3 -march=native*. The only exception was *371.applu331*, which used these flags: *-fopenmp -O2 -march=native*. Additionally, *-ffree-form -fno-range-check* were used for *350.md* and *-std=c99* for *367.imagick*. The SPEC OMP2012 benchmarks include a wide variety of commonly used computational kernels written in Fortran, C, and C++. These kernels make it possible to stress different system components, including CPU, memory, and parallel support libraries, making possible to highlight how each compiler handles bottlenecks. The OMP2012 benchmark suite is widely used to compare HPC systems using OpenMP applications. Benchmarks results that meet the SPEC group reporting guidelines are published for comparison in a public repository where different systems with a variety of designs and compilers can be compared [58].

A two socket Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 8 cores per socket,

2 threads per core was used running the CentOS 7.6.1810 Linux operating system installed. Figure 4.3 was generated using the walltime reported by the SPEC framework and includes the results for the 14 SPEC OMP2012 benchmarks. Figures 4.4, 4.5, and 4.6 were generated using *perf* and include 6 additional programs to the original total of 14. These programs were included because the *367.imagick* benchmark is made up of a process that involves 6 different steps, *convert11, convert2, convert9, vall11, val2,* and *val9. 372.smithwa* has a two step process, *refset1* and *refset2.*

4.4 Results and Analysis

Table 4.2: SPEC OMP2012 Benchmark Description [2]

Benchmark Name	Programming Language	Description
350.md	Fortran	Physics: Molecular Dynamics
351.bwaves	Fortran	Physics: Computational Fluid Dynamics (CFD)
352.nab	C	Molecular Modeling
357.bt331	Fortran	Physics: Computational Fluid Dynamics (CFD)
358.botsalgn	C	Protein Alignment
359.botsspar	C	Sparse LU
360.ilbdc	Fortran	Lattic Boltzmann
362.fma3d	Fortran	Mechanical Response Simulation
363.swim	Fortran	Weather Prediction
367.imagick	C	Image Processing
370.mgrid331	Fortran	Physics: Computational Fluid Dynamics (CFD)
371.applu331	Fortran	Physics: Computational Fluid Dynamics (CFD)
372.smithwa	C	Optimal Pattern Matching
376.kdtree	C++	Sorting and Searching

Table 4.2 gives the description of the 14 benchmarks that make up the SPEC OMP2012 suite. Figure 4.3 shows that for many of the benchmarks, runtimes can vary significantly depending on the choice of compiler. On average, eight of the benchmarks, 350.md, 352.nab, botsalgn, 360.ilbdc, bt331, bwaves, fma3d, and imagick, perform better when using the GCC4 compiler. While benchmark runtimes might differ, their respective Top-Down metrics might be relatively similar for 360.ilbdc, 350.md, and

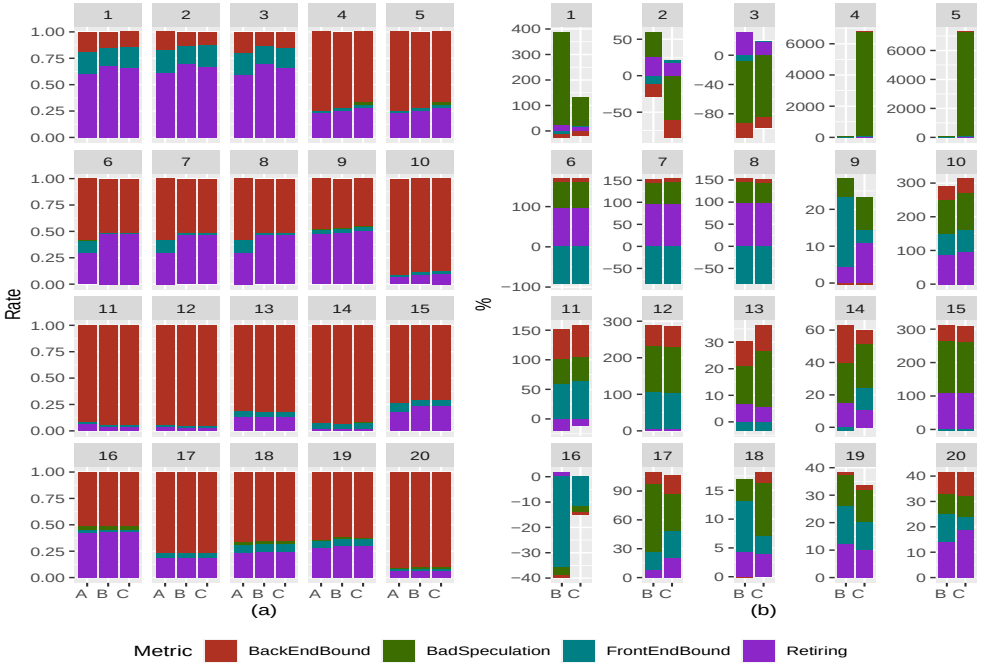


Figure 4.4: Top-Down method categories for SPEC OMP2012 benchmarks: 1. val9, 2. val2, 3. val11, 4. refset2, 5. refset1, 6. convert9, 7. convert2, 8. convert11, 9. 376.kdtree, 10. 371.applu331, 11. 370.mgrid331, 12. 363.swim, 13. 362.fma3d, 14. 360.ilbdc, 15. 359.botsspar, 16. 358.botsalgn, 17. 357.bt331, 18. 352.nab, 19. 351.bwaves, 20. 350.md using compilers: A. GCC4, B. GCC6, C. GCC7. (a) is the regular metric. (b) is the PPP normalized metric.

.

bt331, Figures 4.4(a), 4.5(a) and 4.6(a). Our goal is to show that Top-Down metrics, even when they have similar values, can diverge in magnitude when normalized with a reference compiler. Using Equations 4.3 and 4.4, we computed the PPP normalized metrics using the benchmark results from the three different compilers using the performance metrics results collected via *perf*. The results are shown in Figures 4.4(b), 4.5(b) and 4.6(b). These plots can give us better information of how stalls, BackEnd and FrontEnd Bound categories, and non-stalls, Retiring and Bad Speculation categories, are overvalued, undervalued or have bottleneck parity for each compiler. In the following subsections, we show how these results allow us to extract important insights about the relative magnitude difference of bottlenecks between compilers.

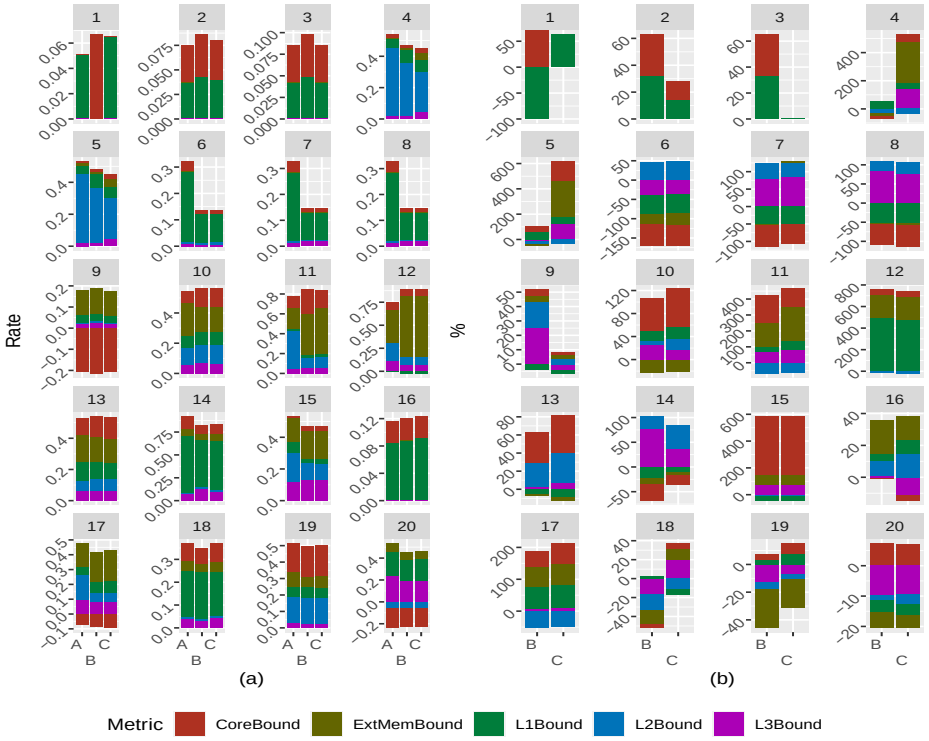


Figure 4.5: BackEnd Bound subcategories, Core Bound and Memory Bound (ExtMemBound, L1Bound, L2Bound and L3Bound) for SPEC OMP2012 benchmarks: 1. val9, 2. val2, 3. val11, 4. refset2, 5. refset1, 6. convert9, 7. convert2, 8. convert11, 9. 376.kdtree, 10. 371.applu331, 11. 370.mgrid331, 12. 363.swim, 13. 362.fma3d, 14. 360.ilbdc, 15. 359.botsspar, 16. 358.botsalgn, 17. 357.bt331, 18. 352.nab, 19. 351.bwaves, 20. 350.md using compilers: A. GCC4, B. GCC6, C. GCC7. (a) is the regular metric. (b) is the PPP normalized metric.

.

- **Overvalued bottlenecks:** The refset1 and refset2 programs are an example of how PPP normalized values can give us a better picture of the magnitude of bottleneck differences when different versions of compilers are used. Refset1 has a Bad Speculation rate of 0.000474 for GCC4, 0.000624 for GCC6 and 0.0307 for GCC7. If we take the ratio of GCC7 and GCC4, we get 64.77 and the ratio of GCC6 and GCC4 is 1.316. To compute the PPP normalized rate, we need to use the PPP exchange rates of 0.994 for GCC6 and 0.884 for GCC7, which yield a PPP normalized rate of 32.5% for GCC6 and 7222% for GCC7. The normalization

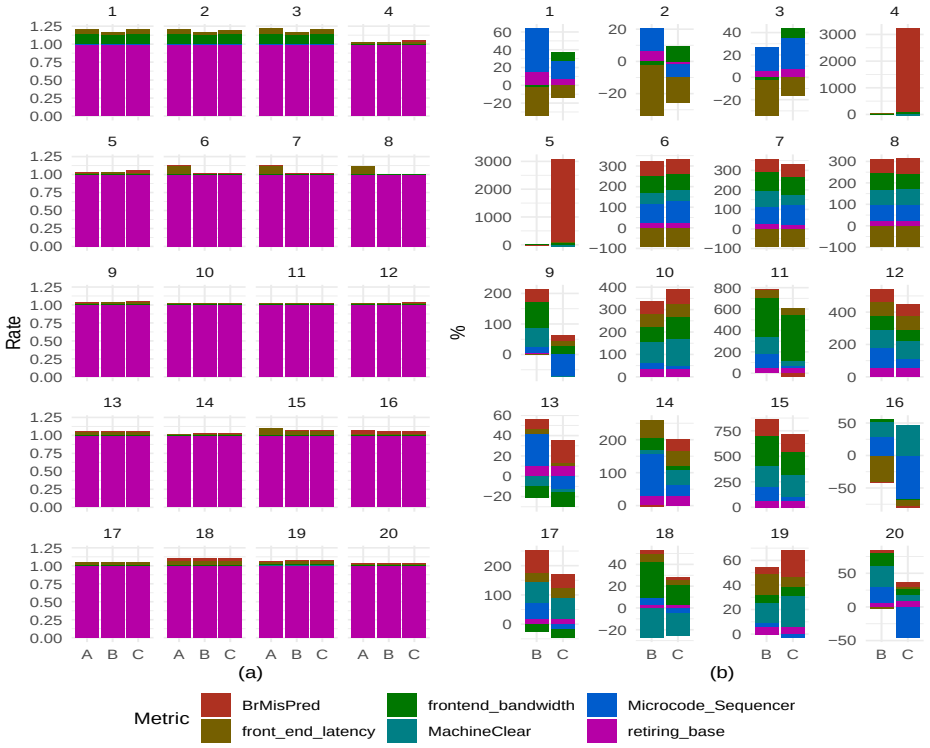


Figure 4.6: FrontEnd Bound (front_end_latency, frontend_bandwidth), Retiring (retiring_base, Microcode_Sequencer) and Bad Speculation (BrMisPred, MachineClear) subcategories for SPEC OMP2012 benchmarks: 1. val9, 2. val2, 3. val11, 4. refset2, 5. refset1, 6. convert9, 7. convert2, 8. convert11, 9. 376.kdtree, 10. 371.applu331, 11. 370.mgrid331, 12. 363.swim, 13. 362.fma3d, 14. 360.ilbdc, 15. 359.botsspar, 16. 358.botsalgn, 17. 357.bt331, 18. 352.nab, 19. 351.bwaves, 20. 350.md using compilers: A. GCC4, B. GCC6, C. GCC7. (a) is the regular metric. (b) is the PPP normalized metric.

.

value, by taking into account the cycles it took to complete the program, shows the effectiveness of the cycles used by the GCC6 and GCC7 generated binaries. In the context of Bad Speculation, GCC7 is massively overvalued – the bottleneck cost in terms of cycles is higher relative to the cycle cost when using GCC4 binaries, and GCC6 is relatively overvalued when compared to the GCC4.

- **Undervalued bottlenecks:** Undervalued rates occur when the reference compiler underperforms the other compilers in a given bottleneck metric. As a result, a bottleneck will take more GCC4 cycles than the number of cycles it takes when using a different compiler. In the case of the 358.botsalgn benchmark, we have FrontEnd Bound rates of 0.0206 for GCC4, 0.0133 for GCC6 and 0.0183 for GCC7, Figure 4.4(a). The PPP exchange rates are 1.01 for both GCC6 and GCC7. The resulting PPP normalized FrontEnd Bound rates are -35.5% for GCC6 and -11.7% for GCC7, Figure 4.4(b). For the given number of cycles, the GCC6 FrontEnd Bound bottlenecks are not as significant as compared to GCC4 and GCC7.

- **Similar bottleneck rates:** There are instances when compilers have similar Top-Down rates and their PPP normalized rates will be similar due to the PPP exchange rate. The 371.applu331 benchmark has GCC4 FrontEnd Bound rate of 0.0188, a GCC6 FrontEnd Bound rate of 0.0211 and a GCC7 FrontEnd Rate of 0.0210, Figure 4.4(a). The PPP exchange rate for GCC6 is 0.697 and 0.685 for GCC7. The resulting normalized PPP FrontEnd Bound rates for GCC6 and GCC7 are 61.5 and 63.6, Figure 4.4(b). GCC6 and GCC7 are overvalued for the FrontEnd Bound category as compared to GCC4 for the 371.applu331 benchmark.

- **Similar PPP rates:** Figure 4.5(a) shows that for 359.botsspar, GCC6 and GCC7 have similar BackEnd Bound rates. Figure 4.5(b) shows that the PPP rates are nearly identical. The CoreBound subcategory is over 400% greater than GCC4, while the External Memory and L3 Bound subcategories add up close to an additional 200% difference. We can conclude that for 359.botsspar, GCC6 and GCC7 are both significantly more BackEnd Bound than GCC4 at a relatively identical rate.

- **PPP parity (PPP rates that equal zero):** We have seen that there have been instances where the non-normalized Top-Down rates are equal, but the PPP normalized metrics showed that there was a relative difference between bottlenecks. There are cases when the metrics are equal and the PPP normalized values are zero, or close to it. In those cases, we can conclude that there is relatively little to no difference between the GCC4 baseline metric and the other compiler generated

metric. The magnitude of the bottleneck effect is similar for both compilers. For example, in Figure 4.5(a), there is little difference between GCC4 and GCC7 for val11. Figure 4.5(b) shows that the resulting PPP column for GCC7 is empty. We can conclude that the BackEnd Bound subcategories are relatively equal for GCC4 and GCC7 while there is a difference between GCC4 and GCC6 in the Core and L1 Bound subcategories of over 60%.

4.5 Conclusion

In this study, we proposed the use of normalized Top-Down bottleneck metrics using the *purchasing power parity* theory to quantify the relative difference in bottleneck metrics for different compilers, GCC4, GCC6 and GCC7. PPP based indexes have been used to track the purchasing power of many currencies by comparing specific goods. The widely known Big Mac Index is an example of a PPP metric where local prices of Big Macs from different countries are compared against US priced Big Macs. The aim is to use the US dollar as the baseline for currency comparisons. The Big Mac Index makes it possible to determine if currencies are undervalued or overvalued when compared to the purchasing power of the US dollar.

We take a similar approach by determining if bottlenecks have a relatively greater or lesser effect on performance when comparing different compilers versions using cycles as the common currency. Our method uses Top-Down bottleneck metrics of benchmark programs as the *good* that is to be compared across different versions of compilers. We use cycles as the currency for the comparison. Each compiler version has the potential of generating bottlenecks of varying magnitude for the same program. Each of these versions could potentially require a different number of cycles to run the program to completion. Our goal is to use bottleneck metrics obtained from GCC4 compiler binaries as a baseline to normalize other bottleneck metrics generated by other compilers. This normalization makes it possible to quantify relative increases or decreases in magnitude of bottlenecks when compared to the baseline compiler.

Our goal was to provide a simple technique to compare complex systems, and we found PPP normalization is a suitable technique that can accomplish this task. Our approach makes it possible for computer architects and compiler developers to track the

drift of bottlenecks by easily identifying trends, and magnify differences that otherwise would go unnoticed. Quantification of the bottleneck drift makes it possible to assess the effects changes to programs or compilers have in the effective use of CPU architectural features.

Chapter 5

Analysis of a ThunderX2 System Using Top-Down and Purchasing Power Parity Methods

In this chapter, we present a study of the Top-Down bottleneck analysis methodology as it is applied to the AArch64 platform. The Top-Down method has been widely used in the Intel platform. In this chapter, we show bottleneck behavior, as represented by the Top-Down methodology, as the number of threads increase in a ThunderX2 system. We combined the Top-Down analysis method with PPP normalization techniques to better understand the relative changes between runs as the number of threads increased [59].

5.1 Introduction

Compilers and parallel frameworks, such as OpenMP, make it easier to use complex multicore and multiprocessor platforms. Performance analysis can be challenging and time consuming, and can require multiple runs, each run potentially testing many compiler options, code changes and system configurations. The Top-Down microarchitectural analysis method is a widely used bottleneck analysis technique that can identify dominant performance bottlenecks through the use of performance counters [1]. A typical approach involves the computing of different metrics, to identify *hotspots*, portions of

the code where the number of stalls is dominant. The user then makes changes to the code or system configuration, and executes the program again. The resulting bottleneck metrics are compared to the original results, to see if there were any improvements. This process can continue until a certain level of performance is achieved, there are no further meaningful gains obtained with subsequent changes, or there are no more resources available to continue with the analysis.

The issue of using absolute rates for comparison is that they might provide an incomplete picture of the change between the rates. Relative changes, normalized with the purchasing power parity technique [22], can provide additional information on performance drift. In this study, we performed a comprehensive Top-Down bottleneck analysis, complementing with PPP normalization techniques, to quantify absolute and relative bottleneck metric changes using the OpenMP framework on a ThunderX2 system with different thread count configurations.

System characterization is a complex process with many configurable options and many benchmarks to test. Each combination will have different bottleneck characteristics resulting in varying runtimes for the same program on the same system. Our approach makes it possible to get a more complete picture of how bottlenecks, and the resulting system performance, evolve as the number of threads is increased in an AArch64 based system.

5.2 Background

Purchasing power parity (PPP) theory has been used to compare the cost of identical products from different countries, each of them with their own currencies. The Big Mac Index (BMI) is the most popular example of PPP theory. It was developed by The Economist magazine [25]. PPP based indexes can compare the strength of the currency by testing how much of an identical product a currency can buy when compared to another country's currency.

PPP theory can be used to determine the relative difference between bottleneck generated by the same benchmark but generated by different compilers. The *currency* used to compare the cost is the number of CPU cycles it took to run the program to completion. The *products* being compared are the Top-Down metrics for each benchmark.

The goal is to show that a metric value can differ or be similar to another, as defined by the Top-Down formulas, while its true cost might be relatively higher or lower to the other benchmark based on the number of cycles it took to complete the benchmark. We used the 64 thread value, $CPU\_Cycles_{64\_threads}$, as the baseline to compute the PPP_Exchange_Rate. Equation 5.1 shows the formula to compute the PPP exchange rate.

$$PPP\_Exchange\_Rate = CPU\_Cycles/CPU\_Cycles_{64\_threads} \tag{5.1}$$

Equation 5.2 computes the PPP index for all Top-Down metrics. The *64_threads* metric is the 64 thread value for the metric which was used as the baseline.

$$\begin{aligned} PPP\_Index = 100 * ((Metric/Metric_{64\_threads}) \\ -PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \end{aligned} \tag{5.2}$$

Equations 5.3 and 5.4 show an example computation of the Frontend Bound PPP indexed metric for the *360.ilbdc* benchmark. The PPP exchange rate was found to be 3.34 per Equation 5.3. This value was then used in Equation 5.4, along the ratio of the Frontend Bound metric for the 256 and 64 thread settings. The resulting PPP indexed value was found to be 297.68%.

$$PPP\_Exchange\_Rate = 281067318213792/84202928639541 = 3.34 \tag{5.3}$$

$$PPP\_Index = 100 * ((0.27873894/0.02098531) - 3.34)/3.34 = 297.68 \tag{5.4}$$

PPP rates close to 0% indicate that the difference between the $Metric$ and *Metric_64_threads* ratio, and $PPP\_Exchange\_Rate$, the numerator in Equation 5.2, is small. This results in a small PPP rate. It takes about the same number of CPU_Cycles units to achieve a similar $Metric$ rate when comparing a different thread setting to the baseline setting. When the ratio of $Metric$ and $Metric_{64\_threads}$ goes to zero, or the $PPP\_Exchange\_Rate$ is larger in magnitude, the PPP rates become negative.

The PPP rates can reach a maximum value of -100%. The $Metric$ numerator has a smaller effect when compared to its baseline when PPP indexed rates are negative. The required number of CPU_Cycles units to achieve parity with the baseline is much larger. A positive PPP rate indicates that the $PPP\_Exchange\_Rate$ is smaller than the $Metric/Metric_{64\_threads}$ rate. This takes place when the $Metric$ value is larger in magnitude than its 64 thread baseline, or the CPU_Cycles magnitude is smaller than its baseline. It needs fewer CPU_Cycles to reach parity with its baseline.

5.3 Experimental setup

We used the Top-Down method in combination with the SPEC OMP2012 benchmarks [57] to perform bottleneck analysis of the ThunderX2 AArch64 architecture using the Arm Allinea Studio armclang, armclang++ and armflang compilers version 20.0 as the number of threads was increased. The OMP2012 benchmarks were compiled using the following flags for the armclang compiler: *-O3 -ffast-math -fopenmp -fsigned-char -mcpu=native*. The armclang++ and armflang compilers used these options: *-O3 -ffast-math -fopenmp -mcpu=native*. Additionally, the *-ffree-form* option was used for *350.md* and *-mcmodel=large* for the *357.bt331* and *363.swim* benchmarks. The test system had a two socket Cavium ThunderX2 CN9980 with 32 cores per socket, 4 threads per core and 256 GB of memory. It had the CentOS Linux release 7.7.1908 (AltArch) version installed. Performance counter data was collected via the following command *perf stat -e list_of_counters* for each of the benchmarks using 5 performance counters at a time.

5.4 Results

Table 5.1: Runtimes in seconds for different thread settings

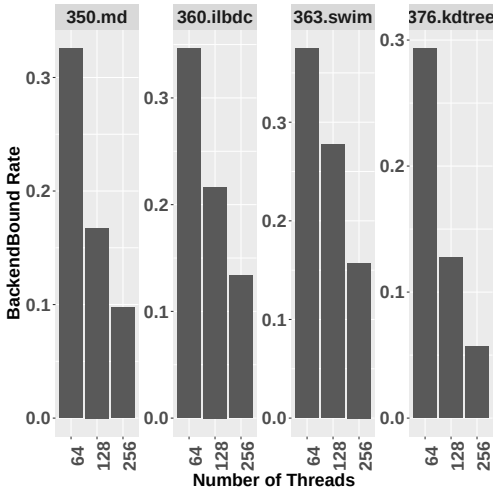
Benchmark	64	128	256
350.md	409	329	235
360.ilbdc	612	513	500
363.swim	708	707	713
376.kdtree	576	403	341

Table 5.2: Top Down Method Metric Formulas for AArch64[3]

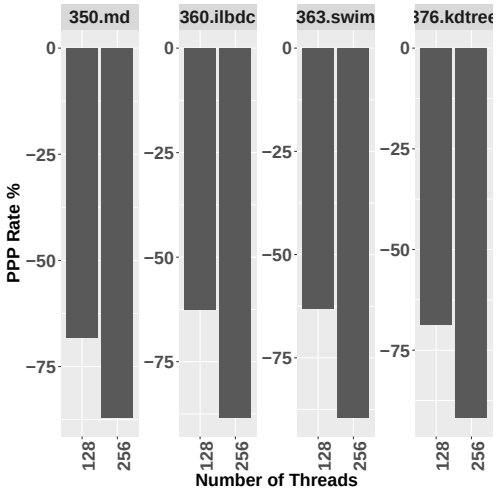
Classification	Formula
Frontend Bound	Stall_Frontend / CPU_Cycles
Backend Bound	Stall_Backend / CPU_Cycles
Allocated	(CPU_Cycles - (Stall_Frontend + Stall_Backend)) / CPU_Cycles
Bad Speculation	((Inst_Spec - Inst_Retired) / Inst_Spec) * Allocated
Retiring Bound	(Inst_Retired / Inst_Spec) * Allocated

We focused on a small subset of SPEC OMP2012 benchmarks that illustrated different Top-Down characteristics. Table 5.1 shows the different runtimes as the number of threads was increased. Because some of the benchmarks had a number of outlier results, the median values of the performance counters were used. Some of the benchmarks see performance gains with an increased number of threads while *363.swim* runtimes stayed relatively the same. This subset of benchmarks was analyzed using the Top-Down and PPP methods to obtain a more complete picture of architectural bottlenecks. Table 5.2 formulas were used to compute the Top-Down metrics. Performance counters used in the Top-Down formulas are: *CPU_Cycles*, which increments with every cycle, *Inst_Retired* that accounts for every architecturally executed instruction, and *Inst_Spec*, the counter that keeps track of speculatively executed operations [6].

The Top-Down method classifies pipeline operations as either stalled or non-stalled. Backend Bound stalls occur when no operations are issued due to the backend's inability to accept operations. Frontend Bound stalls occur when operations are not issued by the frontend resources. Allocated operations can be classified as Retiring Bound, the operation was completed, or Bad Speculation, operations whose work had to be eventually discarded. Stalls can be computed by dividing the number of stalled CPU cycles by the number of total CPU cycles. To compute the Retiring and Bad Speculation rates, the rate of non-stalled CPU cycles is computed, denoted by the Allocated metric, and then it is multiplied by the number of of Retired cycles that were speculated to compute the Retiring rate. Bad Speculation uses the Allocated metric and multiplies it by the rate of speculated instructions that were not retired.



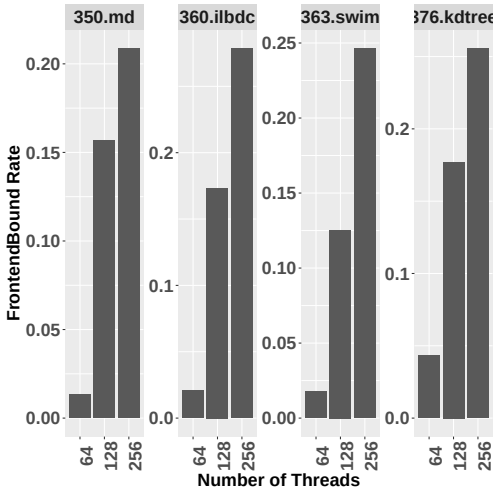
(a) Regular rates



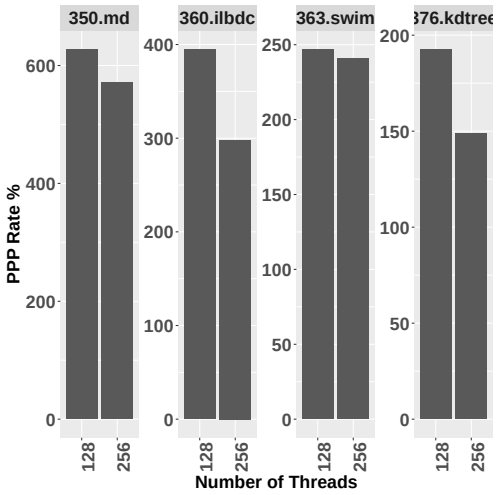
(b) PPP rates

Figure 5.1: Backend Bound

Figure 5.1(a) shows Backend Bound rates decreased as the number of threads increased. Figure 5.1(b) shows negative relative PPP rates for all benchmarks. The benchmarks had an increase of CPU cycles as the number of threads increased. The reason for the decrease in Backend Bound rates was that the number of backend stalls decreased, as it was the case for *350.md*, and *376.kdtree*, or increased at a small rate for *360.ilbdc* and *363.swim*, while CPU cycles increased for all benchmarks.



(a) Regular rates



(b) PPP rates

Figure 5.2: Frontend Bound

Frontend Bound rates increased as shown in Figure 5.2(a) due to the large increases in frontend stalls. Frontend stalls increase as the number of threads increased. The largest increases were recorded when the number of threads increased from 64 to 256. The *350.md* benchmark saw a change of 5414%. The other benchmarks had increases of 4333% for *360.ilbdc*, 3442% for *350.md* and 1287% for *376.kdtree*. The frontend stalls

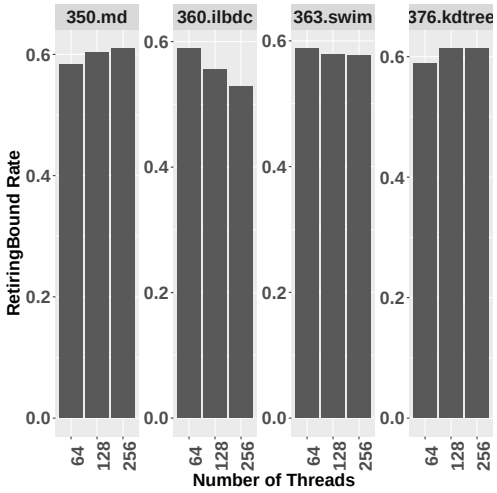
increase for the 128 thread setting was about a third of the rate increased for the 256 thread runs. The CPU cycles also increased but not as dramatically as the frontend stalls count. PPP rates show a larger rates for the 128 threads than for 256 threads. For the 128 thread run, the CPU cycles did not increase as much when compared to the 256 thread run. The large PPP rates show that CPU cycles are overpriced, when too few cycles were generated for the given number of frontend stalls. To reach a lower comparable Frontend Bound to the 64 thread run, the CPU cycle throughput needs to be increased, or the frontend stalls need to be dramatically reduced.

Figure 5.3(a) shows small changes in the Retiring metric. The *350.md* and *376.kdtree* benchmarks had an increase of 0.03 in the Retiring category. *363.swim* stayed relatively the same across all thread settings, and *306.ilbdc* had a decrease of 0.03 for 128 threads and 0.06 for 256 threads. PPP rates decreased as the number of threads increased, Figure 5.3(b). The CPU cycles rates increased along with the thread count, while the number of retired instructions decreased. While the Retiring metric might be relatively similar across the different thread settings, the PPP rates show that for the given number of CPU cycles, the number of retired instructions should increase in order to achieve real parity with the 64 thread baseline.

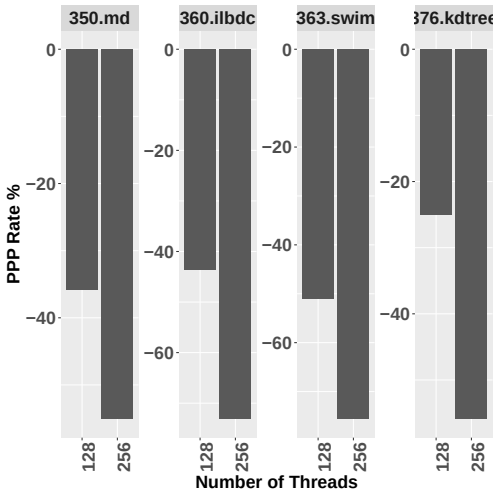
In the Bad Speculation category, there were small changes in the values of the regular rates, Figure 5.4(a). *360.ilbdc* had rates increases of 0.01 as the number of threads increased. *363.swim* stayed relatively the same, while *376.kdtree* and *350.md* saw variations with a magnitude of 0.01. PPP indexed rates showed increasing negative rates as the number of threads was increased. While the regular rates showed comparable behavior across the different thread settings, PPP rates show that bad speculations decreased as the number of threads increased relative to the CPU cycles it took to run the program.

5.5 Conclusion

Our study showed the results of an in-depth bottleneck analysis of a highly threaded multicore system to quantify the absolute and relative differences between thread settings. We showed that standard techniques that based their finding in absolute values might miss information that relative changes, when normalized using PPP tecniques,



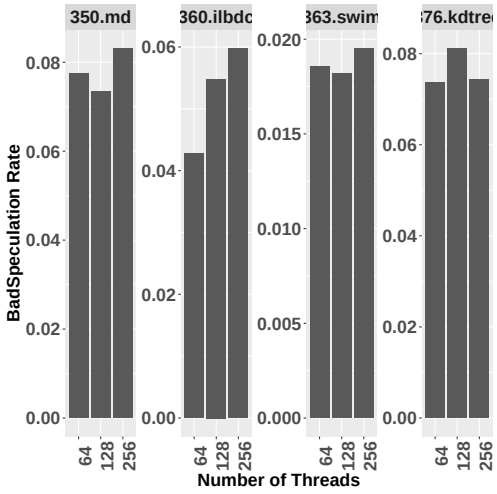
(a) Regular rates



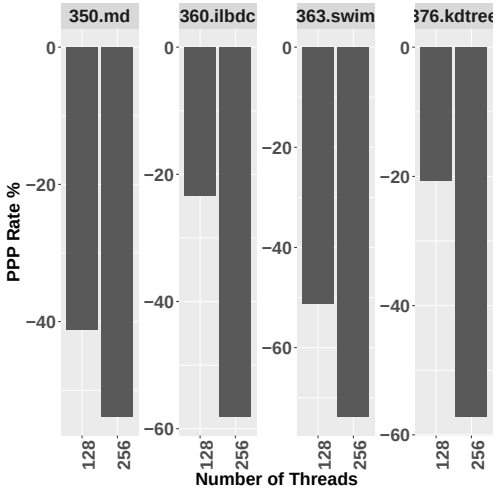
(b) PPP rates

Figure 5.3: Retiring Bound

can provide. PPP normalized rates make it possible to gain an in-depth understanding of the performance implications of different system configurations and aids in the selection of the right setting for a given situation.



(a) Regular rates



(b) PPP rates

Figure 5.4: Bad Speculation

Chapter 6

Analysis of Cache, CPU and Bottleneck Metrics in a Multithreaded AArch64 System

Performance metrics gives us information of how underlying components of the system behave as changes are made to the code, the system, or the compilers options being used to generate the binary program. Ideally, effects generated by changes can be readily identified by changes in the performance ratios. In this chapter, we show that through the use of PPP normalized rates, we can highlight changes and trends that rates hide when comparing regular standard rates.

6.1 Introduction

Recent trends in CPU designs have increased the number of cores to increase CPU performance. These multi-threaded and multicore CPUs allow for better hardware utilization [8]. Multiple levels of caches, each with different sizes and characteristics, make it possible to maximize the amount of data that is provided to the CPU in the least amount of time. To quantify cache and overall system performance, metrics such as instructions-per-cycle (IPC), cache misses, and branch misses are used.

The problem with the use of absolute rates in metrics is that it might provide an

incomplete picture of the change between rates. A suggested approach is to compare ratios such as cycles-per-instructions (CPI) only when *"one part of the ratio is changing"* [21]. This approach is not feasible when changes, such as the number of threads used to run a program, affects both parts of the ratio. In the case of the CPI ratio, the number of CPU cycles and the number of retired instructions – architecturally executed instructions – change when the number of threads are varied. A more practical approach is to normalize rates with a reference baseline to quantify the relative performance metric changes as the number of threads is increased. Relative changes, normalized with the purchasing power parity (PPP) technique [22], can provide additional information on the metric drift. There are other techniques such as the Roofline model [23] that gives users an idea of how their code is performing when compared to memory and floating-point peak performance. Other approaches include the use of statistical methods to model performance based on metrics such as cache hit rates and memory latencies [24]. Both of these approaches give users insight into how performance is affected as different settings or features are changed, but they are specific to the parameters that are used in the statistical model, or the few metrics that are used to determine peak performance. We propose a more general technique that can be applied to any performance metric rate, and can give enough information to determine the effect an increase in the number of threads had in the metric.

In this study, we performed an analysis of CPU and cache performance metrics using PPP indexed rates to quantify absolute and relative metric changes as the number of threads was increased. Our study makes the following contributions:

- We show that, while absolute metric rates might be similar, relative rates can vary.
- We show that absolute performance metric comparisons tend to hide large magnitude changes of performance counter values relative to baseline metrics.
- We identify trends across different metric categories, and identify differences that might have otherwise gone unnoticed by standard evaluation practices.

Performance analysis and system characterization are a complex process with many settings to try and versions of benchmarks to test. Each of these combinations will have

different cache and CPU characteristics resulting in varying runtimes as the number of threads is changed. Our approach makes it possible to obtain a more complete picture of how underlying hardware performs as the number of threads is increased to better understand what portions of the code, or system components, need to be further scrutinized for potential resource utilization improvements.

6.2 Background

6.2.1 CPU Caches

Caches make it possible for CPUs to have readily available data for computation. They can have multiple levels, each with its own characteristics and features. Many techniques have been devised to increase the performance of caches [60], they include:

- Reduction of hit times through the use of smaller caches with simple designs.
- Increase of cache bandwidth with features such as pipelined and non-blocking caches.
- Reduction of miss penalties through the use of write buffers, loop interchange and blocking techniques.
- Reduction of miss rates through the use of hardware and software prefetching techniques.

Metrics, such as cache miss rate, cache bandwidth and miss penalty, are used to assess the performance of the different cache levels. However, there are some issues that should be taken into account when using these metrics [61]:

- *Ratios might be misleading.* A cache miss will be accurately counted but subsequent accesses might be counted as hits before the data has actually been loaded into the cache.
- *Cache miss ratios might not be predictive of performance.* The authors found that L1 and L2 miss rates were not predictive of L3 or DRAM miss rates.

Table 6.1: ThunderX2 performance counters [4] [5]

Event	Description
armv8_pmuv3_0/inst_retired/	Instruction architecturally executed.
armv8_pmuv3_0/ld_retired/	Instruction architecturally executed, condition code check pass, load.
mem_access_rd	Data memory access, read.
mem_access_wr	Data memory access, write.
armv8_pmuv3_0/mem_access/	Data memory access.
armv8_pmuv3_0/l1d_cache/	Level 1 data cache access.
l1d_cache_rd	Level 1 data cache access, read.
l1d_cache_wr	Level 1 data cache access, write.
l1d_cache_refill_rd	Level 1 data cache refill, read.
l1d_cache_refill_wr	Level 1 data cache refill, write.
armv8_pmuv3_0/l2d_cache/	Level 2 data/unified cache access.
armv8_pmuv3_0/l2d_cache_refill/	Level 2 data/unified cache refill.
armv8_pmuv3_0/cpu_cycles/	Cycle.
armv8_pmuv3_0/l2d_cache_wb/	Level 2 data/unified cache writeback.
armv8_pmuv3_0/l1d_cache_wb/	Level 1 data cache write-back.
L1-dcache-load-misses	Perf hardware cache event.
L1-dcache-loads	Perf hardware cache event.
L1-dcache-store-misses	Perf hardware cache event.
L1-dcache-stores	Perf hardware cache event.

In this study, we analyzed the behavior of different caches as the number of threads was increased from 64, to 128 and to 256 on a Cavium ThunderX2 system. We use the different performance counters shown in Table 6.1 that are available via the *perf* profiling tool. To get a more complete picture of cache behavior, we studied the performance counters that track write, read, write-back, and refill cache operations. Table 6.2 shows the metrics that were used to analyze the performance of the different cache features. We use the purchasing power parity (PPP) technique to quantify relative changes between the ratios as the number of threads was increased. The PPP technique is discussed next.

Table 6.2: Memory and Computing Ratios [6]

	Name	Ratio
1	Instructions per cycle (IPC)	armv8_pmuv3_0/inst_retired/ / armv8_pmuv3_0/cpu_cycles/
2	Loads per cycle (LPC)	armv8_pmuv3_0/ld_retired/ / armv8_pmuv3_0/cpu_cycles/
3	Loads per instruction (LPI)	armv8_pmuv3_0/ld_retired/ / armv8_pmuv3_0/inst_retired/
4	L1D cache load miss	L1-dcache-load-misses / L1-dcache-loads
5	L1D cache store miss	L1-dcache-store-misses / L1-dcache-stores
6	L1D cache read	l1d_cache_rd / armv8_pmuv3_0/l1d_cache/
7	L1D cache write	l1d_cache_wr / armv8_pmuv3_0/l1d_cache/
8	L1D cache refill write	l1d_cache_refill_wr / l1d_cache_wr
9	L1D cache refill read	l1d_cache_refill_rd / l1d_cache_rd
10	L1D cache write-back	armv8_pmuv3_0/l1d_cache_wb/ / armv8_pmuv3_0/l1d_cache/
11	L2D cache refill	armv8_pmuv3_0/l2d_cache_refill/ / armv8_pmuv3_0/l2d_cache/
12	L2D cache write-back	armv8_pmuv3_0/l2d_cache_wb/ / armv8_pmuv3_0/l2d_cache/
13	Memory access read	mem_access_rd / armv8_pmuv3_0/mem_access/
14	Memory access write	mem_access_wr / armv8_pmuv3_0/mem_access/

6.2.2 Purchasing Power Parity

Purchasing power parity (PPP) theory underlies different methods to compare the cost of identical products, such as lattes, and iPods, between different countries, each of them with different currencies [22], [56], [62]. The most famous index is the Big Mac Index (BMI), which was developed by The Economist magazine [25]. The goal of these

PPP based indexes is to compare the strength of the currency by testing how much of the same product a currency can buy when compared to another currency. A currency is overvalued – when the product bought using that currency is more expensive – or undervalued – when the product is cheaper – when compared to a base currency.

The following is an illustrative example of the purchasing power of the Chinese yuan versus the US dollar as described in The Economist magazine. For this example, the dollar to yuan exchange rate is \$1 = 6.4 yuan. The quoted Big Mac price was \$5 and 20 yuan. Equation 6.1 computes the Big Mac exchange rate which is based on its local price.

$$20/5 = 4 \tag{6.1}$$

Equation 6.2 computes the Big Mac Index. It shows that the yuan is 37.5% undervalued as compared to the US dollar.

$$(4 - 6.4) * 100/6.4 = -37.5 \tag{6.2}$$

PPP theory can be used to determine the relative difference between performance metric ratios, such as IPC, or cache misses. These metrics are generated by the same benchmark and same compiler but using a different number of threads when run. To compute the PPP index rate, we first compute the *PPP exchange rate*. This rate uses the *perf* reported performance counter value that is used as the denominator in the metric computation. In our study, we used the value of the 64 thread counter as the baseline. The numerator will be either the 128 or 256 thread value of the same counter as shown in Equation 6.3.

$$PPP\_Exchange\_Rate = Counter\_Denominator/Counter\_Denominator_{baseline} \tag{6.3}$$

The metric ratio approach used in Equation 6.4 is similar to the one used in the exchange rate computation. We divided the metric rate of the 128 or 256 thread run by the baseline 64 thread metric value. We then used the exchange rate from Equation 6.3 to

compute the PPP_Index rate.

$$PPP\_Index = 100 * ((Metric/Metric_{baseline}) \\ -PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \tag{6.4}$$

Equations 6.5 and 6.6 show an example computation of the PPP index for the IPC metric of the *363.swim* benchmark to compare IPC results between 64 and 128 threads. We use the number of CPU cycles as reported by the *perf* tool. Equation 6.5 computes the PPP exchange rate, using the number of cycles generated by the 128 thread run, 175899129324168, and dividing it by the baseline 64 thread run CPU cycles count, 87329901733972. The resulting ratio is then used in Equation 6.6.

$$175899129324168/87329901733972 = 2.014191 \tag{6.5}$$

$$100 * ((0.3677932/1.1645002) - 2.014191)/2.014191 = -84.3\% \tag{6.6}$$

Equation 6.6 was used to compare the IPC values of the 64 and 128 thread runs, 0.36 and 1.16 respectively. Based on the resulting PPP index, we can conclude that the 128 thread run of *363.swim* is 84% undervalued as compared to the baseline. It takes more CPU cycles when using 128 threads to achieve a similar IPC rate to the 64 thread baseline run. An alternative comparison approach is to compute the percent difference between the two IPC rates. Computing the percent difference between 128 and 64 thread IPC rates results in a -68% difference. This shows that the 128 thread IPC rate is about two thirds the value of the 64 thread IPC rate. This result is not as informative as the computed PPP index since it can't be determined how the percentage difference relates to the *armv8_pmuv3_0/cpu_cycles/* or *armv8_pmuv3_0/inst_retired/* performance counter measurements.

We can summarize the meaning of the PPP indexed rates as follows: PPP rates close to 0% indicate that the difference between $Metric/Metric_{baseline}$ and *PPP_Exchange_Rate*, numerator in Equation 6.4 is minimal, resulting in a small PPP rate. This means that it takes about the same amount of *Counter_Denominator* units to achieve a similar *Metric* rate when comparing a different thread setting to the baseline setting. As the value of $Metric/Metric_{baseline}$ goes to zero, or the *PPP_Exchange_Rate* becomes

comparatively larger in magnitude, the PPP rates will become negative. It could potentially reach a maximum value of -100%. Negative PPP rates indicate that $Metric$ has a relatively smaller effect when compared to its baseline. Relatively speaking, the number of $Counter\_Denominator$ units to achieve parity with the baseline is much larger. A positive PPP rate is achieved when the $PPP\_Exchange\_Rate$ is smaller than the $Metric/Metric_{baseline}$ rate. In this case, the $Metric$ value can be larger in magnitude than its baseline, or the $Counter\_Denominator$ value is smaller than its baseline. It needs fewer $Counter\_Denominator$ units to reach parity with its baseline.

Depending on the type of metric being analyzed, a metric might provide useful information to improve performance with a lower relative rate, such as cache misses, while others could see performance benefits if its relative rate is higher, such as IPC. PPP index rates make it easier to identify differences in performance metrics when a new setting is compared to its baseline. In this study, the PPP technique was used to compare different performance metric rates in order to get a more complete picture of CPU and cache behavior as the number of threads was increased. We identified differences and trends that could have gone unnoticed if just regular metrics had been used for the analysis.

6.3 Experimental setup

We used the SPEC OMP2012 benchmarks [57], [2] to evaluate performance and cache metrics of the Arm Allinea Studio armclang, armclang++ and armflang compilers version 20.0. The OMP2012 benchmarks were compiled using the following flags for the armclang compiler: *-O3 -ffast-math -fopenmp -fsigned-char -mcpu=native*. The armclang++ and armflang compilers used these options: *-O3 -ffast-math -fopenmp -mcpu=native*. Additionally, the *-ffree-form* option was used for *350.md* and *-mcmodel= large* for the *357.bt331* and *363.swim* benchmarks. The SPEC OMP2012 benchmarks are described in Table 6.3. Benchmarks results that followed the SPEC group reporting guidelines can be published in a publicly available repository [58]. Many vendors and computing centers publish results for public comparisons. We used the default OpenMP settings as described in Table 6.4.

A two socket Cavium ThunderX2 CN9980 with 32 cores per socket, 4 threads per

Table 6.3: SPEC OMP2012 Benchmark Description [2]

350.md	Fortran	Physics: Molecular Dynamics
351.bwaves	Fortran	Physics: Computational Fluid Dynamics (CFD)
352.nab	C	Molecular Modeling
357.bt331	Fortran	Physics: Computational Fluid Dynamics (CFD)
358.botsalgn	C	Protein Alignment
359.botsspar	C	Sparse LU
360.ilbdc	Fortran	Lattice Boltzmann
362.fma3d	Fortran	Mechanical Response Simulation
363.swim	Fortran	Weather Prediction
367.imagick	C	Image Processing
370.mgrid331	Fortran	Physics: Computational Fluid Dynamics (CFD)
371.applu331	Fortran	Physics: Computational Fluid Dynamics (CFD)
372.smithwa	C	Optimal Pattern Matching
376.kdtree	C++	Sorting and Searching

Table 6.4: Arm Allinea Studio version 20.0 default OpenMP settings

Setting	Value
OMP_ALLOCATOR	omp_default_mem_alloc
OMP_CANCELLATION	FALSE
OMP_DEFAULT_DEVICE	0
OMP_DISPLAY_AFFINITY	FALSE
OMP_DISPLAY_ENV	TRUE
OMP_DYNAMIC	FALSE
OMP_MAX_ACTIVE_LEVELS	1
OMP_MAX_TASK_PRIORITY	0
OMP_NUM_THREADS	256
OMP_PLACES	value is not defined
OMP_PROC_BIND	false
OMP_SCHEDULE	static
OMP_STACKSIZE	16M
OMP_TARGET_OFFLOAD	DEFAULT
OMP_THREAD_LIMIT	2147483647
OMP_TOOL	enabled
OMP_WAIT_POLICY	PASSIVE

core, and 256 GB of memory was used running the CentOS Linux release 7.7.1908 (AltArch) version. Each core has the following cache hierarchy: 32 KB L1d, 32 KB L1i, and 256 KB L2. Each CPU includes a 32MB distributed L3 cache. Table 6.1 shows the counters that were collected via the following command *perf stat -e list_of_counters* for each of the benchmarks using 5 performance counters at a time.

6.4 Results

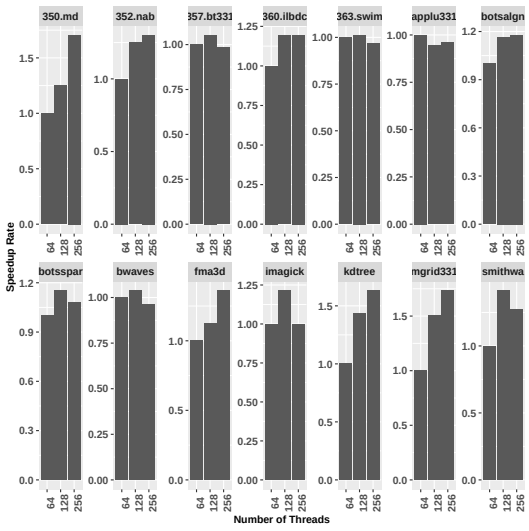


Figure 6.1: Speedup results for the SPEC OMP2012 benchmark using the Arm Allinea Studio 20.0 compilers.

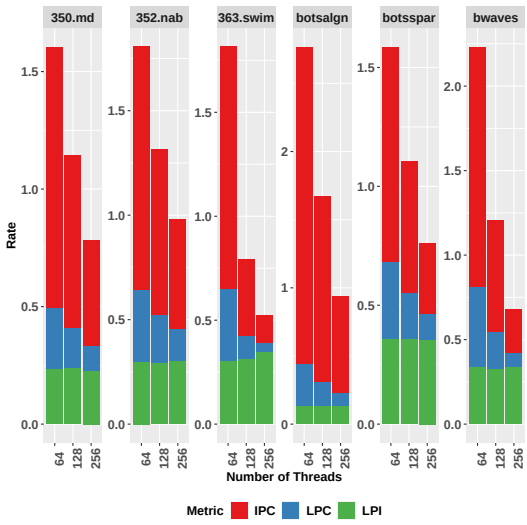
Figure 6.1 shows benchmark speedups using the reported runtimes provided by the SPEC OMP2012 reporting tool. Most benchmarks show an increase in performance as the number of threads increase. A few benchmarks showed significant speedup such as *350.md*, *376.kdtree*, and *370.mgrid331*. Others showed a decrease in performance as the number of threads were increased, such as *371.applu331*. There were a few benchmarks like, *367.imagick*, which saw performance gains when the number of threads was increased from 64 to 128, and a decrease when the threads were increased to 256. Performance metrics derived from performance counters were used to understand the behavior of the different components that lead to different performance results. In this study, a subset of the SPEC OMP2012 suite was analyzed using the different

metrics depicted in Table 6.3. *350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar* and *358.botsalgn* benchmarks were selected because of their varying speed up rates which translated into different performance metric patterns. The PPP indexing technique was used to understand the overall behavior of the system, and the system's cache hierarchy in particular.

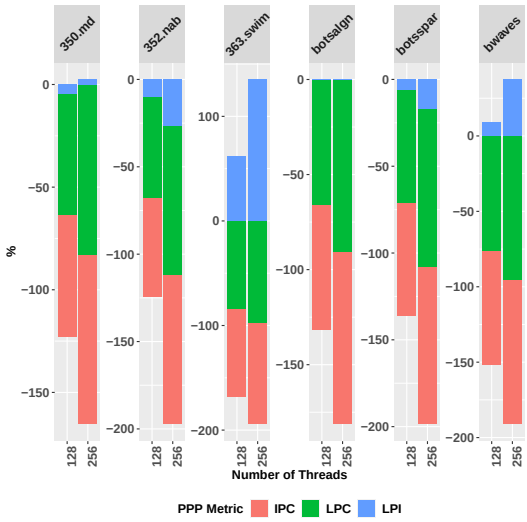
6.4.1 IPC, LPC, and LPI Rates

Instructions-per-cycle (IPC) and loads-per-cycle (LPC) rates are used to characterize overall application performance. These two metrics give an idea of the amount of work the CPU and memory systems are achieving [61]. Figure 6.2(a) shows that IPC and LPC rates decreased as the number of threads increased. This was due mostly by the large increased in CPU cycles. The *351.bwaves* benchmark had a 301% increase for 256 threads and 91% for 128 threads in the CPU cycles count. The other large increase in CPU cycles was observed in *363.swim* with a 320% increase for 256 threads and 101% for 128 threads. At the same time, the number of retired instructions and loads decreased by 27% for 256 threads and almost 12% for 128 threads for the *351.bwaves* benchmark. *363.swim* recorded a drop in retired instructions of 51% for 256 threads and 36% for 128 threads. The loads count fell by 34% for 128 threads and 44% for 256 threads. Figure 6.2(b) shows large negative rates for the PPP indexes across the benchmarks. This drop in rates was attributed to the increase in CPU cycles as the number of threads increased. The negative PPP rates indicate that for the higher thread counts, their corresponding CPU cycles are undervalued. The 128 and 256 thread runs are using more CPU cycles but are doing less work as compared to the 64 thread baseline runs. In order to achieve the higher baseline IPC and LPC rates, 128 and 256 threads runs needed to increase the number of retired instructions.

The loads-per-instructions (LPI) gives us a sense of the amounts of loads for a program. For the benchmark set that was studied, the LPI rates were very similar across benchmarks, except for *363.swim*. For this benchmark, LPI rates for 64 and 128 threads were 0.30, while the 256 thread run had a rate of 0.34. The reason was that the instruction retired count dropped by 51% while the retired loads count dropped by 44%. The 128 thread run had count drops of 36% for retired instructions and 34% for retired loads. The other benchmarks had similar increase and decrease count fluctuations of the



(a) Regular rates



(b) PPP index

Figure 6.2: IPC, LPC, and LPI regular and PPP indexed rates for 350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar, 358.botsalgn benchmarks using 64, 128 and 256 threads.

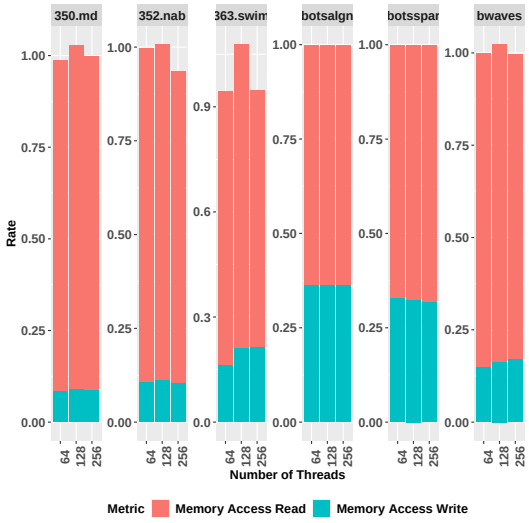
retired loads and instructions counters, which resulted in very similar LPI rates across the different thread counts. When the retired loads and instructions counts dropped,

the PPP indexed had a positive rate, which was the case of the *363.swim* benchmark. In this case, the retired instructions for the 126 and 256 thread runs were overvalued, meaning that you needed fewer retired instructions for the amount of retired loads to achieve the same LPI rate that was obtained by the 64 thread baseline run. When the counts increased, the PPP indexed had a negative rate. The retired instructions are undervalued, and you need more retired instructions for the given number of retired loads to achieve similar rates as the baseline. This was the case for the *352.nab* benchmark. There were cases when the LPI rate remained the same across the three different thread counts. The *358.botsalgn* benchmark had similar LPI rates because there was little change in the retired loads and instructions counts as the number of threads was increased. This resulted in PPP rates that were close to 0%.

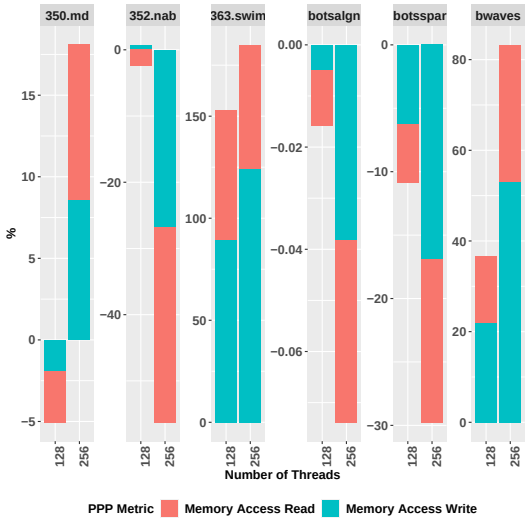
LPI rates *"can effectively double as compared to the load/store architecture"*, because the x86 instruction set architecture makes it possible for most instructions to include a memory operand [61]. In this study, we did not see a doubling of LPI rates as compared to load and store metrics, but we identified common trends in the PPP indexed rates for the LPI metric shown in Figure 6.2(b) and the memory access reads and writes in Figure 6.3(b). The PPP rates differ in magnitude, but the trends was the same. *358.botsalgn* had LPI PPP rates very close to 0%, while its memory access reads and writes were less than 0.04% each. *350.md* had negative PPP rates for its 128 thread runs, and a positive rate for its 256 thread runs. *352.nab* and *359.botsspar* had negative rates for both thread counts, while *351.bwaves* and *363.swim* had positive rates for both 128 and 256 threads. The reason for the identical trends were the magnitude changes of the counters. *armv8_pmuv3_0/ld_retired/*, *mem_access_rd*, *mem_access_wr*, *mem_access*, and *armv8_pmuv3_0/inst_retired/* increased or decreased in similar fashion for the same thread counts in the benchmarks. Just like in the case of the x86 ISA, the number of instructions with memory operand functionality in the AArch64 architecture will correlate with the number of memory access write and read operations.

6.4.2 Memory Access Metrics

We used memory access read and write metrics to get an overall picture of the number of read and write operations that were made by the CPU's cores. *armv8_pmuv3_0/mem_*



(a) Memory access RD and WR rates.



(b) PPP index

Figure 6.3: Memory access metrics for 350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar, 358.botsalgn benchmarks using 64, 128 and 256 threads.

access/ counts all memory read and write requests made by a CPU's core. Write-back and refill operations to any of the caches are not counted. *mem_access_rd* and

mem_access_wr track read and write specific operations respectively. Figure 6.3(a) shows the ratio of memory access read and writes divided by overall memory accesses as described in Table 6.2. We can see that memory access writes stayed consistent for most benchmarks, only *363.swim* showed a 0.05 rate increase from 64 threads to 128 and 256 threads. The memory access read ratios dropped for *352.nab*. It saw a drop of 0.06 from 64 and 128 to 256 threads. The *363.swim* benchmark memory access read rate increased when using 128 threads, while 64 and 256 threads had lower rates. Memory access writes were slightly higher for 128 and 256 threads.

Figure 6.3(b) shows that the *358.botsalgn* benchmark had few changes in the memory access rates as the number of threads increased as reported by the low PPP indexed rates. All other benchmarks experienced larger PPP rates for the 256 thread runs. *363.swim* had the largest memory access write PPP rate changes, 89%, for 128 threads and 124% for 256 threads. Their respective memory access write regular rates were reported as similar, 0.21. The difference in their PPP indexed rates can be explained by the larger count drop of memory access reads and writes for 256 threads. Memory access counts dropped by 32% for 128 threads while the 256 thread run had 41% less memory accesses. Memory access writes dropped by 13% for 128 threads and 23% for the 256 thread run. As the number of threads was increased, memory accesses and memory accesses write dropped proportionally, thus resulting in similar regular rates but different PPP rates. The corresponding memory access read counts for *363.swim* dropped by 24% for 128 threads and 45% for 256 threads, resulting in memory access read rates of 0.86 for 128 threads and 0.73 for 256 threads. PPP rates show a 63% increase for 128 threads and 60% for 256 threads. The count drop in memory access reads was higher than memory access writes, resulting in only double digit PPP indexes as compared to the much larger PPP indexes for memory access writes.

The positive PPP rates for *350.md* 256 threads and *351.bwaves* can be attributed to the count drop for all counts in memory accesses, memory access reads, and memory access writes. It took fewer memory accesses to achieve the same memory access read and write ratios, so they became more expensive than the baseline 64 thread runs. The opposite was seen for *359.botsspar*, *352.nab* and *350.md* when using 128 threads. The values for all three performance counters increased, thus making memory accesses less valuable. You need more memory accesses to achieve the same memory write and read

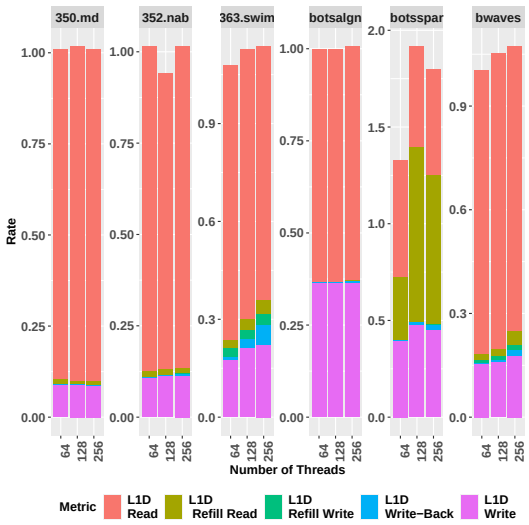
metric rates of the 64 thread baseline. *358.botsalgn* saw a small drop in PPP rates because there were little changes in the magnitude of the three counters as the number of threads increased.

6.4.3 L1 Metrics

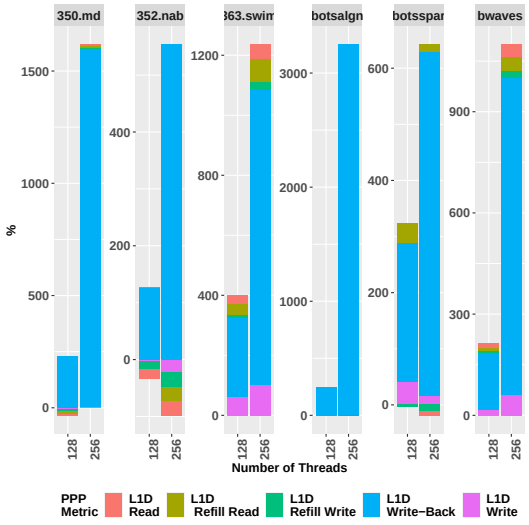
The performance counter *armv8_pmuv3_0/l1d_cache/* accounts for each read and write operation that generates a cache access operation on the L1 data cache. Accesses made to the refill, and write-back buffers are also counted. *l1d_cache_wr* and *l1d_cache_rd* counters track L1 cache write and read operations respectively. *l1d_cache_refill_rd* and *l1d_cache_refill_wr* track read and write operation which causes a refill request to at least the L1 cache. These requests include any access that causes data to be fetched from outside the L1 cache, even when the data is not ultimately allocated in the cache. *armv8_pmuv3_0/l1d_wb/* counts each write-back that causes L1 cache data to be written to another cache in the memory hierarchy.

Figure 6.4(b) shows the large increase in the PPP indexed rates for L1D cache write-backs. The percentage difference was rather large between 64 and 256 thread runs. Figure 6.4(a) shows that the 256 thread runs for *363.swim*, *359.botsspar* and *351.bwaves* have a visible write-back rate. Percentage differences of the *armv8_pmuv3_0/l1d_cache_wb/* increased with the number of threads as compared to the 64 thread baseline regardless of the magnitude changes of the other *armv8_pmuv3_0/l1d_cache* related metrics. Increases in the write-back metric are to be expected. An increasing number of threads can generate more writes, as shown in Figures 6.3 and 6.4, and these updates will generate additional coherency operations. The high PPP rates show the increased cost in terms of *armv8_pmuv3_0/l1d_cache*, as the *armv8_pmuv3_0/l1d_cache_wb/* incident count increases. It takes many more 64 thread generated *l1d_cache* events, to achieve the same write-back rate of the 128 and 256 thread runs.

Figure 6.4(a) shows that L1D cache refill reads can vary significantly for *359.botsspar*, *363.swim*, and *351.bwaves*. The PPP rates for *359.botsspar* are 35% and 12% for 128 and 256 threads respectively. The number of L1D cache refill events increased by over 100% for both, 128 and 256 threads, while the number of recorded L1D cache reads fell by 35% for 128 threads and 12% for 256 threads. PPP indexed rates for *363.swim* show that there was an increase of 35% for 128 threads and 75% for 256 threads. For this



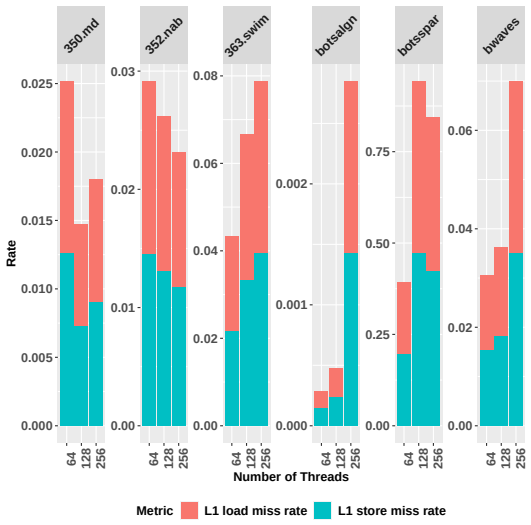
(a) Regular rates



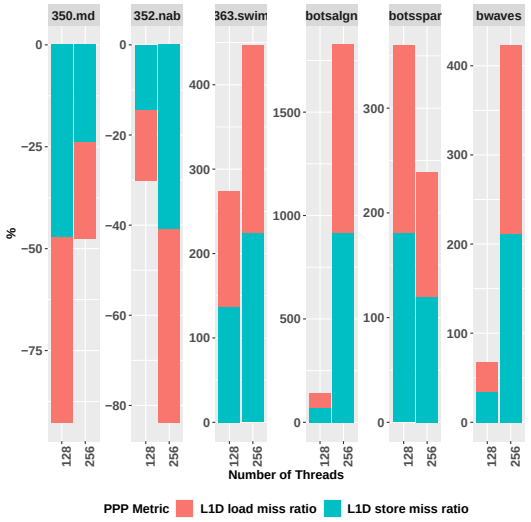
(b) PPP index

Figure 6.4: L1D cache metrics for 350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar, 358.botsalgn benchmarks using 64, 128 and 256 threads.

benchmark the number of refill read events stayed the same, while the number of L1D cache reads decreased for both by 35% for 128 threads, and 75% for 256 threads.



(a) Miss and store rates



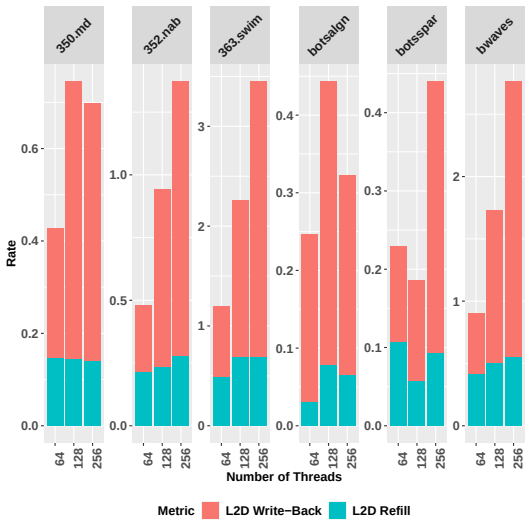
(b) PPP index

Figure 6.5: L1 data cache metrics for 350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar, 358.botsalgn benchmarks using 64, 128 and 256 threads.

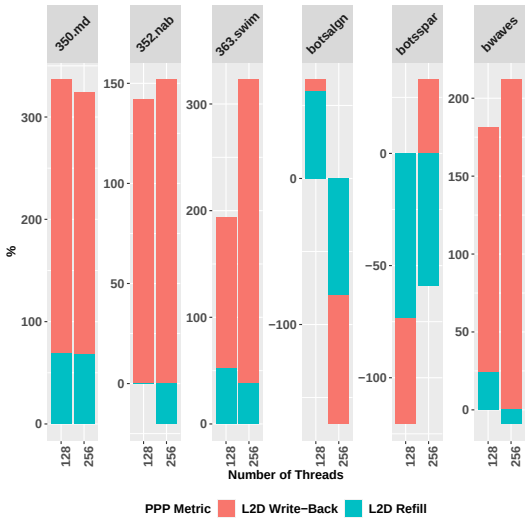
Figure 6.5(a) shows the L1 data cache store miss and load miss rates. *350.md* store and load misses dropped by 37% and 35% respectively for 128 threads, while the number

of loads and stores increased by about 9%. When the thread count was increased to 256, the number of load and store misses fell by about 33% while the number of loads and stores fell by 6%. The PPP indexed load and store miss rates, Figure 6.5(b), are lower for 128 threads, −45% for load misses and −47% store misses, than the 256 thread results, −23% for both rates. 128 thread load and stores are less valuable than 256 threads because it took more of them to achieve similar miss rates when compared to the 64 thread baseline. In the case of the *352.nab* benchmark, there was a reduction of the load and store miss rates. This was achieved by the increase of load and store operations, 5% for 128 threads and 36% for 256 threads. The number of misses fell by about 5% for 128 threads and increased by 10% for store misses and 6% for load misses when using 256 threads. PPP rates are lower for 256 than 128 threads because it takes more store and load operations in 256 threads to achieve the low store and load miss rates.

The higher load and store miss rates for *359.botsspar*, *351.bwaves*, *358.botsalgn* and *363.swim* can be attributed to the increase in misses, the decrease in load and store operations, or both, as the number of threads are increased. *358.botsalgn* had a 70% increase in load and store misses for 128 threads and 913% increase in store misses and 908% increase in load misses. *363.swim* had a relative same number of store and load misses, while the number of store and load operations fell by 34% for 128 threads and 43% for 256 threads. The corresponding PPP indexed rates are large, particularly for *358.botsalgn*. This benchmark had a PPP indexed load and store miss rate of more than 69% for 128 threads and over a 910% increase for 256 threads. For the *359.botsspar* benchmark, the number of store and load operations dropped by 14% when using 128 threads and 2% for 256 threads, while the number of stores and load misses increased by about 105% for 128 threads and 110% for 256 threads. The corresponding PPP indexed miss rates were higher for 128 threads, close to 180%, and 119% for 256 threads. The large positive PPP indexed rates indicate that the *l1-dcache-stores* and *l1-dcache-loads* operations are overvalued. It takes fewer of them to achieve similar L1D cache store and store miss rates as compared to the 64 thread baseline.



(a) Regular rates



(b) PPP index

Figure 6.6: L2D cache refill and write-back metrics for 350.md, 352.nab, 363.swim, 351.bwaves, 359.botsspar, 358.botsalgn benchmarks using 64, 128 and 256 threads.

6.4.4 L2 Metrics

The *armv8_pmuv3_0/l2d_cache/* performance counter tracks read and write operations that cause a cache access to at least the L2 cache. All accesses to a cache line are counted including refills and write-backs from the L1 data, instruction or unified caches. Accesses to other L2 components such as refill, write-back buffers are also counted. *armv8_pmuv3_0/l2d_refill/* counts accesses counted by the L2 cache that cause a refill of a refill demand of the L1 or L2 cache from outside the L1 or L2 caches. *armv8_pmuv3_0/ l2d_wb/* counts write-backs that cause data to be written from the L2 cache to outside the L1 or L2 cache.

Figure 6.6(a) shows that the L2D write-backs were higher for 128 and 256 threads as compared to 64 threads. For the *350.md* and *358.botsalgn* benchmarks, the rates were higher only for the 128 thread runs. The L2D refill rates also varied as the number of threads was increased. But the variation was not as pronounced as the write back metrics. The refill rates were similar across thread counts for *350.md*, and increased when compared to 64 threads for *352.nab*, *363.swim*, *359.botsalgn* and *351.bwaves*. The refill rate decreased for *359.botsspar*. The PPP indexed rates shown in Figure 6.6(b) show that the write-back vary negatively only for *358.botsalgn* when 256 threads are used and *359.botsspar* for 128 threads. In both instances, you had large increases in the number of *armv8_pmuv3_0/l2d_cache/* and *armv8_pmuv3_0/l2d_cache_wb/* counts, resulting in very similar L2D cache write-back rates. Negative PPP indexed rates reflect the fact that it takes many more *armv8_pmuv3_0/l2d_cache_wb/* events to reach parity with their 64 thread baseline. For the positive PPP indexed write-back rates, a much larger *l2d_cache_wb* count as compared to the *l2d_cache* denominator, resulted in larger write-back metrics. In this case, the higher thread count runs were overvalued, it takes a smaller *l2d_cache* count to reach a similar write-back rate as the baseline because of the increased in the magnitude of *l2d_cache_wb*.

The positive PPP rates for *350.md* and *363.swim*, Figure 6.6(b), are the result of higher rates for the refill metric. In the case of *350.md*, the *l2d_cache* and the *l2d_cache_refill* counts decreased at about the same rate, between 41% to 47%, as the number of threads increased. *363.swim* saw an increase in the *l2d_cache_refill* count while the number of *l2d_cache* instances remained the same or slightly lower. PPP rates for *358.botsalgn* and *351.bwaves* were positive for 128 threads and negative for 256

threads. The positive PPP indexed rate can be attributed to the higher *l2d_cache_refill* while the *l2d_cache* decreased or increased at a smaller rate. For the instances that the PPP indexed was negative, the *l2d_cache* event count increased significantly, 920% for the 256 thread *358.botsalgn* run and 46% for the *351.bwaves* 256 thread *351.bwaves* run. Large increases in the *l2d_cache* were also reported by the *352.nab* and *359.botsspar* benchmarks, which resulted in negative PPP rates.

6.5 Conclusion

In this study, a number of benchmarks were analyzed using different performance metrics. Each metric gives the user an idea of how efficiently different components of the system were used. Relative rates make it possible to see if the use of resources improved when the number of threads was increased.

We proposed the use of normalized performance metrics using the *purchasing power parity* theory to quantify the relative difference in performance and cache/memory metrics. PPP based indexes have been used to track the purchasing power of many currencies by comparing specific goods. The widely known Big Mac Index is an example of a PPP metric where local prices of Big Macs from different countries are compared against US priced Big Macs. The aim is to use the US dollar as the baseline for currency comparisons. The Big Mac Index makes it possible to determine if currencies are undervalued or overvalued when compared to the purchasing power of the US dollar.

We take a similar approach by determining if performance counter value changes have a relatively greater or lesser effect on cache and CPU metrics when comparing different thread counts using different counters as the common currency. Our method used well known metrics as the *good* that was to be compared across runs. As the number of threads was increased, the potential of generating cache and CPU performance ratios of varying magnitude for the same program also increased. Each of these variations could potentially require a different number of cycles, memory accesses, cache accesses, retired instructions, among many others, to run the program to completion. Our goal was to use 64 thread run metrics as a baseline to normalize other metrics generated by runs with a higher thread count. This normalization made it possible to quantify relative increases or decreases, in the magnitude of metrics, when compared to the baseline run.

Our goal was to provide a simple technique to compare complex systems, and we found PPP normalization was a suitable technique that can accomplish this task. Our approach makes it possible for computer architects and compiler developers to track the drift of bottlenecks by easily identifying trends, and magnify differences that otherwise would go unnoticed. Quantification of the metric drift makes it possible to assess the effects changes to programs or compilers have in the effective use of CPU architectural features, and cache hierarchies in particular.

Future work will focus on statistical modeling techniques such as linear regression to quantify the significance of a metric to overall performance. Small changes in some metrics might not be as significant as changes in other cases. For instance, writeback buffers were identified as potential sources for denial-of-service attacks [63]. When writeback buffers become full in non-blocking caches, it has the effect of blocking the entire cache until buffer space becomes available. To determine the threshold at which writeback buffers become a problem will entail a more in-depth analysis. Statistical modeling takes multiple runs to generate sufficient data points. PPP rates can assist in identifying metrics of increasing concern, making it possible for the user to focus on a smaller set of metrics to analyze. A smaller parameter study will potentially result in a faster and more efficient analysis process.

Additional work will characterize the AArch64 platform through studies with different configurations such as hyperthreading disabled and different thread affinity settings. Our results, Figure 6.1, show that some of the benchmarks experience little to no performance gains as the number of threads was increased. These findings confirm previously reported results on a similar system but with an earlier version of the same compiler [58].

Chapter 7

Revisiting Effects of Spectre-Meltdown Security Patches

There is always a possibility that the application of security patches might have a detrimental effect on performance. The Spectre-Meltdown security flaws raised concerns that its subsequent fixes might have detrimental effects on performance by curtailing a CPU's abilities to prefetch data. Numerous publications showed that the effects introduced by the security fixes, while not negligible were small in most circumstances and manageable in others. In this chapter, we showed the relative effects the application of two of the Spectre-Meltdown fixes had on performance in terms of Top-Down bottleneck ratios. We showed that while the overall performance was not affected for most benchmarks, the Top-Down profiles were relatively affected [16].

7.1 Introduction

Operating systems are complex computer programs that are continuously evolving to accommodate changes and updates to the underlying hardware it runs on. Like any other piece of software, frequent updates are released to address security issues, improve usability, enhance performance, and fix software bugs. These fixes have the potential of

affecting performance, and it is essential to gain an understanding on the effect software patches have on a system. It is through the use of well known performance metrics that a proper assessment of security patches can be made by quantifying their effect, not only on overall performance, but on the different subsystems that make up a CPU.

In January 2008, two major vulnerabilities were reported, Spectre and Meltdown [64],[65]. These vulnerabilities made it possible for attackers to gain access to data, stored in memory or caches, by bypassing security mechanisms. The exploits took advantage of CPU features that make it possible to use speculative execution to increase CPU performance. It was fear that the security fixes would have a major detrimental effect on performance by possible curtailing the speculation capabilities of CPUs.

A number of studies on the effects which the Spectre and Meltdown security patches had on performance were published. In one study, a number of Cray supercomputers were used to analyze the effects patches had on runtime performance. A number of benchmarks were tested, and it was found that the overall impact of the security patches was minimal [66]. Another study, showed the effects different patches had on two computational intensive workflows, pMatlab and Keras with TensorFlow, on an Intel based cluster [67]. It reported that significant negative effects, up to 21% for pMatlab and 16% for TensorFlow, once the CPU microcode update was applied.

To quantify the effect a change on the configuration or code had on performance, performance metrics such as the ones derived from the Top-Down bottleneck analysis are used [1]. This approach, the comparison of metrics after a change, is called differential analysis, and it makes it possible to associate specific changes on the system or code with changes on performance metrics [20]. A problem can arise when absolute rates are compared. The issue is that the comparison might provide an incomplete picture of changes between rates. Relative changes, normalized with the purchasing power parity technique [22], can provide additional information on the metric drift. This technique has been used to account for differences across GCC compiler suite releases [17] using the Top-Down bottleneck classification method. PPP made it possible to identify significant relative variations in different Top-Down categories. The Top-Down classification, in conjunction with PPP normalization, has been similarly applied to the AArch64 architecture [59], where it was used to analyze strong scaling and its resulting bottlenecks.

In this study, a comprehensive Top-Down and PPP analyses were made to quantify absolute and relative bottleneck metric changes, bottleneck drift, of a system when the Meltdown and Spectre security patches were enabled. Our study makes the following contributions:

- We found little difference in all but one of the benchmarks between patch settings.
- We showed that bottleneck profiles can differ even when security patches had little effect on performance.
- We showed that relative rates can vary significantly, while absolute bottleneck can remain relatively similar.
- We highlighted trends and differences in metrics that might have otherwise gone unnoticed by standard evaluation practices.

Performance analysis and system characterizations is a time consuming and complex process. Checking the impact of security patches requires multiple testing of different programs. Our approach makes it possible to compare bottleneck metrics by quantifying their absolute and relative changes when patches are applied. This makes it possible to obtain a more complete picture of the effect security patches had on systems.

7.2 Background

Table 7.1: Top-Down Metric formulas for Intel Skylake processor [1], [7].

Metric	**Formula**
CORE_CLKS	CPU_CLK_UNHALTED.THREAD_ANY / 2
CLKS	CPU_CLK_UNHALTED.THREAD
SLOTS	4 * CORE_CLKS
Recovery_Cycles	INT_MISC.RECOVERY_CYCLES_ANY / 2
L2_Bound_Ratio	(CYCLE_ACTIVITY.STALLS_L1D_MISS -CYCLE_ACTIVITY.STALLS_L2_MISS)/ CLKS

Continued on next page

Table 7.1 – *Continued from previous page*

Metric	**Formula**
LOAD_L2_HIT	MEM_LOAD_RETIRED.L2_HIT* (1+MEM_LOAD_RETIRED.FB_HIT/ MEM_LOAD_RETIRED.L1_MISS)
Frontend Bound	
Frontend Bound	IDQ_UOPS_NOT_DELIVERED.CORE / SLOTS
DSB	(IDQ.ALL_DSB_CYCLES_ANY_UOPS - IDQ.ALL_DSB_CYCLES_4_UOPS) / CORE_CLKS
Branch Resteers	(INT_MISC.CLEAR_RESTEER_CYCLES+ BACLEARS.ANY) / CLKS
Bad Speculation	
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + (4*Recovery_Cycles))/SLOTS
Branch Mispredicts	(BR_MISP_RETIRED.ALL_BRANCHES/ (BR_MISP_RETIRED.ALL_BRANCHES + MACHINE_CLEARS.COUNT)) * Bad Speculation
Machine Clears	Bad Speculation - Branch Mispredicts
Retiring	
Retiring	UOPS_RETIRED.RETIRE_SLOTS / SLOTS
Microcode Sequencer	((UOPS_RETIRED.RETIRE_SLOTS / UOPS_ISSUED.ANY) IDQ.MS_UOPS) /SLOTS
Base	Retiring - Microcode Sequencer
Backend Bound	
Backend Bound	1 - (Frontend Bound + Bad Speculation + Retiring)
Backend Bound: Memory Bound	
Store Bound	EXE_ACTIVITY.BOUND_ON_STORES / CLKS

Continued on next page

Table 7.1 – *Continued from previous page*

Metric	**Formula**
L1 Bound	(CYCLE_ACTIVITY.STALLS_MEM_ANY -CYCLE_ACTIVITY.STALLS_L1D_MISS)/CLKS
L2 Bound	(LOAD_L2_HIT / (LOAD_L2_HIT + L1D_PEND_MISS.FB_FULL)) * L2_Bound_Ratio
L3 Bound	(CYCLE_ACTIVITY.STALLS_L2_MISS -CYCLE_ACTIVITY.STALLS_L3_MISS)/ CLKS
DRAM Bound	(CYCLE_ACTIVITY.STALLS_L3_MISS / CLKS) + L2_Bound_Ratio - L2_Bound
Backend Bound: Core Bound	
Divider	ARITH.DIVIDER_ACTIVE / CLKS
UPC	UOPS_RETIRED.RETIRE_SLOTS / CLKS
Few_Uops_Executed _Threshold	EXE_ACTIVITY.2_PORTS_UTIL * UPC / 5
Core_Bound_Cycles	EXE_ACTIVITY.EXE_BOUND_0_PORTS + EXE_ACTIVITY.1_PORTS_UTIL + Few_Uops_Executed_Threshold
Ports Utilization	**if** ARITH.DIVIDER_ACTIVE < EXE_ACTIVITY.EXE_BOUND_0_PORTS **then** Ports Utilization = Core_Bound_Cycles / CLKS **else** Ports Utilization = (Core_Bound_Cycles - EXE_ACTIVITY.EXE_BOUND_0_PORTS) / CLKS

7.2.1 Spectre and Meltdown Vulnerabilities

The Meltdown vulnerability allows an attacker to gain read access to all memory, even when lacking the appropriate privileges to do so [65]. The Spectre exploit allows attackers to gain access to private information through branch mispredictions [64]. For this study, we focused on the effect the patches had on pipeline bottlenecks. The following variant security patches were provided the OS vendor and applied to the system: [68], [69]:

- *Spectre, variant 1:* This is a kernel patch fix that is always enabled. It provides bounds checking during branching to prevent arbitrary bypassing.
- *Spectre, variant 2:* This fix includes microcode and kernel patches. It can be disabled to prevent performance impacts. It prevents data leakage through indirect branch poisoning.
- *Meltdown, variant 3:* This is a kernel fix. It can be disabled to prevent performance impacts. It prevents an attacker from reading memory through speculative cache loading.

In the following subsections, we discuss the methods used to analyze the performance impact of the security patches for Spectre, variant 2, and Meltdown, variant 3, had on the system.

7.2.2 Top-Down Classification Method

The Top-Down analysis method is a bottleneck classification technique that identifies dominant bottlenecks of an application. This method tracks CPU pipeline slots - resources needed to process a micro-operation (uop). Uops are low level hardware operations of microarchitectural instructions which were generated to represent the application being executed by the CPU. Pipeline slots are assigned into four main categories: Frontend Bound, Backend Bound, Retiring and Bad Speculation [1]. A simple classification is applied to pipeline slots to assign the bottleneck to the right category. If a slot was allocated, it will be classified as Retiring if the slot is eventually retired. It will be assigned to the Bad Speculation category if it is not retired. If the slot cannot be allocated, it will be assigned to the Backend Bound category if it is a back end stall. Otherwise, it will be assigned to the Frontend Bound category. Back end stalls occur when there are not enough resources in the back end portion of the pipeline to handle new slots. Front end stalls take place when the front end can not supply slots to the back end portion of the pipeline. Non stalled slots are classified as Bad Speculation, when a slot will never retire due to an incorrect speculation, or slots were blocked by the pipeline due to recovery operations due to an earlier bad speculation. Retired slots are the slots that successfully completed their operations.

To apply the Top-Down analysis technique, a user would first compute the main category metrics to identify which classification has the highest bottleneck rate. Once a category is identified, the user can narrow down the metrics needed to analyze by just focusing in the subcategories of the selected main category. The user can continue generating metrics until the source of the problem is identified. Since our goal is to provide a comprehensive view of the different components that make up the processor, our experimental runs included multiple categories. This made it possible to get a more complete picture of bottlenecks across the processor, and a better understanding of how the different processor components were affected by the use of security patches. Table 7.1 lists the main Top-Down categories and subcategories that were used in this paper, along the corresponding formulas needed to compute the metrics. More in-depth descriptions, definitions and techniques of the Top-Down metrics, and how to use the Top-Down analysis method, were made available by the CPU vendor [51].

7.2.3 Purchasing Power Parity

Purchasing power parity theory underlies different methods to compare the cost of identical products such as lattes, and iPods between different countries, each of them with different currencies [22], [56], [62]. The most famous PPP index is the Big Mac Index (BMI), which was developed by The Economist magazine [25]. The goal of the BMI is to compare the strength of the currency by testing how much of the same product a currency can buy when compared to another currency. A currency is overvalued – when the product bought using that currency is more expensive – or undervalued – when the product is cheaper – when compared to a base currency.

The following is an illustrative example of the purchasing power of the Chinese yuan versus the US dollar as described in The Economist magazine. For this example, the dollar to yuan exchange rate is \$1 = 6.4 yuan. The quoted Big Mac price was \$5 and 20 yuan. Equation 7.1 computes the Big Mac exchange rate which is based on its local price.

$$20/5 = 4 \tag{7.1}$$

Equation 7.1 shows that on the basis of Big Mac burger prices, the exchange rate

should be set at 4 yuans per dollar. Since the actual exchange rate is 6.4 yuans per dollar, Equation 7.2 shows that the yuan is 37.5% undervalued as compared to the US dollar.

$$(4 - 6.4) * 100/6.4 = -37.5 \tag{7.2}$$

PPP theory can be used to determine the relative difference between bottlenecks generated by the same benchmark but generated under different system configurations. The *currency* used to compare the cost is the number of cycles it took to run the program to completion. The *product* being compared are the Top-Down metrics for each benchmark. The goal is to show that a metric value can differ, or be similar to another, as defined by the Top-Down formulas, while its true cost might be relatively higher or lower, when compared to a baseline run. PPP normalized rates close to 0% imply parity between the patches disabled and enabled metrics. It takes about the same number of CPU_clk cycles for a similar number of pipeline slots to achieve similar Top-Down metric rates. For positive PPP rates, it implies that the patches enabled CPU_clk cycles are overvalued. It requires less cycles to achieve same metric magnitude when compared to a configuration with the security patches disabled. Negative PPP rates imply that the CPU_clk cycles for a configuration with patches enabled are undervalued. It requires more cycles to achieve the same metric value when compared to a configuration with security patches disabled.

Top-Down metrics were computed using the formulas described in Table 7.1. The use of *CPU_CLK_UNHALTED. THREAD*, or *CPU_CLK_UNHALTED.THREAD_ANY* to compute the PPP Exchange Rate, Equation 7.3, was based on which PMU event the Top-Down metric formula used in its computation. Some metrics use *CLKS* while others use the *CORE_CLKS* performance metric. The baseline CPU_clk values used for comparison were obtained from runs with the security patches disabled.

$$PPP\_Exchange\_Rate = CPU\_clk/CPU\_clk_{baseline} \tag{7.3}$$

Equation 7.4 computes the PPP index. The baseline, represented by the variable $Metric_{baseline}$, was the resulting Top-Down metric value with the security patches disabled.

$$PPP = 100 * ((Metric/Metric_{baseline}) \\ -PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \tag{7.4}$$

The following is an example of how to compute the drift in terms of relative difference of the Retiring metric for the *370.mgrid331* benchmark. The Retiring metric was computed using the formulas provided by the Top-Down method as shown in Table 7.1, and the results were found to be 0.06789046 when patches were enabled, and 0.09773369 when patches were disabled. The PPP exchange rate was computed using the PMU event, *CPU_CLK_UNHALTED. THREAD_ANY* with a resulting value of 1.46, Equation 7.5. The drift between security patch settings was found to be -52.42%, Equation 7.6. When patches were enabled, the *CPU_CLK_UNHALTED.THREAD_ANY* were overvalued. The system used more *CPU_CLK_UNHALTED.THREAD_ANY* cycles to retire a similar number of uops when patches were disabled.

$$81671652350125/55924600475776 = 1.46 \tag{7.5}$$

$$100 * ((0.06789046/0.09773369) - 1.46)/1.46 = -52.42 \tag{7.6}$$

In this paper, we use the relative change between metrics, the difference in Top-Down metrics between patch settings divided by the metric value when patches were disabled, as an additional indicator of the changes between patch settings. For the example just described, *370.mgrid331* had a relative change in its Retiring metric of -30.54%. The relative change and PPP normalized rates give us a sense of the relative change of the Top-Down metric between patch settings, not the change of its effect on the system. Additionally, PPP normalized rates give us information on the relative change when taking into account the number of core cycles that were used to compute the metrics, which is useful when putting large percentage values in relative changes in perspective.

Table 7.2: SPEC OMP2012 Benchmark Description [2]

Benchmark Name	Programming Language	Description
350.md	Fortran	Physics: Molecular Dynamics
351.bwaves	Fortran	Physics: Computational FluidDynamics (CFD)
352.nab	C	Molecular Modeling
357.bt331	Fortran	Physics: Computational Fluid Dynamics (CFD)
358.botsalgn	C	Protein Alignment
359.botsspar	C	Sparse LU
360.ilbdc	Fortran	Lattic Boltzmann
362.fma3d	Fortran	Mechanical Response Simulation
363.swim	Fortran	Weather Prediction
367.imagick	C	Image Processing
370.mgrid331	Fortran	Physics: Computational Fluid Dynamics (CFD)
371.applu331	Fortran	Physics: Computational Fluid Dynamics (CFD)
372.smithwa	C	Optimal Pattern Matching
376.kdtree	C++	Sorting and Searching

7.3 Experimental setup

We used the Top-Down method in combination with the SPEC OMP2012 benchmarks [57], [2] to measure the effects of Spectre and Meltdown patches have on the Intel 2021.1 compiler suite. The following compiler options were used as the default to compile most benchmarks: *-fopenmp -O3 -march=skylake-avx512 -g -pg*. Some of the benchmarks required additional or different options. The *371.applu331* benchmark used *-fopenmp -O2 -march=skylake-avx512 -g -pg*. *367.imagick* required the compiler option *-std=c99* to be added to the default options. Additionally, the option *-FR* was added to *350.md*, while *-mcmodel=medium* needed to be added to the *363.swim* and *357.bt331* benchmarks. The SPEC OMP2012 benchmarks are described in Table 7.2. OMP2012 results that followed the SPEC reporting guidelines can be submitted for publication [58]. A two

socket Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 8 cores per socket, 2 threads per core was used running the CentOS 7.6.1810 Linux version installed. *perf record* collected the data from eight performance counters per experimental run. There were at least five data points per performance counter for both patch settings. To compute Top-Down metric rates, the average of the performance counter values was used.

It is possible to disable the Spectre variant 2 and Meltdown variant 3 through an interface made available by the Red Hat Linux vendor, which is also available to the CentOS distribution. The vendor also made available a script to check the state of the security patches, to see whether or not the system currently has its patches enabled or disabled [69]. In this study, version 3.1 of the verification script was used. To disable the security patches, a 0 was stored in the following files located in */sys/kernel/debug/x86/*: *ibrs_enabled*, *retp_enabled* and *pti_enabled*. Patches were enabled by replacing the 0 with a 1 in the same files.

7.4 Analysis of Results

Figure 7.1 shows the speedup gains or losses when the security patches were enabled. There were at least 55 runs for each benchmark for both patch settings, and the averages were taken to compute the speedups. The plot shows that most benchmarks suffer about a $0.01x$ speedup loss. The exception is *360.ilbdc*, which experienced a small gain of $0.02x$, and *370.mgrid331*, which had a negative effect close to $0.04x$. While these are not significant effects on runtime, we further analyzed the effects of the security patches through the use of the Top-Down classification method to see how bottlenecks were affected on a subset of benchmarks. We showed that while benchmark runtimes were similar, their bottleneck profiles were different. With the use of PPP techniques, we were able to highlight and quantify these relative differences when compared to the baseline, a system with its security patches disabled.

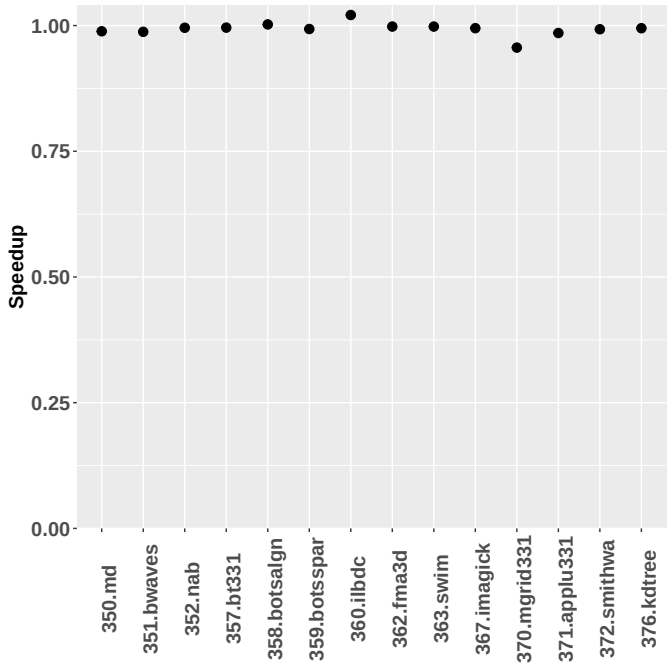


Figure 7.1: Speedup comparison for the Intel 2021.1 compiler suite when patches are enabled using SPEC OMP2012 benchmarks with 32 threads and SMT enabled. Higher is better.

Table 7.3: Performance Counters of Significance

Performance Counter	Benchmark	Category
BACLEARS.ANY	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Branch Resteers
BR_MISP_RETIRED. ALL_BRANCHES	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Branch Mispredicts
CPU_CLK_UNHALTED .THREAD	360.ilbdc	CLKS
CPU_CLK_UNHALTED .THREAD_ANY	359.botsspar, 360.ilbdc, 359.botsspar	CORE_CLKS
CYCLE_ACTIVITY.STALLS_ L1D_MISS	359.botsspar	L1, L2 Bound
CYCLE_ACTIVITY.STALLS_ L2_MISS	359.botsspar	L2, L3 Bound
CYCLE_ACTIVITY.STALLS_ L3_MISS	359.botsspar, 370.mgrid331	L3, DRAM Bound
CYCLE_ACTIVITY.STALLS_ MEM_ANY	358.botsalgn	L1 Bound
EXE_ACTIVITY.2_PORTS_ UTIL Threshold	359.botsspar	Few_Uops_Executed_
EXE_ACTIVITY. EXE_BOUND_0_PORTS	358.botsalgn, 359.botsspar, 360.ilbdc	Ports Utilization
IDQ.ALL_DSB_CYCLES_ 4_UOPS	358.botsalgn, 359.botsspar	DSB
IDQ.ALL_DSB_CYCLES_ ANY_UOPS	358.botsalgn, 359.botsspar	DSB

Continued on next page

Table 7.3 – *Continued from previous page*

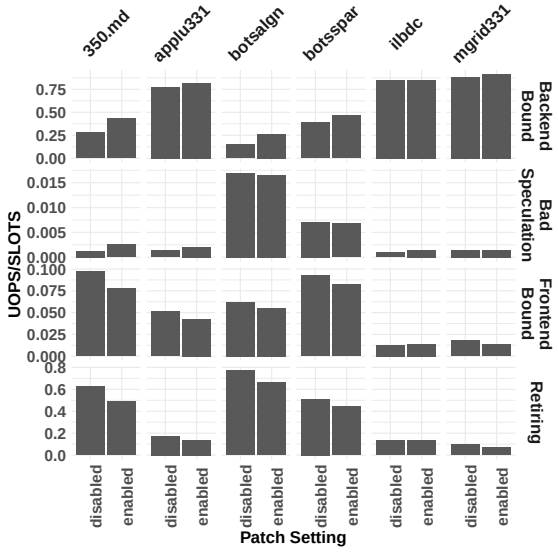
Performance Counter	**Benchmark**	**Category**
IDQ.MS_UOPS	358.botsalgn, 359.botsspar, 370.mgrid331	Microcode Sequencer
IDQ_UOPS_NOT_ DELIVERED.CORE	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331	Frontend
INT_MISC.CLEAR_ RESTEER_CYCLES	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331	Branch Resteers
INT_MISC.RECOVERY_ CYCLES_ANY	358.botsalgn, 359.botsspar, 360.ilbdc,370.mgrid331, 371.applu331	Recovery_Cycles
MACHINE_CLEARS.COUNT	358.botsalgn, 359.botsspar, 360.ilbdc,370.mgrid331, 371.applu331	Branch Mispredicts
MEM_LOAD_RETIRED. FB_HIT	358.botsalgn, 359.botsspar	L2 Bound
MEM_LOAD_RETIRED. L1_MISS	358.botsalgn	L2 Bound
UOPS_ISSUED.ANY	358.botsalgn, 359.botsspar	Bad Speculation, Microcode Sequencer
UOPS_RETIRED.RETIRE_ SLOTS	358.botsalgn, 359.botsspar	Bad Speculation, Retiring, Microcode Sequencer

Table 7.3 shows the performance counters of significance that were used to compute the Top-Down metrics. When the results followed a normal distribution, the unpaired two-sample t-test was used. When the results did not follow a normal distribution, the non parametric two-samples Wilcoxon rank test was used. We had a minimum

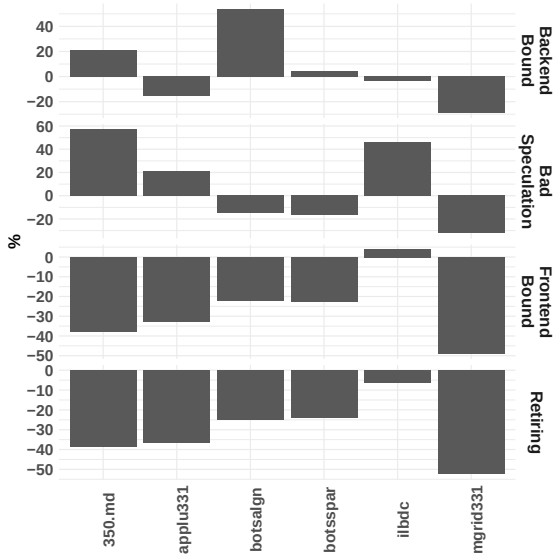
of five runs per counter for both settings, patches enabled and disabled. P-values for the performance counters were identified as significant at values less than 0.05. For the Retiring category, there were more uops delivered by the Microcode Sequencer for *358.botsalgn*, *359.botsspar*, and *370.mgrid331* when patches were enabled. Additionally, *359.botsspar* had increases in number of uops issued by the resource allocation table while at the same time the number the number of retiring slots increased when patches were enabled. *358.botsalgn* had the opposite effect.

For the Frontend Bound metric, *358.botsalgn*, *359.botsspar*, *360.ilbdc* and *370.mgrid 331* had increases in the number of uops not delivered to the resource allocation table per thread when the patches were enabled. A higher number in none delivered uops could potentially translate in the frontend under-supplying the CPU's backend portion of the pipeline. Similarly, the *358.botsalgn*, *359.botsspar*, *360.ilbdc*, *370.mgrid331* and *371.applul331* reported an increase in the number of times frontend resources are resteered when encountering branch instructions in a fetch line with patches enabled. *358.botsalgn* and *359.botsspar* had decreases in the number of uops and 4 uops cycles that were delivered to the instruction decode queue unit. These decreases occurred when patches were enabled.

In the Bad Speculation category, the following benchmarks had increases of statistical significance when patches were enabled. *358.botsalgn*, *359.botsspar*, *360.ilbdc*, *370. mgrid331* and *371.applu331* had increases in the number of events that require the clearing of the pipeline, the number of mispredicted retired instructions, and the number of stalls due to recoveries from earlier clear events increased for these benchmarks. In the core bound category, *358.botsalgn*, *359.botsspar* and *360.ilbdc* had statistically significant increases of cycles with where no uops were executed on all ports. *359.botsspar* had an increase in the number cycles in which 2 uops were executed on all ports. In the memory bound classification, the number of execution stalls due data misses increased for *359.botsspar* for L1D, *359.botsspar* for L2, *359.botsspar* and *370.mgrid331* for L3. Additionally, *359.botsspar* reported statistically significant increases for execution stalls due to memory subsystem outstanding loads.



(a) Regular rates



(b) PPP normalized Rates

Figure 7.2: Results of the Top-Down architectural bottleneck classification main categories.

7.4.1 Top-Down Metrics

Figure 7.2(a) shows the effects the security patches had on the main Top-Down categories. With the exception of *360.ilbdc*, all Frontend Bound values dropped. This was driven by two factors when patches were enabled: the number of *CPU_CLK_ UNHALTED.THREAD_ANY* increased for all benchmarks, and the number of uops not delivered to the resource allocation table when the backend portion of the pipeline was not stalled, the *IDQ_UOPS_NOT_DELIVERED.CORE* performance counter, stayed relatively the same. In the case of *360.ilbdc*, the opposite was true, the number of CPU clock cycles stayed relatively the same while the number of uops not delivered increased by more than 11%. This resulted in an increase of 7.5% in the Frontend Bound metric when the security patches were enabled. Figure 7.2(b) shows the corresponding PPP normalized rates. Except for *360.ilbdc*, which had a PPP rate of 3.67%, all benchmarks had negative PPP rates that ranged from −22% for *358.botsalgn* and *359.botsspar*, to −49% for *370.mbrid*. As the number of cycles, CPU_CLK_UNHALTED.THREAD_ANY, increased, the number of uops not delivered increased modestly. Resulting in less not delivered uops per cycles, making the cycles undervalued. The runs with patches enabled handled relatively the same number of stalls with more core cycles.

The Retiring metric values decreased for all benchmarks when the patches were applied. It had a drop of −30.53% for *370.mgrid331*, while *350.md* and *371.applu331* had drops of about −20%. *360.ilbdc* had a drop of −2.73%. This is attributed to the increase in core cycles, CPU_CLK_UNHALTED.THREAD_ANY, while the number of retired slots remained relatively the same when patches were enabled. PPP rates decreased for all benchmarks. Expect for *360.ilbdc*, all had at least a −24% drop, with *370.mgrid* recording a drop of −52%. As the number of core cycles increased with the security patches enabled, the number of retired slots remained relatively the same. *370.mgrid* had the largest increase in core cycles, 46%, while *360.ilbdc* had the smallest increase, 3.73%. This explains the difference in magnitude in PPP rates.

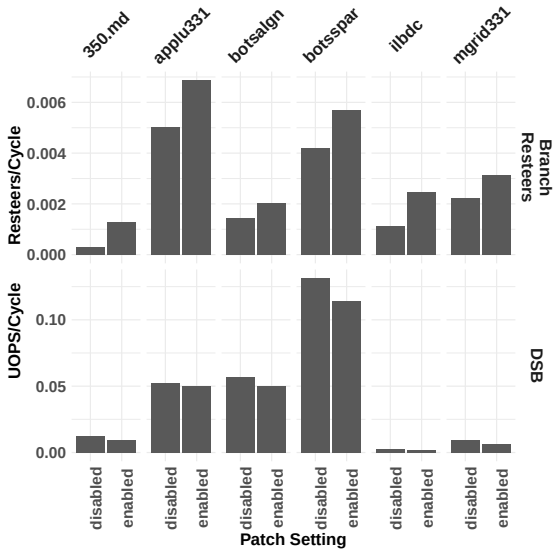
The Bad Speculation metric had an increase of 101% for *350.md*, 50% for *371.applu331*, and 51% for *360.ilbdc* when patches were enabled. This is due to an increase of *UOPS_ISSUE_ANY* and *Recovery_Cycles* while the number of *UOPS_RETIRED. RETIRED_SLOTS* stayed relatively the same. The other benchmarks, *359.botsspar*, *358.botsalgn* and *370. mgrid331*, had less than a −4% decrease in Bad Speculation

rates. Since the number of clock cycles, the denominator in the formula, also increased but at a larger rate, the Bad Speculation rate decreased when patches were enabled. While regular rates showed a decrease of −4%, PPP normalized rates were larger in magnitude, 32.23% for *370.mgrid*, 16.63% for *359.botsspar* and 14.84% for *358.botsalgn*. The effects of large increases in core cycles, 46% for *370.mgrid331*, and 14% for *359.botsspar* and *358.botsalgn*, resulted in decreases of PPP rates. There were more core cycles to do the same amount of work once the patches were enabled, which resulted in a depreciation in the value of core cycles. The other benchmarks experienced gains in PPP normalized rates. *350.md* had a gain of 57.15%, *371.applu331* had a gain of 21.06%, and *360.ilbdc* had a gain of 46.42%. The *Recovery_Cycles* metric increased at a much larger rate for this subset of benchmarks than the benchmarks with negative PPP rates. *350.md* had an increase in *Recovery_Cycles* of 1334%, while *371.applu331* had an increase of 1446% and *360.ilbdc* had an increase of 983%. Positive PPP rates show that core cycles were overvalued, the same amount of work is being done with less core cycles relatively to baseline runs.

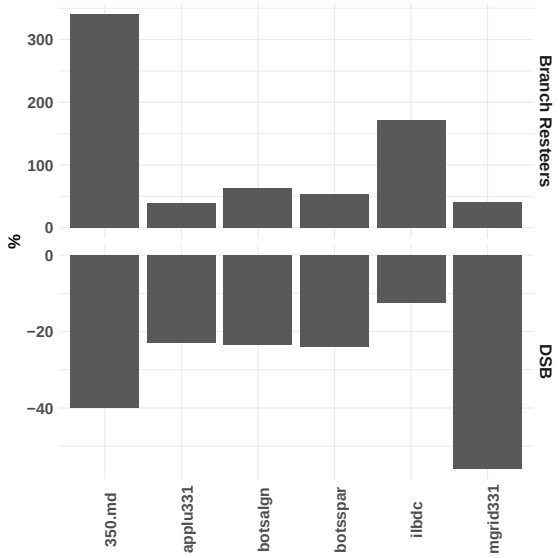
The *360.ilbdc* benchmark saw less than a 1% difference in the Backend Bound metric between patch settings. For *360.ilbdc*, the Bad Speculation, Front End, and Retiring metrics stayed relatively the same, resulting in a very similar Backend Bound rate. All other benchmarks had an increase in the Backend Bound rates because of lower Retiring rates when patches were enabled. *370.mgrid331* had a 4% increase, while *371.applu331* recorded an increase of 6%. Other benchmarks had large increases. *359.botsspar* had an increase of 19%, *350.md* had a 55% increase, and *botsalgn* had a 76% increase. PPP rates increased for benchmarks that had large increases in Backend Bound rates. For instance, *350.md* had a core cycles increase of 28% but its Backend Bound rate had a larger effect when it increased by 54%, resulting in PPP rate of 20%. *370.mgrid331* had a core rate increase of 46% and a Backend Bound rate increase of 3.90%, resulting in a PPP rate of −28%. When patches were enabled, the number of cycles increased for some benchmarks at higher rates than there were uops to be processed resulting in negative PPP rate, while others had proportionally fewer cycles for an increasing number of uops resulting in overvalued cycles and positive PPP rates.

The following subsections describe the effects of the security patches had on different Top-Down subcategories.

Frontend Bound



(a) Regular rates



(b) PPP normalized Rates

Figure 7.3: Frontend Bound subcategories.

Frontend Stalls track the fraction of slots that were affected when the frontend

of the pipeline undersupplies the pipeline's backend. Two subcategories were examined in this paper: DSB and Branch Resteers. The DSB metric tracts the fraction of CPU cycles that were affected by the decoded uop cache, DSB, fetch pipeline. Branch Resteers account for the CPU stalls due to branch resteers, delays from a corrected path, after a mispredicted branch. Figure 7.3(a) shows that the Branch Reesters metric increased when the security patches were enabled. This is due to increases in the number of cycles the issue stage had to wait to recover from bad speculation events, while at the same time the number of CPU_CLK_UNHALTED.THREAD decreased or stayed the same. For example, *350.md* had a Branch Reesters rate increase of 329% due mostly in part to an increase of 234% in INT_MISC.CLEAR_RESTEER_CYCLES, and a slight decrease of −2.58% for CPU_CLK_UNHALTED.THREAD. *360.ilbdc* had a increase of 117.93% for Branch Reesters resulting from an increase of 61.81% in INT_MISC.CLEAR_RESTEER_CYCLES and a decrease of −19.65% in CPU_CLK_UNHALTED.THREAD. Normalized PPP rates for the Branch Resteers metric were all positive. While CPU core cycles increased, Branch Resteers increased at higher rates. The largest percentage increases in PPP rates reflect large increases in Branch Resteers while lower increasing rates for core cycles. That is the case for *350.md*, which had a PPP rate of 340.42%. Its core cycles rate increased by 28.16% while its Branch Resteers rate increased by 329.04%. Similarly, *360.ilbdc* had a PPP rate of 171.25%, because of an increase of 3.73% in core cycles and a 117.94% increase in Branch Resteers when patches were enabled. Core cycles were overvalued, when compared to their baseline, because fewer core cycles had to do relatively less work.

The DSB metric decreased when patches were enabled. While the number of uops that were delivered to the instruction decode queue remained the same or had a slight decrease, there were increases in CPU_CLK_UNHALTED.THREAD_ANY, the divisor. This resulted in more CPU cycles for the same number of delivered uops for all the benchmarks. For instance, *370.mgrid331* and *350.md* had increases of 46% and 28.16% for CPU_CLK_UNHALTED.THREAD_ANY, resulting in DSB decreases of −35.54% and −23.05% respectively while the uops delivered stayed relatively the same. DSB PPP rates were negative for all benchmarks. The number of core cycles increased while DSB rates decreased when patches were enabled. The benchmarks with the highest rates reflect the large increases in core clocks and decreases in the DSB rate. That is

the case for *370.mgrid*. It had a PPP rate of −55.87%, an increase of 46.04% for its CPU_CLK_UNHALTED.THREAD_ANY value and a decrease of −35.55% for its DSB rate. The lowest PPP rate was reported by *360.ilbdc*. It had a small increase in core cycles, 3.73%, and a decrease in DSB rate of −9.29%.

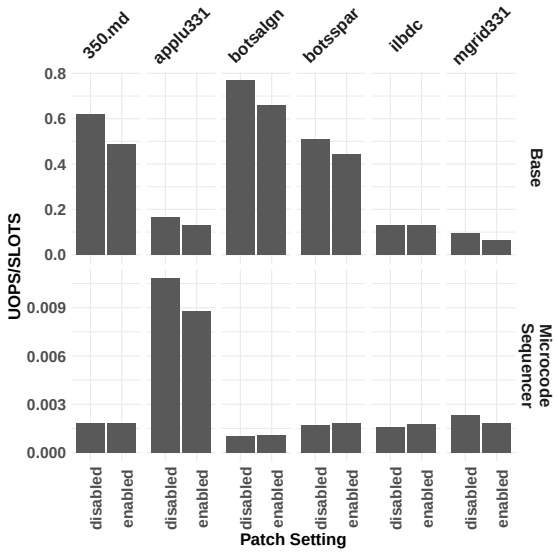
Retiring

The Retiring category represents the fraction of pipeline slots of useful work, the uops that were eventually retired. The Microcode Sequencer metrics is a retiring subcategory that accounts for pipeline slots of uops that were retired and were fetched by the microcode sequencer ROM. The Base metric tracks retired uops that did not originate from the microcode sequencer. Figure 4(a) shows that Base rates across all benchmarks decreased when patches were enabled. This was due to lower Retiring rates. PPP rates for the Base metric rates were negative, reflecting the increasing number of core cycles as the Base rates decreased, Figure 4(b).

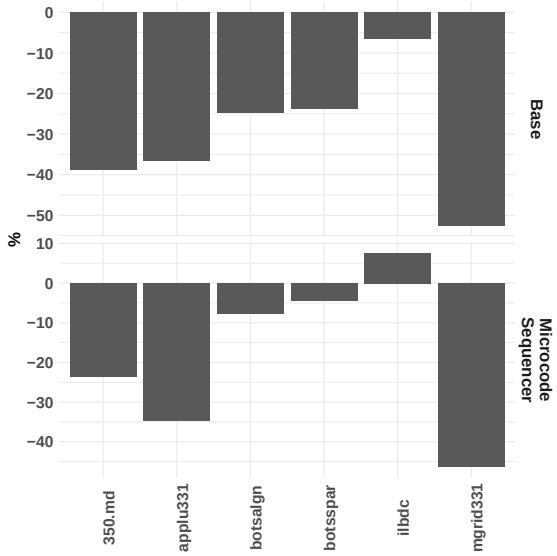
When the security patches were enabled, *370.mgrid331* and *371.applu331* had lower Microcode Sequencer rates, −21.54% and −18.56% respectively. This resulted from an increase in core cycles, CPU_CLK_UNHALTED.THREAD _ANY, the divisor in the metric formula, while the other performance counters remained the same. The other benchmarks had similar or slightly higher Microcode Sequencer rates because the number of uops delivered by the microcode sequencer, IDQ.MS_UOPS, increased at a similar or higher rate than CPU_CLK_UNHALTED.THREAD_ANY. *360.ilbdc* was the only benchmark with a positive PPP rate, 7.51%. This was due to an increase of 11.52% in the Microcode Sequence rate when patches were enabled while the number of core cycles increased modestly, 3.73%. All other benchmarks had negative PPP rates, up to −46.28% for *370.mgrid331* because of the large increase of core cycles and a decrease in the Microcode Sequencer rate when the security patches were enabled.

Backend Bound

The Backend Bound metric measures the fraction of slots where no uops were delivered to the backend portion of the pipeline due to bottlenecks in the computational or memory subsystems. This metric is further divided into Memory and Core Bound subcategories. In this study, the following memory subsystem stalls due to load accesses



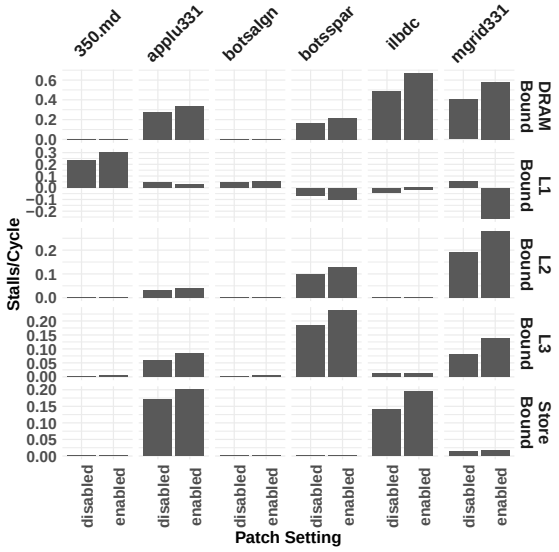
(a) Regular rates



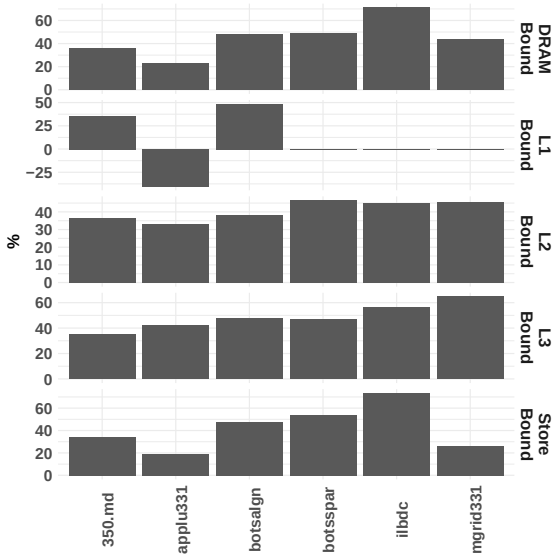
(b) PPP normalized Rates

Figure 7.4: Retiring subcategory.

were tracked through their corresponding Top-Down metrics: L1, L2, L3 and DRAM. Additionally, the Store Bound metric tracks stalls due to store memory accesses.



(a) Regular rates



(b) PPP normalized Rates

Figure 7.5: Memory Bound subcategories which are part of the Backend Bound classification.

The numerator of the L1 Bound metric is the difference between the number of execution stalls due to outstanding loads in the memory subsystem, CYCLE_ACTIVITY.

STALLS_MEM_ANY, minus the number of stalls due to outstanding L1 cache miss demand load, CYCLE_ACTIVITY.STALLS_L1D_MISS. When the security patches were applied, both type of stalls increased, Figure 7.5(a). Some of the benchmarks had negative values because the CYCLE_ACTIVITY.STALLS_L1D_MISS values were larger in magnitude. This was the case for *359.botsspar*, *360.ilbdc* and *370.mgrid331*, which had a negative L1 Bound rate only when patches were enabled. *350.md*, *358.botsalgn* and *371.applu331* had all positive L1 Bound rates. *350.md* had an increase of 31.39% and *358.botsalgn* an increase of 30.67% with patches enabled due to increases in stalls and a decrease in core cycles, CPU_CLK_UNHALTED.THREAD. *371.applu331* had a decrease in the L1 Bound rate of −40.83%. It had a small decrease in the core cycles, and a higher increase in stalls due to L1 cache miss activity, CYCLE_ACTIVITY.STALLS_L1D_MISS, than stalls due to the memory subsystem, CYCLE_ACTIVITY.STALLS_MEM_ANY.

L2 Bound rates were higher when the security patches were enabled. This was attributed to large increases in the L2_Bound_Ratio rates, higher execution stalls for L1 cache misses, CYCLE_ACTIVITY.STALLS_L1D_MISS, and a decrease in CPU_CLK_ UNHALTED.THREAD. *370.mgrid 331* had an increase of 32.11% while *359.botsspar* and *371.applu331* had increases in the low 20s. L3 Bound rates increased with patches enabled. This was due mostly by increases in execution stalls for L2 cache misses, CYCLE_ACTIVITY.STALLS_L2_MISS, and a decrease in core cycles, CPU_CLK_ UNHALTED.THREAD. *370.mgrid331* had an increase of 66.46%, while *371.applu331* had an increase of 40% and *359.botsspar* an increase of 29.27%.

DRAM Bound rates increased when patches were enabled due mainly to increases in stalls while L3 cache miss load demands were waiting, CYCLE_ACTIVITY.STALLS_L3_ MISS, and decreases in core cycles, CPU_CLK_UNHALTED.THREAD. The L2_Bound_ Ratio also increased but had a smaller effect on the DRAM Bound results. *370.mgrid331* had a DRAM Bound rate increase of 45.08%, while others had increases of 37.41% *360.ilbc*, 31.32% for *359.botsspar*, and 21.42% for *371.applu331*. Store Bound rates also increased when patches were applied. This was the result of increases in the number of cycles when the store buffer was full, EXE_ACTIVITY.BOUND_ON_STORES, and a decrease in the number of core cycles, CPU_CLK_UNHALTED.THREAD. Two benchmarks, *360.ilbdc* and *371.applu331*, recorded gains of 39.25% and 17.06% respectively.

PPP rates for DRAM, L2, L3 and Store Bound metrics were consistently positive across all benchmarks, Figure 7.5(b). This was the result of increasing stall rates across these metrics while the number of cycles decreased. There were fewer cycles to handle the increasing number of stalls, so the cycles became overvalued when the security patches were enabled. PPP rates also showed that while not all the benchmarks had significant stall rates in some of the categories, the impact of the patches was significant across all of them. That is the case of the Store Bound metric. For this category, *371.applu331*, and *360.ilbdc* had the largest Store Bound rates of at least 0.14, but the effects the patches had on all benchmarks were found to be of at least 19%, which was the case for *371.applu331* and as much as 53.46% for *359.botsspar*. Large relative changes, as reported by PPP rates, of a metric that is small in magnitude will not have a big effect on the overall Top-Down classification results, it does gives us information on the relative effect the security patches are having on the metric.

The L1 Bound PPP rates for *359.botsspar*, *360.ilbdc* and *370.mgrid331* were not computed because they provided no useful information since the regular rates were negative. The regular rates were negative because of the large increases in the execution stalls due to L1 cache misses, CYCLE_ACTIVITY.STALLS_L1D_MISS, when the security patches were enabled. Not all benchmarks reported negative L1 Bound rates. *350.md* and *358.botsalgn* followed the premise previously stated that an increasing number of stalls in combination with a decreasing number of core cycles resulted in positive PPP rates. The PPP rates were 34.87% for *350.md* and 48.35% for *358.botsalgn*. *371.applu331* had a negative PPP rate of −39.96% because its drop in the L1 Bound rate when the patches were applied.

Core Bound is the second set of subcategories of the Backend Bound classification. They represent all non-memory related bottlenecks. The Divider metric tracks the fraction of cycles in which divide and square root operations used the DIV unit. When patches were enabled, the number of cycles when the divide unit was busy, ARITH.DIVIDER_ACTIVE, increased, while the number of CPU_CLK_UNHALTED.THREAD remained the same or decreased. *360.ilbdc* had an increase of cycles that required root or division operations of 47.67% while the number of core cycles decreased by −19.65%. For benchmarks that had smaller increases in the Divider metric, the increase of cycles used in division and root operations was smaller, while the

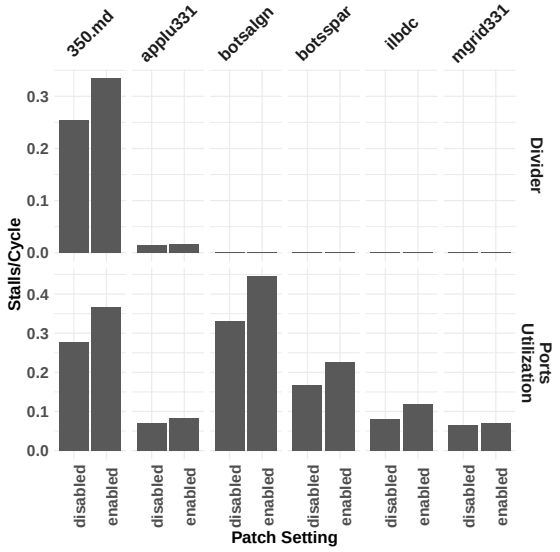
number of core cycles remained relatively the same. That is the case for *370.mgrid*, where CPU_CLK_UNHALTED.THREAD had an increase of 1.15% and an increase in ARITH.DIVIDER_ACTIVE of 13.63%.

The Ports Utilization metric tracks the fraction of CPU cycles affected by limitations in computational resources that don't involve the DIV unit. For benchmarks *350.md* and *371.applu331*, the performance counter ARITH.DIVIDER_ACTIVE was smaller in magnitude than EXE_ACTIVITY.EXE_BOUND_0_PORTS, as a result, the Ports Utilization metric depended only on the Core Bound metric and CPU_CLK_UNHALTED.THREAD. Both of these benchmarks reported increases in the Ports Utilization metric when patches were enabled. This was attributed to increases in the Core Bound rates while the number of core cycles decreased slightly. The other four benchmarks had higher EXE_ACTIVITY.EXE_BOUND_0_PORTS rates, so the formula used to compute Ports Utilization had to include the EXE_ACTIVITY.EXE_BOUND_0_PORTS performance counter in the computation of Ports Utilization. *358.botsalgn*, *359.botsspar*, and *360.ilbdc* had increases of 34.64%, 33.75% and 44.63% respectively when patches were enabled. This was the result of increases in Core Bound rates, and a decrease in core cycles. *370.mgrid* experienced only a 7.31% increase rate because there was a small increase in core cycles and a smaller increase in its Core Bound rate.

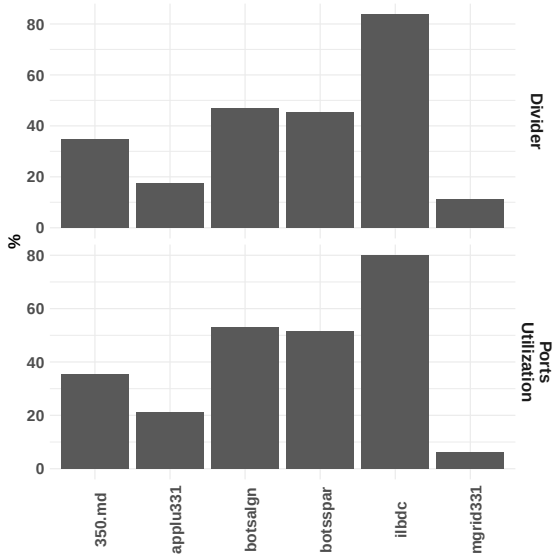
Ports Utilization and Divider PPP normalized rates followed the same patterns. The rates were all positive, due to a decreasing number of CPU_CLK_UNHALTED.THREAD, while the Ports Utilization and Divider regular rates increased as the patches were enabled. The highest PPP rates occurred when the Ports utilization had the largest increase while the core cycles decreased the most. This is the case for *360.ilbdc*. It had a PPP rate of 80.01%, because of an increase in the Ports Utilization rate of 44.63% and a drop in the core cycle count of −19.65%. The same benchmark had the highest Divider rate, 83.79% which resulted from a Divider rate increase of 47.67%.

Bad Speculation

The Bad Speculation metric is used to account for the slots that were wasted due to incorrect speculation. These uops will never get retired. In this study, we analyzed one additional subcategory, Branch Mispredicts, which had relevance due to its rates. The Branch Mispredicts metric tracks slots that were affected by wasted uops that were



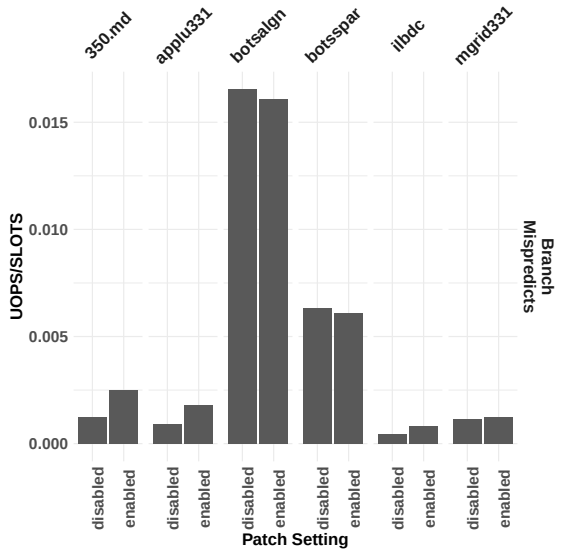
(a) Regular rates



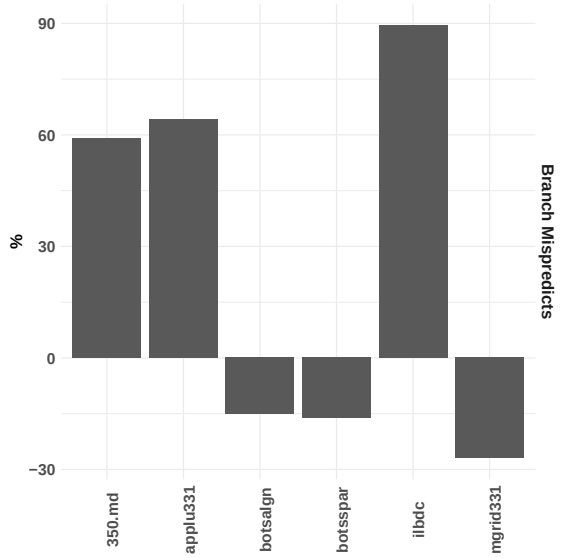
(b) PPP normalized Rates

Figure 7.6: Core Bound subcategories, which are part of the Backend Bound classification.

fetched from an incorrectly speculated path, or stalls that occur when the out-of-order portion of the machine needs to recover its state from a speculative path. With patches enabled, *350.md* had an increase in the Branch Misprediction rate of 103.74% due to an



(a) Regular rates



(b) PPP normalized Rates

Figure 7.7: Bad Speculation subcategories.

increase in the Bad Speculation metric. In this case, the branch misprediction machine clear fraction, BR_MISP_RETIRED.ALL_BRANCHES divided by difference between BR_MISP_RETIRED.ALL_BRANCHES and Machine_CLEARS.COUNT, rate stayed

relatively the same while the number of bad speculation events increased. *360.ilbdc* and *371.applu331* had increases of 96.57% and 104.25% in their Branch Misprediction rates respectively, due to increases in the Bad Speculation rate and the misprediction machine clears fraction. These increases resulted from both, an increase of bad speculation events and the branch mispredictions machine clears fraction. *358.botsalgn* and *359.botsspar* had small Branch Misprediction decreases, less than -4% due to a decrease in the Bad Speculation rate, while the misprediction machine clears fraction remained the same. For these two benchmarks, the effect of the patches was a decrease in the number of bad speculation events resulting in a lower Branch Mispredicts rate. *370.mgrid* had a 6.82% increase due to an increase in the misprediction machine clears ratio.

PPP rates were affected by variations in the Branch Misprediction rates, since all benchmarks had increased in CPU core cycles. When patches were enabled, *350.md* had a PPP rate of 58.98%, *371.applu331* had a rate of 64.15%, and *360.ilbdc* had a rate of 89.50%. More work for relatively less number of core cycles resulted in the core cycles being overvalued when the security patches were enabled. The opposite is true for *358.boltsalgn* that a PPP rate of -14.92%, *359.botsspar* which had a rate of -16.15%, and *370.mgrid331* that had a PPP rate of -26.86%. For these benchmarks, the Branch Misprediction rates either dropped or they stayed relatively at the same levels, while the number of CPU core cycles increased. This resulted in less work for an increasing number of cycles making the core cycles undervalued.

7.5 Conclusion

In this study, we analyzed the effects that the Spectre and Meltdown security patches had on CPU pipeline bottlenecks. Previous studies reported the effects patches had on performance, by focusing on two computationally intensive workflows [67] on an Intel based cluster, and on a diverse set of multiple benchmarks on different Cray based clusters [66]. The first study ran different tests under different conditions: before patches were applied, and with patches applied one at a time. This strategy was very comprehensive because some of the security patches, the BIOS and microcode fixes, could not be disabled once they were applied. The authors found that there was a negative effect when patches were applied and even when they disabled some of the patches via the

vendor provided tunnable feature, the performance degradation on their workflows was significant. The microcode and BIOS fixes had a major impact on performance. The second study reported minimal effect on their results. The systems used in their experiments were compared before and after all of the recommended patches were applied.

Our work compares the effects patches had on the CPU's pipeline by comparing Top-Down bottleneck metrics. We did not run experiments before all patches, including the microcode and BIOS fixes, were applied so the performance baseline included the Spectre, variant 1 fix. We compared the effects of the Spectre variant 2 and Meltdown variant 3 had on the test system. This comparison was possible because the OS vendor added tunnable features that can enable or disable the two security patches, variant 2 and variant 3, to prevent a decrease in performance. To quantify relative changes of the metrics between the patch settings, we modified the Big Mac Index, a PPP theory based technique. This made it possible to compare Top-Down metric rates against a baseline performance counter, either *CPU_CLK_UNHALTED.THREAD*, or *CPU_CLK_UNHALTED.THREAD_ANY*. The goal was to determine if the number of cycles used for a given operation, stalls for instance, was relatively higher, lower or similar when compared to the same metric when the patches were disabled. This relative difference can be used to identify situations like the ones observed in the Backend Bound rates, Figs. 7.2(a) and 7.2(b), where the rates dropped or stayed the same for the regular rates, but the PPP normalized rates fell. This is the case for *371.applu331*, which had an increase in the Backend Bound rate of 5.70% but a decrease in the PPP rate of −15.05%. This drop was due to an increase of 24.43% in core cycles. Similarly for *370.mgrid331*, its regular Backend Bound rates stayed relative little change between patch settings, 3.90%. Its PPP normalized rate was found to be 28.86%, because its cycle count increased by 46.04% between patch settings. For both benchmarks, there were more cycles for the amount of stalls as compared to the baseline, so the cycles became overvalued.

Other techniques, such as the Roofline model [23], can give users an idea of how their code is performing relative to memory and floating-point peak performance. Another approach is to use statistical methods to model performance based on metrics such as cache hit rates and memory latencies [24]. These tools can provide information on how performance is affected when changes to the system settings or the code base are

made. But they have some limitations. Statistical models provide information specific to the parameters that were used to create the model. These parameters were selected after being found to be of significance to the model. The Roofline model provides information of the changes made to the system configuration, or code changes in terms of memory and floating-point performance. Our study uses a more general technique that was applied to different metrics, including different categories of the Top-Down classification method.

We showed that Top-Down classification metrics varied when the security patches were enabled. We were able to quantify the relative changes when compared to a baseline run. Additionally, the use of PPP normalized rates made it possible to put into context the large percentage changes reported by the relative difference between metrics. The next step is to understand the effects these relative changes, which are not reflected in regular metrics, have on power efficiency. Our goal is to identify relationships between CPU pipeline bottlenecks and power efficiency before and after patches are applied. Other works have focused on the effect Spectre and Meltdown patches had on power efficiency by focusing in models based on performance metrics, for instance instructions-per-cycle, and branches-per-cycle, to develop models [70]. Our future work will focus in understanding the relation between PPP rates and power efficiency.

Chapter 8

Conclusion

Performance evaluation and characterization of codes is a time consuming endeavor. There can be a large number of system settings, compiler options and code optimization techniques which need to be explored and the resulting performance metrics analyzed in order to improve performance. In the first portion of this thesis, we discussed how current approaches could be improved upon provide better performance through the use of a more methodical parameter search. In chapter three, we showed that standard practices of limited searches of the parameter space one variable at a time or a few combinations of parameters based on previously reported *peak performance* settings is not an optimal optimization approach. Limited parameter space searches might miss a better performing parameter configuration. Additionally, searches solely based on previously published optimal results might not be applicable to a different system due to hardware or software differences. Regular experimental setups that include only high and low settings are not appropriate when dealing with qualitative parameters that have more than two settings. We showed, through the use of Orthogonal Arrays, it was possible to fully search the large parameter space of the HPL benchmark configured to use both a CPU and GPU simultaneously. Each of these devices has different optimal HPL settings, and only a systematic and efficient experimental design can make it possible to find an optimal HPL configuration settings in a timely manner.

In the second portion of this thesis, we showed that there are relative differences between performance metric ratios that are missed when regular rates are compared. Performance metrics can shed information of how well utilized the different components

are that make up a computing system. Standard practice calls for the comparison of performance metrics after a change in the system configuration is made, a different compiler option is used to generate a new binary, or a change in the code is made. The hope is to see the effects that change had on the underlying system as described by the performance metric. The challenge is that some changes might have subtle effects that comparisons of regular, non-normalized, rates do not highlight. It is through the use of a technique from the field of economics, Purchasing Power Parity (PPP) theory, that we were able to highlight relative changes between rates.

The Big Mac Index is the most famous implementation of PPP theory. It has been used to compare the relative purchasing power of a currency when it is used to buy a Big Mac burger in different countries. We used the same technique, but instead of comparing burgers, we compared the relative changes between performance ratios in terms of the ratio's denominator. We showed that the same tasks under different configurations, might have similar performance metric rates, while at the same time have large relative differences in terms of the ratio's denominator. We showed that difference can be quantified into a percentage rate. This approach makes it possible to better identify trends and highlight differences between changes – differences that might have otherwise gone unnoticed by standard evaluation practices.

Performance optimization and characterization can be a complex task. Having access to precise information on the effect changes had on the system will make it easier to ascertain how the different subsystem components are being utilized. This will make it possible to improve performance of the application in a systematic and timely manner.

References

[1] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[2] Spec omp2012 documentation. [Online]. Available: https://spec.org/omp2012/Docs/index.html

[3] F. Lebeau. (2020, march) Top-down performance analysis. [Online]. Available: https://www.brighttalk.com/webcast/17792/384060/top-down-performance-analysis

[4] *MARVELL ThunderX2 PMU Events (Abridged).* Marvell, 2019.

[5] perf: Linux profiling with performance counters. [Online]. Available: http://perf.wiki.kernel.org

[6] *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile.* Arm, 2020. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/

[7] pmu-tools: Intel pmu profiling tools. [Online]. Available: https://github.com/andikleen/pmu-tools

[8] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore cpus," *Computer*, vol. 43, no. 3, pp. 24–32, 2010.

[9] J. Diamond, M. Burtscher, J. D. McCalpin, B.-D. Kim, S. W. Keckler, and J. C. Browne, "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 32–43.

[10] L. Yi, C. Li, and J. Guo, "Cpi for runtime performance measurement: The good, the bad, and the ugly," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 106–113.

[11] D. Montgomery, *Design and Analysis of Experiments, 8th Edition.* John Wiley & Sons, Incorporated, 2012.

[12] J. Yi, D. Lilja, and D. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 281–291.

[13] C. R. Rao, "Factorial experiments derivable from combinatorial arrangements of arrays," *Supplement to the Journal of the Royal Statistical Society*, vol. 9, no. 1, pp. 128–139, 1947. [Online]. Available: http://www.jstor.org/stable/2983576

[14] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[15] Y. A. Huerta, B. Swartz, and D. J. Lilja, "Determining work partitioning on closely coupled heterogeneous computing systems using statistical design of experiments," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017, pp. 118–119.

[16] Y. A. Huerta and D. J. Lilja, "Revisiting the effects of the spectre and meltdown patches using the top-down microarchitectural method and purchasing power parity theory," in *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 2021.

[17] Y. A. Huerta, B. Swartz, and D. J. Lilja, "Enhancing the top-down microarchitectural analysis method using purchasing power parity theory," in *International Workshop on Languages and Compilers for Parallel Computing LCPC 2020*. Springer, 2021.

[18] Y. Huerta and D. Lilja, "Analysis of a thunderx2 system using top-down and purchasing power parity methods," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437359.3467027

[19] Intel Corporation. (2021) Intel vtune profiler performance analysis cookbook. (accessed on 19 September 2021). [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html

[20] P. E. McKenney, "Differential profiling," in *MASCOTS '95. Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1995, pp. 237–241.

[21] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[22] K. W. Clements, *Currencies, commodities and consumption.* Cambridge [England] ; New York: Cambridge University Press, 2013.

[23] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[24] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 643–656.

[25] The Economist. (2020) The big mac index. (accessed on 18 June 2021). [Online]. Available: https://www.economist.com/news/2020/01/15/the-big-mac-index

[26] Intel microarchitecture code named skylake events. [Online]. Available: https://download.01.org/perfmon/index/skylake.html

[27] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[28] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 661–672.

[29] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[30] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.

[31] G. Box, J. S. Hunter, and W. Hunter, *Statistics for experimenters: design, innovation, and discovery.* Wiley-Interscience, 2005.

[32] L. Breiman, *Classification and Regression Trees*, ser. (The Wadsworth statistics / probability series). Wadsworth International Group, 1984.

[33] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–10.

[34] R. Rusitoru, "Armv8 micro-architectural design space exploration for high performance computing using fractional factorial," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, ser. PMBS '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:10.

[35] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff, "Statistical selection of compiler options," in *Modeling, Analysis, and Simulation of*

Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on, Oct 2004, pp. 494–501.

[36] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "Improving computer architecture simulation methodology by adding statistical rigor," *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1360–1373, Nov 2005.

[37] Top500 Supercomputer Sites, http://top500.org.

[38] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 455–470.

[39] S. Pakin, C. Storlie, M. Lang, R. E. Fields, E. E. Romero, C. Idler, S. Michalak, H. Greenberg, J. Loncaric, R. Rheinheimer, G. Grider, and J. Wendelberger, "Power usage of production supercomputers and production workloads," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 274–290, 2016.

[40] X. Tang, J. Zhai, B. Yu, W. Chen, and W. Zheng, "Self-checkpoint: An in-memory checkpoint method using less space and its practice on fault-tolerant hpl," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 401–413.

[41] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51.

[42] E. Castillo, *Process Optimization: A Statistical Approach*, ser. International Series in Operations Research & Management Science. Springer US, 2007.

[43] A. Hedayat, N. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*, ser. Springer Series in Statistics. Springer New York, 2012.

[44] K. Mouzakitis, "High performance linpack and gadget 3 investigation," M.S. thesis, EPCC, The Univ. of Edinburgh, 2014.

[45] D. Rohr, J. d. Cuveland, and V. Lindenstruth, "A model for weak scaling to many gpus at the basis of the linpack benchmark," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2016, pp. 192–202.

[46] M. Sindi. Howto high performance linpack (hpl) on nvidia gpus. [Online]. Available: http://hpl-calculator.sourceforge.net/Howto-HPL-GPU.pdf

[47] B. Subramaniam and W.-c. Feng, "Gbench: benchmarking methodology for evaluating the energy efficiency of supercomputers," *Computer Science - Research and Development*, vol. 28, no. 2, pp. 221–230, 2013.

[48] T. Z. Tan, R. S. M. Goh, V. March, and S. See, "Data mining analysis to validate performance tuning practices for hpl," in *2009 IEEE International Conference on Cluster Computing and Workshops*, Aug 2009, pp. 1–8.

[49] perf: Linux profiling with performance counters. [Online]. Available: http://perf.wiki.kernel.org

[50] The gnu compiler collection website. [Online]. Available: http://gcc.gnu.org

[51] Intel vtune profiler. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html

[52] A. Yasin, J. Haj-Yahya, Y. Ben-Asher, and A. Mendelson, "A metric-guided method for discovering impactful features and architectural insights for skylake-based processors," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3369383

[53] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 202–211.

[54] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 643–656.

[55] Intel vtune profiler performance analysis cookbook. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top.html

[56] "The latte index: Using the impartial bean to value currencies." [Online]. Available: https://www.visualcapitalist.com/latte-index-currencies/

[57] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran, "Spec omp2012 — an application benchmark suite for parallel systems using openmp," in *OpenMP in a Heterogeneous World*, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 223–236.

[58] Spec omp2012 results. [Online]. Available: https://www.spec.org/omp2012/results/

[59] Y. Huerta and D. Lilja, "Analysis of a thunderx2 system using top-down and purchasing power parity methods," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437359.3467027

[60] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier Science, 2017.

[61] J. Diamond, M. Burtscher, J. D. McCalpin, B. Kim, S. W. Keckler, and J. C. Browne, "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 32–43.

[62] The Economist Intelligence Unit. (2017) The ipod index. (accessed on 8 January 2021). [Online]. Available: https://www.intelligenteconomist.com/the-ipod-index/

[63] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 357–367.

[64] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[65] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[66] V. G. Vergara Larrea, M. J. Brim, W. Joubert, S. Boehm, M. Baker, O. Hernandez, S. Oral, J. Simmons, and D. Maxwell, "Are we witnessing the spectre of an hpc meltdown?" *Concurrency and Computation: Practice and Experience*, vol. 31, no. 16, p. e5020, 2019, e5020 cpe.5020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5020

[67] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, C. Yee, A. Reuther, and J. Kepner, "Measuring the impact of spectre and meltdown," pp. 1–5, 2018.

[68] Red Hat, Inc. (2018) Meltdown & spectre - kernel side-channel attacks. (accessed on 18 June 2021). [Online]. Available: https://access.redhat.com/security/vulnerabilities/speculativeexecution

[69] ——. (2020) Controlling the performance impact of microcode and security patches for cve-2017-5754 cve-2017-5715 and cve-2017-5753 using red hat enterprise linux tunables. (accessed on 18 June 2021). [Online]. Available: https://access.redhat.com/articles/3311301

[70] B. Herzog, S. Reif, J. Preis, W. Schröder-Preikschat, and T. Hönig, "The price of meltdown and spectre: Energy overhead of mitigations at operating system level," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 8–14. [Online]. Available: https://doi.org/10.1145/3447852.3458721