BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100011

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Revisiting the effects of the Spectre and Meltdown patches using the top-down microarchitectural method and purchasing power parity theory

Yectli A. Huerta [a,c,d,*], David J. Lilja [a,b,d]

[a] Scientific Computation
[b] Electrical and Computer Engineering Department
[c] Minnesota Supercomputing Institute
[d] University of Minnesota, USA

ARTICLE INFO

Keywords:
Bottleneck analysis
System characterization
Performance measurement

ABSTRACT

Software patches are made available to fix security vulnerabilities, enhance performance, and usability. Previous works focused on measuring the performance effect of patches on benchmark runtimes. In this study, we used the Top-Down microarchitecture analysis method to understand how pipeline bottlenecks were affected by the application of the Spectre and Meltdown security patches. Bottleneck analysis makes it possible to better understand how different hardware resources are being utilized, highlighting portions of the pipeline where possible improvements could be achieved. We complement the Top-Down analysis technique with the use a normalization technique from the field of economics, purchasing power parity (PPP), to better understand the relative difference between patched and unpatched runs. In this study, we showed that security patches had an effect that was reflected on the corresponding Top-Down metrics. We showed that recent compilers are not as negatively affected as previously reported. Out of the 14 benchmarks that make up the SPEC OMP2012 suite, three had noticeable slowdowns when the patches were applied. We also found that Top-Down metrics had large relative differences when the security patches were applied, differences that standard techniques based in absolute, non-normalized, metrics failed to highlight.

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 [1,2]. 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 of the Spectre and Meltdown security patches had on performance were published. In one study, a number of Cray supercomputers were used to analyse 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 [3]. Another study, showed the effects different patches had on two computational intensive workflows, pMatlab and Keras with TensorFlow, on a Intel based cluster [4]. 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 [5]. 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 [6]. 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 [7], can provide additional information on the metric drift. This technique has been used to account for differences across GCC compiler suite releases [8] using the Top-Down bottleneck classification method. PPP

* Corresponding author at: Scientific Computation .
E-mail addresses: yhuerta@umn.edu (Y.A. Huerta), lilja@umn.edu (D.J. Lilja).

https://doi.org/10.1016/j.tbench.2021.100011
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 4 November 2021

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 [9], where it was used to analyse 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.

2. Background

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 [2]. The Spectre exploit allows attackers to gain access to private information through branch mispredictions [1]. 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: [11,12]:

- *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 analyse the performance impact of the security patches for Spectre, variant 2, and Meltdown, variant 3, had on the system.

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 [5]. 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 cannot 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 analyse 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 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 [13].

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 [7,14,15]. The most famous PPP index is the Big Mac Index (BMI), which was developed by The Economist magazine [16]. 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. Eq. (1) computes the Big Mac exchange rate which is based on its local price.

$$20/5 = 4 \tag{1}$$

Eq. (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, Eq. (2) shows that the yuan is 37.5% undervalued as compared to the US dollar.

$$(4 - 6.4) * 100/6.4 = -37.5 \tag{2}$$

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

Table 1
Top-Down Metric formulas for Intel Skylake processor [5,10].

Metric	Formula
CORE_CLKS	CPU_CLK_UNHALTED.THREAD_ANY/2
CLKS	CPU_CLK_UNHALTED.THREAD
SLOTS	4 * CORE_CLKS
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
Recovery_Cycles	INT_MISC.RECOVERY_CYCLES_ANY/2
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
L2_Bound_Ratio	(CYCLE_ACTIVITY.STALLS_L1D_MISS-CYCLE_ACTIVITY.STALLS_L2_MISS) / CLKS
LOAD_L2_HIT	MEM_LOAD_RETIRED.L2_HIT* (1+MEM_LOAD_RETIRED.FB_HIT/MEM_LOAD_RETIRED.L1_MISS)
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

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 1. The use of *CPU_CLK_UNHALTED. THREAD*, or *CPU_CLK_UNHALTED.THREAD_ANY* to compute the PPP Exchange Rate, Eq. (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.

$$PPP\_Exchange\_Rate = CPU\_clk/CPU\_clk_{baseline} \tag{3}$$

Eq. (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.

$$PPP = 100 * ((Metric/Metric_{baseline}) - PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \tag{4}$$

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 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, Eq. (5). The drift between security patch settings was found to be −52.42%, Eq. (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.

$$81671652350125/55924600475776 = 1.46 \tag{5}$$

$$100 * ((0.06789046/0.09773369) - 1.46)/1.46 = -52.42 \tag{6}$$

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 2
SPEC OMP2012 Benchmark description [17].

Benchmark name	Programming language	Description
350.md	Fortran	Physics: Molecular dynamics
351.bwaves	Fortran	Physics: Computational Fluid Dynamics (CFD)
352.nab	C	Molecular Modelling
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

3. Experimental setup

We used the Top-Down method in combination with the SPEC OMP2012 benchmarks [17,18] 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 2. OMP2012 results that followed the SPEC reporting guidelines can be submitted for publication [19]. A two socket Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz 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 [12]. 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.

4. Analysis of results

Fig. 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 analysed 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.

Table 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 of retiring slots increased when patches were enabled. *358.botsalgn* had the opposite effect.

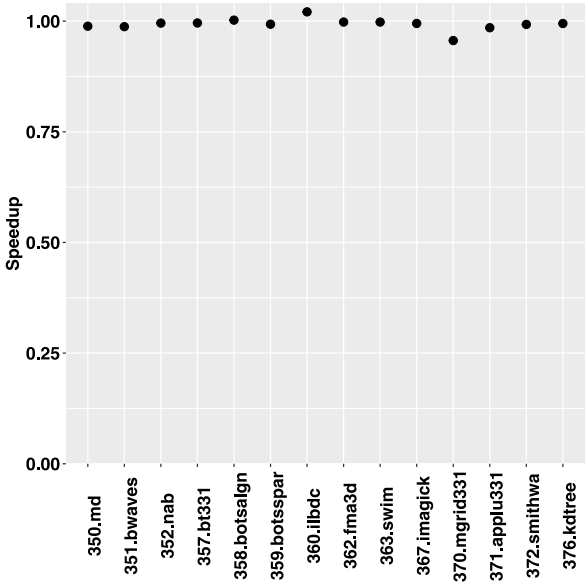


Fig. 1. Speedup comparison for the Intel 2021.1 compiler suite when patches are enabled using SPEC OMP2012 benchmarks with 32 threads and SMT enabled. Higher is better.

For the Frontend Bound metric, *358.botsalgn*, *359.botsspar*, *360.ilbdc* and *370.mgrid331* 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.

Table 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	359.botsspar	Few_Uops_Executed_Threshold
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
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

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.

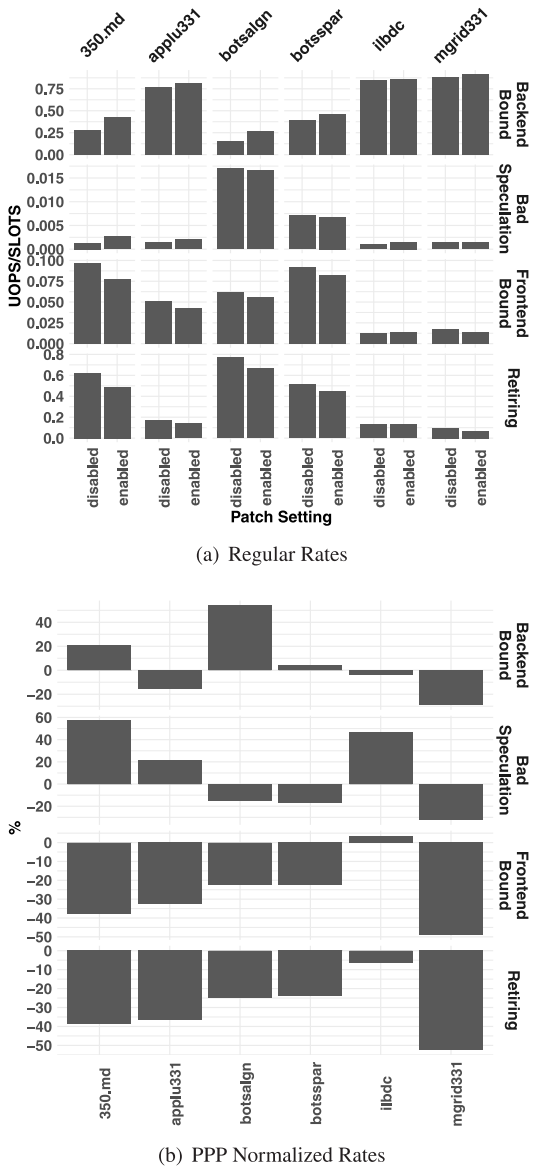
4.1. Top-Down metrics

Fig. 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. Fig. 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.mgrid*. 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.

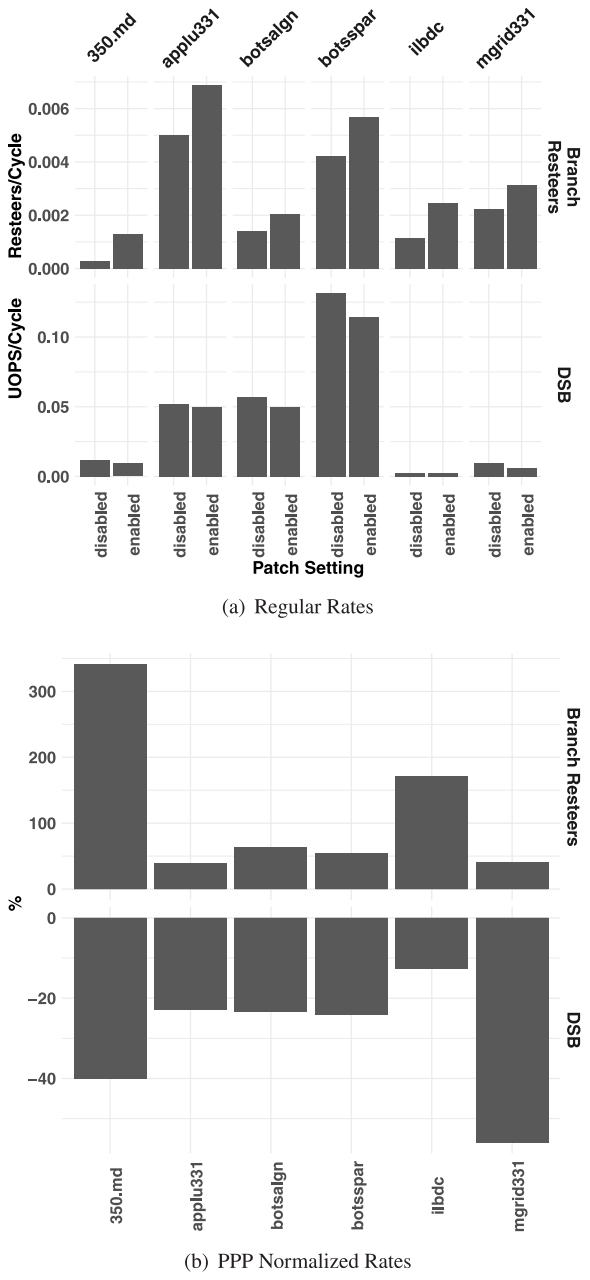
The *360.ilbdc* benchmark saw less than a 1% difference in the Backend Bound metric between patch settings. For *360.ilbdc*, the Bad Speculation, Front End, and Retiring metrics stayed relatively the same, resulting in a very similar Backend Bound rate. All other benchmarks had an increase in the Backend 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



(a) Regular Rates

(b) PPP Normalized Rates

Fig. 2. Results of the Top-Down architectural bottleneck classification main categories.



(a) Regular Rates

(b) PPP Normalized Rates

Fig. 3. Frontend Bound subcategories.

−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.

4.1.1. Frontend bound

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. Fig. 3(a) shows that the Branch Resteers 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 Resteers 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 Resteers 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, Fig. 3(b). 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%.

4.1.2. Retiring

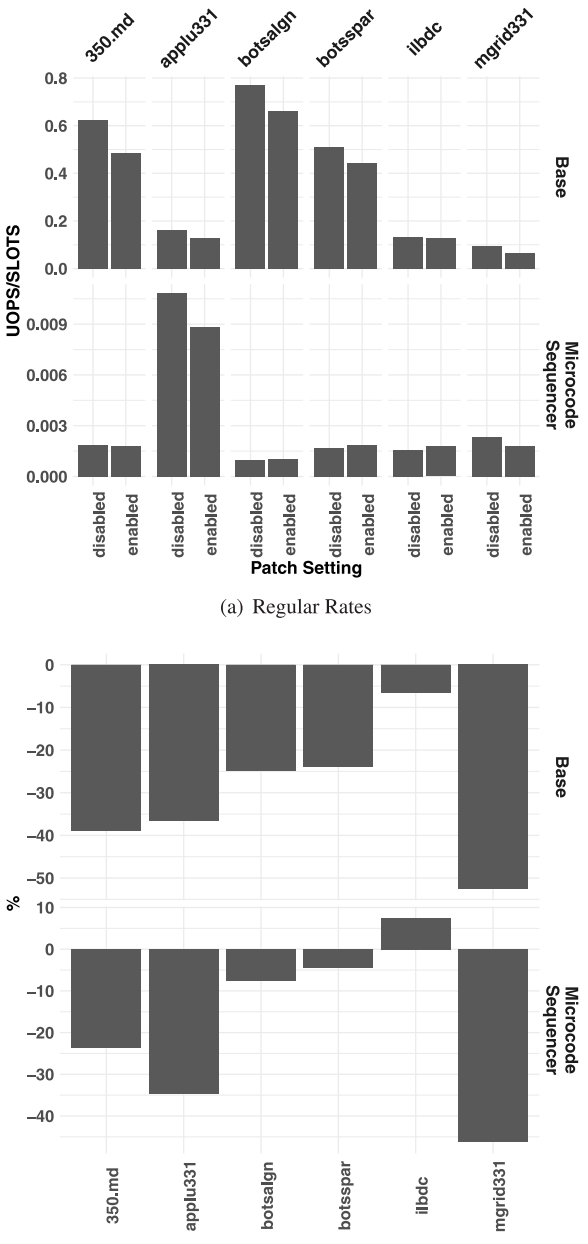
The Retiring category represents the fraction of pipeline slots of useful work, the uops that were eventually retired. The Microcode Sequencer metrics is a retiring subcategory that accounts for pipeline slots of uops that were retired and were fetched by the microcode sequencer ROM. The Base metric tracks retired uops that did not originate from the microcode sequencer. Fig. 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, Fig. 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.

4.1.3. 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 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.

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, Fig. 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%



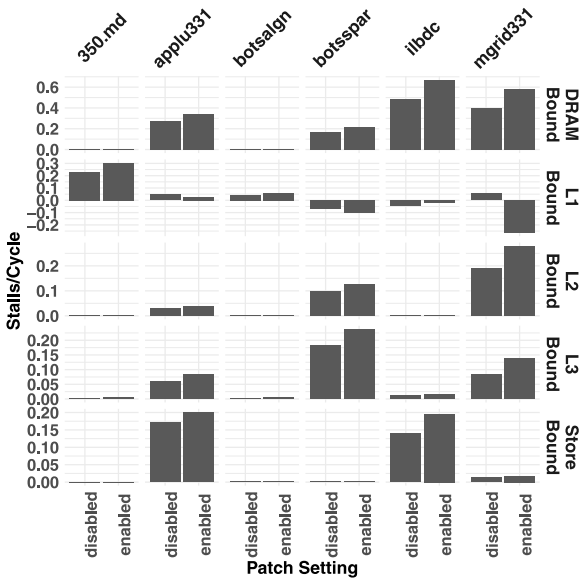
(a) Regular Rates

(b) PPP Normalized Rates

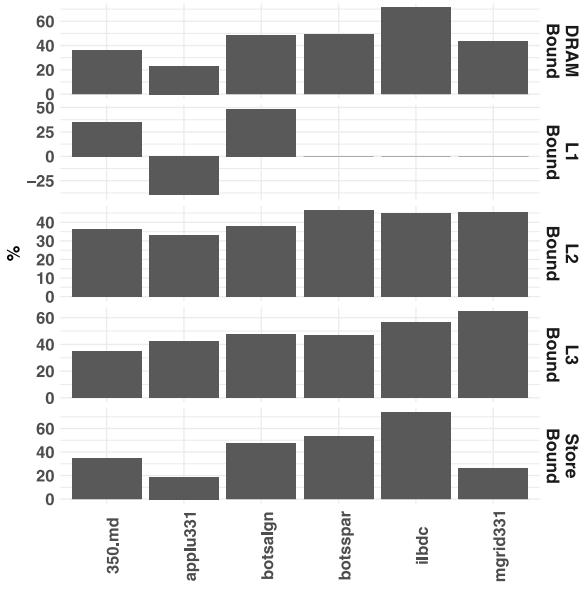
Fig. 4. Retiring subcategory.

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



(a) Regular Rates



(b) PPP Normalized Rates

Fig. 5. Memory Bound subcategories which are part of the Backend Bound classification.

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.

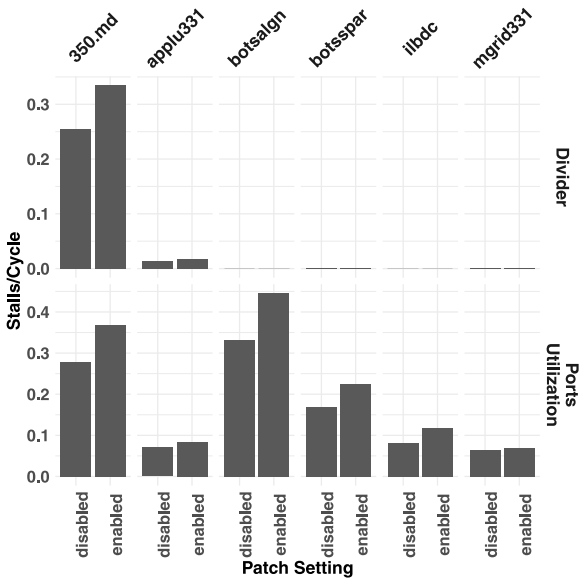
PPP rates for DRAM, L2, L3 and Store Bound metrics were consistently positive across all benchmarks, Fig. 5(b). This was the result of increasing stall rates across these metrics while the number of cycles decreased. There were fewer cycles to handle the increasing number of stalls, so the cycles became 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.

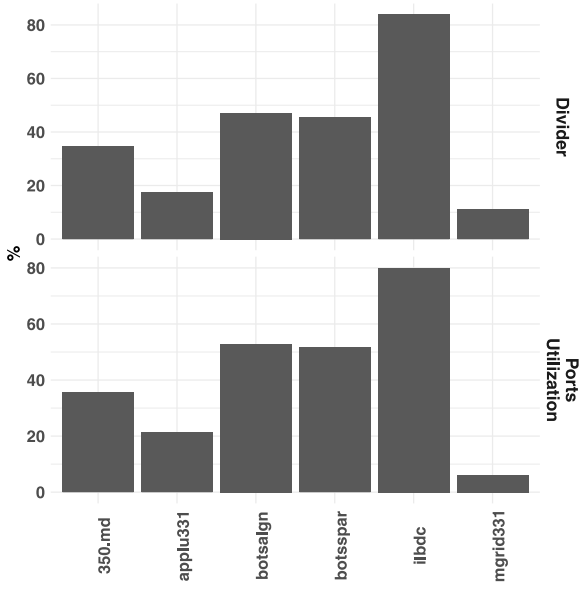
Core Bound is the second set of subcategories of the Backend Bound classification. They represent all non-memory related bottlenecks. The Divider metric tracks the fraction of cycles in which divide and square root operations used the DIV unit. When patches were enabled, the number of cycles when the divide unit was busy, ARITH.DIVIDER_ACTIVE, increased, while the number of CPU_CLK_UNHALTED.THREAD remained the same or decreased. Fig. 6(a) shows that the *360.ilbdc* benchmark 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 do not 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, Fig. 6(b). 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

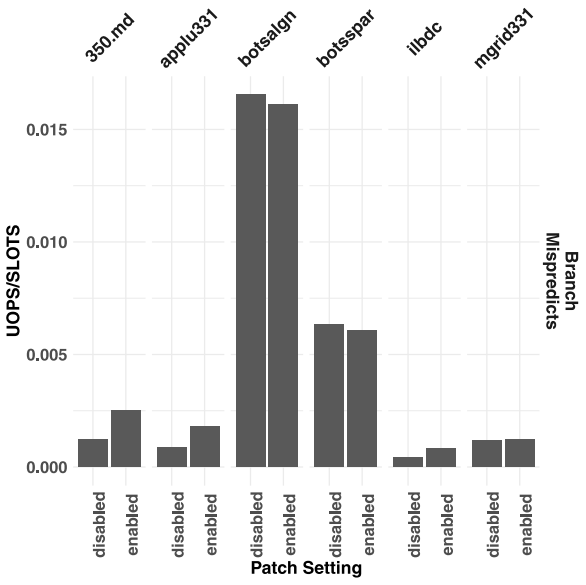


(a) Regular Rates

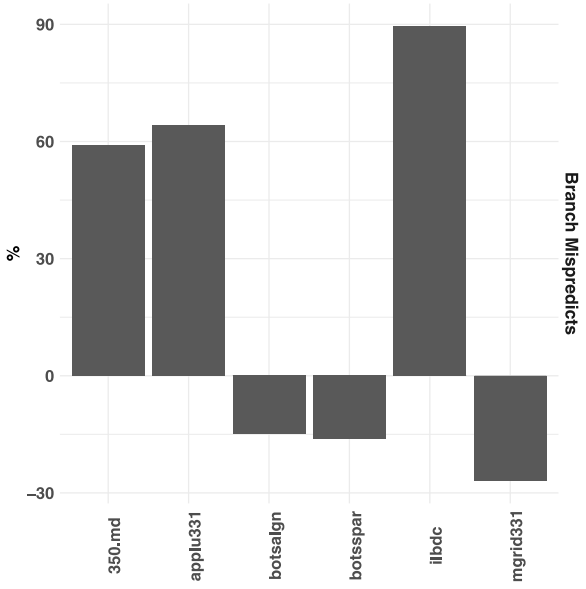


(b) PPP Normalized Rates

Fig. 6. Core Bound subcategories, which are part of the Backend Bound classification.



(a) Regular Rates



(b) PPP Normalized Rates

Fig. 7. Bad Speculation subcategories.

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%.

4.1.4. 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 analysed 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 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 increase in the Bad Speculation metric, Fig. 7(a). 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 Fig. 7(b). 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.

5. Conclusion

In this study, we analysed 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 [4] on an Intel based cluster, and on a diverse set of multiple benchmarks on different Cray based clusters [3]. 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 tunable 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 tunable 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. 2(a) and 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 [20], 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 [21]. 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 [22]. Our future work will focus in understanding the relation between PPP rates and power efficiency.

CRediT authorship contribution statement

Yectli A. Huerta: Conceptualization of this study, Methodology, Data curation, Writing – Original draft preparation. **David J. Lilja:** Editing of draft, final draft revision.

References

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

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

[3] V.G. Vergara Larrea, M.J. Brim, W. Joubert, S. Boehm, M. Baker, O. Hernandez, S. Oral, J. Simmons, D. Maxwell, Are we witnessing the spectre of an HPC meltdown? Concurr. Comput.: Pract. Exper. 31 (16) (2019) e5020, http://dx.doi.org/10.1002/cpe.5020, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5020, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5020, e5020 cpe.5020.

[4] 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, J. Kepner, Measuring the impact of spectre and meltdown, in: 2018 IEEE High Performance Extreme Computing Conference, HPEC, 2018, pp. 1–5, http://dx.doi.org/10.1109/HPEC.2018.8547554.

[5] 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.

[6] 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.

[7] K.W. Clements, Currencies, Commodities and Consumption, Cambridge University Press, 2013, http://dx.doi.org/10.1017/CBO9781139045612.

[8] Y.A. Huerta, B. Swartz, 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.

[9] Y. Huerta, D. Lilja, Analysis of a ThunderX2 system using top-down and purchasing power parity methods, in: Practice and Experience in Advanced Research Computing, PEARC '21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3437359.3467027.

[10] A. Kleen, pmu-tools: Intel PMU profiling tools, 2021, URL: https://github.com/andikleen/pmu-tools, (accessed on 18 June 2021).

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

[12] Red Hat, Inc., 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, 2020, URL: https://access.redhat.com/articles/3311301, (accessed on 18 June 2021).

[13] Intel Corporation, Intel vtune profiler user guide, 2021, URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html.
[14] Visual Capitalist, The latte index: Using the impartial bean to value currencies, 2017, URL: https://www.visualcapitalist.com/latte-index-currencies/, (accessed on 10 May 2021).
[15] The Economist Intelligence Unit, The iod index, 2017, URL: https://www.intelligenteconomist.com/the-ipod-index/, (accessed on 8 January 2021).
[16] The Economist, The big mac index, 2020, URL: https://www.economist.com/news/2020/01/15/the-big-mac-index, (accessed on 18 June 2021).
[17] Standard Performance Evaluation Corporation, SPEC OMP2012 documentation, URL: https://spec.org/omp2012/Docs/index.html.
[18] 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, K. Kumaran, SPEC OMP2012 — An application benchmark suite for parallel systems using openmp, in: B.M. Chapman, F. Massaioli, M.S. Müller, M. Rorro (Eds.), OpenMP in a Heterogeneous World, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 223–236.
[19] Standard Performance Evaluation Corporation, SPEC OMP2012 results, URL: https://www.spec.org/omp2012/results/.
[20] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785.
[21] G. Ayers, J.H. Ahn, C. Kozyrakis, P. Ranganathan, Memory hierarchy for web search, in: 2018 IEEE International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2018, pp. 643–656, http://dx.doi.org/10.1109/HPCA.2018.00061.
[22] B. Herzog, S. Reif, J. Preis, W. Schröder-Preikschat, 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, EuroSec '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 8–14, http://dx.doi.org/10.1145/3447852.3458721.

Yectli A. Huerta is a Ph.D. candidate in Scientific Computation at the University of Minnesota. He works as an HPC Systems Administrator at the Minnesota Supercomputing Institute.

David J. Lilja received a Ph.D. and an M.S., both in Electrical Engineering, from the University of Illinois at Urbana-Champaign, and a B.S. in Computer Engineering from Iowa State University in Ames. He is currently Professor of Electrical and Computer Engineering at the University of Minnesota in Minneapolis, where he also serves as a member of the graduate faculties in Computer Science, Scientific Computation, and Data Science. He was elected a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow of the American Association for the Advancement of Science (AAAS) for contributions to the statistical analysis of computer performance.