Impact of Increased Cache Misses on Runtime Performance of MPX-enabled Programs

Niti Sharma

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science & Application

Xun Jian, Chair

Dongyoon Lee

Changhee Jung

April 25, 2019

Blacksburg, Virginia

Keywords: Spatial Security, Memory Protection Extensions, Caches, Benchmarks, runtime, Overheads, TLB.

Impact of Increased Cache Misses on Runtime Performance of MPX-enabled Programs

Niti Sharma

(ABSTRACT)

Low level languages like C and C++ provide high performance and direct control over memory management. But these languages are prone to memory safety violations. Intel introduced a new ISA extension-Memory Protection Extension(MPX), a hardware-assisted full-stack solution, to protect against the memory safety violations. While MPX efficiently prevents memory errors like buffer overflows and out of bound memory accesses, it comes at the cost of high performance overheads. Also, the cache locality worsens in MPX protected applications.

In our research, we analyze if there is a correlation between increase in cache misses and runtime degradation in programs compiled with MPX support. We analyze 15 SPEC CPU benchmark programs for different input sizes on Windows platform, compiled with Intel's ICC compiler. We find that for input sizes train(medium) and ref(large), the average performance overheads are 140% and 144% respectively. We find that 5 out of 15 benchmarks do not have any runtime overheads and also, do not have any change in cache misses at any level. However for rest of the 10 benchmarks, we find a strong correlation between runtime overheads and cache misses overheads, with the correlation coefficients ranging from 0.8 to 0.36 for different input sizes. Based on our findings, we conclude that there is a direct correlation between runtime overheads and increase in cache misses. We also find that instructions overheads and runtime overheads have a positive correlation, with the coefficient values ranging from 0.7 to 0.33 for different input sizes.

Impact of Increased Cache Misses on Runtime Performance of MPX-enabled Programs

Niti Sharma

(GENERAL AUDIENCE ABSTRACT)

Low level programming languages like C and C++ are primary choices to write low-level systems software such as operating systems, virtual machines, embedded software, and performance-critical applications. But these languages are considered as unsafe and prone to memory safety errors. Intel introduced a new technique- Memory Protection Extensions(MPX) to protect against these memory errors. But prior research found that applications supported with MPX have increased runtimes(slowdowns).

In our research, we analyze these slowdowns for different input sizes(medium and large) in 15 benchmark applications. Based on the input sizes, the average slowdowns range from 140% to 144%. We then examine if there is a correlation between increase in cache misses under MPX and the slowdowns. A hardware cache is a component that stores data so that future requests for that data can be served faster. Hence, cache miss is a state where the data requested for processing by a component or application is not found in the cache. Whenever a cache miss happen, the processor waits for the data to be fetched from the next cache level or from main memory before it can continue to execute. This wait influences the runtime performance of the application. Our evaluations find that 10 out of 15 applications which have increased runtimes, also have increase in cache misses. This shows a positive correlation between these two parameters. Along with that, we also found that increase in instruction size in MPX protected applications also has a direct correlation with the runtime degradation. We also quantify these relationships with a statistical measure called correlation coefficient.

Dedication

Dedicated to my parents, Saroj and Azad Kumar Sharma.

Acknowledgments

First and foremost, I express my gratitude to my advisor Dr. Xun Jian for advising me throughout my research. It was only with his patience and constant support that I was able to finish my research.

I would also like to thank Dr. Dongyoon Lee and Dr. Changhee Jung for serving as my committee members. It was a great pleasure acting as Teaching Assistant for Dr. Lee's course. With his course, I was able to refresh so many concepts related to my research. He was patient with all my queries and extended all the support in helping me complete my research. I am really grateful to Dr. Jung for pointing in the right direction when I was stuck in setting up my experiments. A major part of setting up the benchmark run could only be achieved with the help of his student. Their constructive feedbacks helped me improve my work.

Thanks to my best friend, Aabhas Bhatia for his constant motivation and support. Last but not the least, I would like to thank my family: my parents and my brother for all the love, motivation and support.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Low level languages like C and C++ are known for their high performance, direct access to the underlying hardware, and complete control over memory management. Therefore, they remain the primary choices to write low-level systems software such as operating systems, virtual machines, language runtimes, embedded software, and performance-critical applications. While these low-level languages allow complete control over the memory layout, this access can also lead to violations of memory safety [50]. Memory safety violations can be classified into two categories: Spatial errors and Temporal errors.

Spatial memory safety ensures that whenever a pointer is dereferenced, it always refers to an address which is within the bounds of its referent. These "legal" bounds are defined whenever an object is allocated. Pointers that point outside of their associated object may not be dereferenced [40]. Therefore, activities like trying to access outside the bounds of an array, dereferencing pointers obtained from invalid pointer arithmetic etc. violate spatial security. To generalize, a spatial memory error occurs whenever a program reads or writes to an unintended memory region. Spatial errors are also known as Buffer overflows [25] or out-of-bound accesses. Temporal memory safety is a property that ensures that all memory dereferences are valid at the time of the dereference [40]. Therefore, a temporal memory error occurs when a program tries to access an object which doesn't exist. The access becomes illegal if it is attempted before the object was created or after it was deleted.

There are several reliability and security related issues associated with these memory safety

violations. These violations may result in crashes, data loss or even buggy behaviour. Even more intimidating is the possibility that the hackers can exploit these errors to build a security attack. Since 2016, over 5000 memory security vulnerabilities have published as reported by the US National Vulnerability Database [3]. Since such systems often consist of millions of lines of code, it is not feasible to transition the computing ecosystem away from C and its variants.

Therefore, many bounds-checking techniques have been proposed over the years to fend off these attacks. But these state-of-the-art techniques suffer from high performance overheads. Therefore, Intel released a new technique Memory Protection Extensions (Intel MPX) as an alternative that has lower runtime overheads. This technique provides efficient hardware assistance which it does by software-based bounds checking. To enable this check, it introduced new instructions and bounds registers.

Prior research has found that despite the effective spatial memory security it provides, MPX still exhibits high performance overheads. This can go as high as 4 times w.r.t native version. After applying all the compiler optimization, it can be amortized to approximately 50% [39]. In our research, we focus on analyzing the impact of cache misses on the performance overheads. Specifically, we examine if there is any correlation between increase in cache misses and runtime overheads in applications compiled with MPX support.

We explore the performance overheads using SPEC CPU 2006 [5] and SPEC CPU 2017 [6] benchmarks. We select 15 benchmarks which compile successfully with MPX flags and run without raising any errors. We use Intel's ICC compiler on Windows platform. We collected the runtimes across two sizes: train and ref with MPX enabled. Then, we compare the runtime overheads with respect to the native version(without MPX). The MPX technique checks pointer bounds for each access. These bounds are stored into a bounds table (if they cannot be stored in bounds registers anymore). The bounds loading process is non-contiguous and therefore, the MPX-protected applications are expected to have worse cache locality.

Therefore, we expect to see increase in cache misses when applications are compiled with MPX. As we know that cache misses cause poor runtime performance [20, 48], we expect that whenever we see runtime degradation because of MPX, we should see an increase in cache misses.

Using VTune Amplifier, we find the raw numbers of cache misses and number of instructions for a benchmark program run. Then, we use these statistics to report our numbers in terms of cache misses per kilo instructions to understand these overheads. Along with the cache misses, another factors that may impact the performance is increase in TLB (Translation Lookaside buffer) misses and instructions overheads. TLB misses are even more expensive than data and instruction cache misses because of page walk.

We use VTune Ampilifer to collect important statistics which included L1 cache misses, L2 cache misses, L3 cache misses, TLB misses and total instructions. With our findings, we answer the following questions:

- How much performance overhead is incurred over different input sizes when spatial security is imposed using MPX ?
- Is there a relationship between runtime overheads and instructions overheads?
- Is there a correlation between increase in cache misses over different hierarchy levels and performance overheads with MPX?
- Do TLB misses have any impact on the performance overheads?

Chapter 2

Background

In this chapter, we discuss some background information to understand the motivation behind this research. The section 2.1 introduces the pressing issue of Memory safety violations in low level languages. We then focus on Spatial Memory security as it is the scope of our research. We will briefly touch upon the state-of-the-art techniques that enable spatial security(along with Temporal security in some of them) in section 2.2. With this discussion, we will understand the need of introducing the MPX technique by Intel.

2.1 Memory Safety Violations

Systems software provides platform to other software. It is designed to run a computer's hardware and application programs. Specifically, it can be considered as an interface between the hardware and user applications. Some of the examples of systems software are Operating Systems, boot program, virtual machines, language runtimes, embedded software, assembler and others [11]. Low level languages like C and C++ are used to write these systems software, some of which are critical applications [13]. However, these languages are considered as unsafe as their direct access to memory layout often leads to memory safety violations. Unlike memory safe languages like Java, C and C++ do not do any Bounds checking and pointer dereference check during runtime. Some of the common memory errors faced by developers are listed below:

- **Dangling and Wild Pointers.** These are the pointers that point to a memory location even after that location has been deleted or freed. Let us discuss the example of arrays to explain dangling pointers. In C and C++, pointers can refer to an individual array element rather than referring whole array. Therefore, this pointer can persist even after the array in question has ceased to exist [30]. For instance, we have stored copy of an address in a pointer during some operation and later, we have deleted that array. Now, the pointer is pointing to location of de-allocated memory. An operation to dereference this dangling pointer may cause unpredictable behaviour, program crash or a segmentation fault.

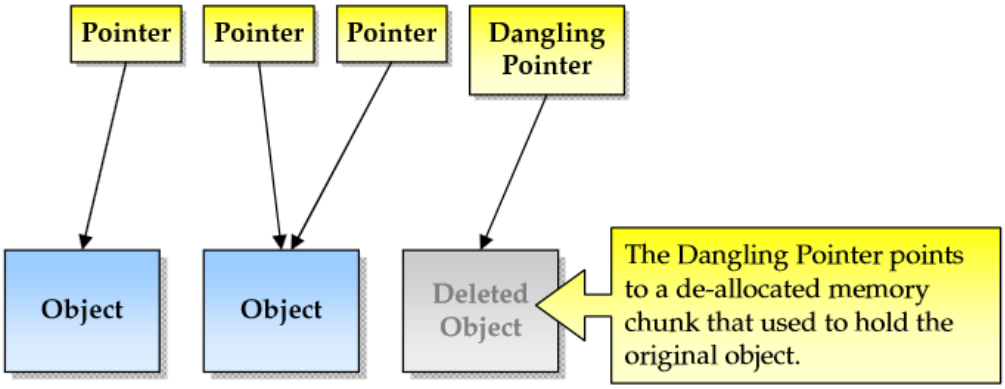


Figure 2.1: Dangling Pointer [29].

Whereas a Wild pointer is the one that is not initialized with any address. By default, it holds a garbage value. Uninitialized pointer's behavior is totally undefined because it may point some arbitrary location that can cause program crash [19].

- **Out-of-bound access.** These languages have no provision to check the out-of-bounds accesses. Particularly, in case of arrays, a programmer can access an address that is out of permitted bounds. In this case, no exception is raised and an undefined behaviour is exhibited.

Listing 2.1 shows a code in which we have an array with 10 elements. While the first print statement at line 5 prints an element which is present within the bounds of the array, the second print statement at line 8 is accessing an element which lies outside the legal bounds of the array.

Listing 2.1: Code vulnerable to out of bound accesses

```
1   #include <stdio.h>
2   int main()
3   {
4       int arr[] = {1,2,3,4,5,6,7,8,9,10};
5       printf("arr [0] is %d\n", arr[0]);
6
7       // arr[10] is out of bound
8       printf("arr[10] is %d\n", arr[100]);
9       return 0;
10  }
```

Compiling and running this C code for the first time yields an output where the first print statement shows expected behaviour. However, the second statement prints some garbage value as seen in the Listing 2.2.

Listing 2.2: Results for first run

```
arr [0] is 1
arr[10] is -817563373
```

A second run as shown in Listing 2.3 shows the same value for first print but some random (garbage) value for the second run. To an unsuspecting user, these values may ”seem normal”. However, these values are incorrect and introduce hard to fix bugs in the code.

Listing 2.3: Results for second run

```
arr [0] is 1
arr[10] is -252893517
```

- **Buffer Overflows.** It is also known as buffer overrun. Buffer Overflows occur when a fixed-length buffer cannot handle the data which is more than its size. A buffer is a temporary area for data storage. Since the extra information has to go somewhere, it spills onto adjacent memory space. Some implications of this are data corruption or data overwriting in the invaded space [16].

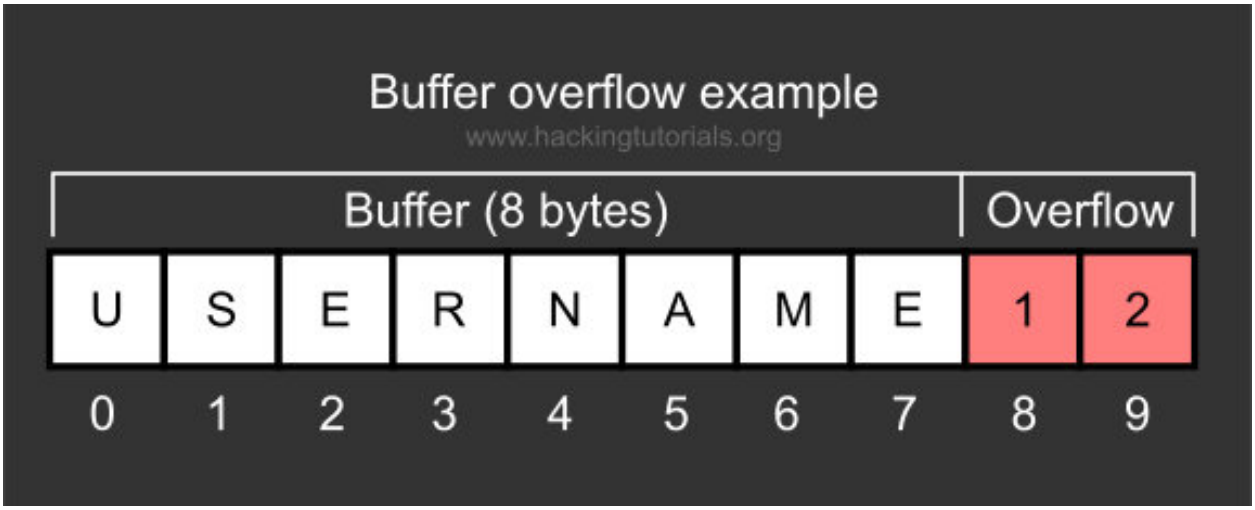


Figure 2.2: Buffer Overflow [49]

These memory violations can be grouped into two categories. The first one, pertaining to the dangling and wild pointers, are termed as Temporal errors. Whereas, the other two-Out of bound access and Buffer Overflow, are termed as spatial errors. These memory-safety violations can be cause of several problems like program crash, data loss and hard to resolve bugs. A cybercriminal can exploit these vulnerabilities to gain access to an illegal memory region and can steal confidential data too.

Taking an example of Buffer overflow errors, cybercriminals can exploit these errors by altering the execution path of the application by overwriting parts of its memory [16]. The extra data that will overflow may contain malicious code to trigger specific actions like gaining unauthorized access to the system, damaging files or stealing private information [43]. These attacks that exploit the memory vulnerabilities in low-level languages are a growing concern. Thousands of memory vulnerabilities are published every year according to the US National Vulnerability Database [3].

Despite these security vulnerabilities, C and C++ remain prevalent because they provide high performance, direct access to the underlying hardware, and explicit control over memory usage and runtime behavior of the program. C predates many newer ”safe” languages. The idea of safety came into picture only with experience in C. Also, C is relatively small and easy to implement [7]. Moreover, since systems often consist of millions of lines of code, changing the computing ecosystem from C/C++ to a memory safe language is not feasible any time soon [13].

These errors can be hard to reproduce, hard to debug, and potentially expensive to correct as well. A lot of research has been done to come up with solutions that can enforce memory safety.

2.2 Techniques to impose Memory security

In this section, we will briefly mention state-of-the-art solutions to provide protection against memory errors. State-of-the-art solutions include pure static analysis [28, 52], hardware-based checking [31, 35, 44, 51], probabilistic methods [22, 33, 37], and extensions of the C/C++ languages. Dor et. al introduced a tool, CSSV to statically detect all Buffer Overflows in C[28]. Xie et. al developed ARCHER (ARray CHeckER) a static, effective

memory access checker. ARCHER uses path-sensitive, interprocedural symbolic analysis to bound the values of both variables and memory sizes [52].

Out of all these methods, runtime bounds-checking is regarded as the most preferable way of defending against all the memory attacks [36, 47]. However, existing bounds-checking techniques come at the cost of high performance overheads which range from 50 to 150%, making them impractical to adopt on a wider scale. In 2013, Intel introduced a new ISA extension that enables runtime bounds-checking. Known as Memory Protection Extensions (Intel MPX), it provides protection with lower runtime overheads compared to the state-of-the-art bounds-checking techniques. It provides hardware assistance using new instructions and registers that do software-based bounds checking.

Chapter 3

Intel's MPX

In this chapter, we introduce Intel's MPX in section 3.1. In section 3.1, we detail the implementation of MPX in the hardware stack. We do this so as to build the basis for our main research question. In 3.2, we discuss the prior work done by Oleksenko et. al. This is main prior work which we use as our motivation. In section 3.3, we explain the motivation behind our research and our research question. In section 3.4, we list our contributions and how our work is different from the prior work.

3.1 Introduction

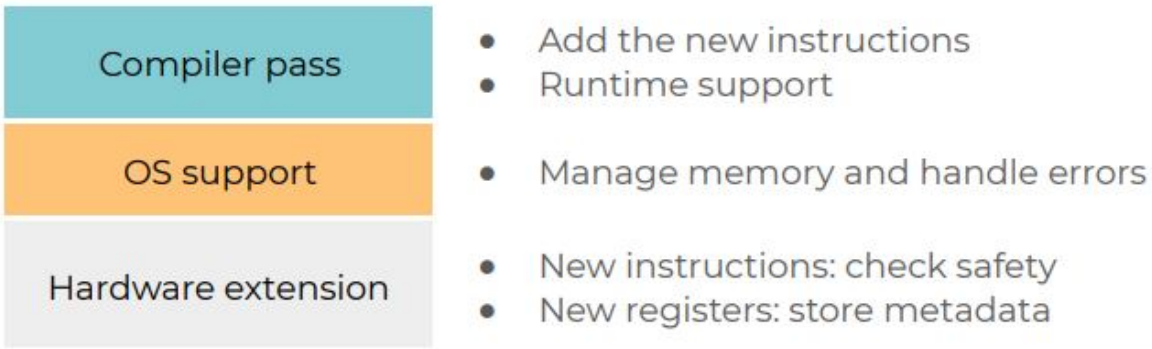


Figure 3.1: Modifications in Hardware Stack by MPX [38]

Intel's MPX(Memory Protection Extensions) leverages the runtime bounds-checking technique. It was first introduced as part of the Skylake microarchitecture in 2015. MPX is a ready solution and has been adopted by latest architectures (that is, KabyLake, CoffeeLake) as well. Intel's MPX provides hardware assisted memory protection by making modifications at each level of the hardware stack as shown in Figure 3.1.

- **Hardware level.** The changes are done by addition of seven new instructions that check memory safety. Also, a set of 128-bit registers are added that hold pointer metadata. The bounds made by instruction **bndmk** are stored in the bound(bnd) registers. Instructions, bndcl and bndcu/bndcn have the task of comparing the pointer value against bounds information stored in bnd. Instruction bndmov moves the pair of bounds between registers and also to/from memory. When bnd registers are not sufficient, the pointer bounds have to be stored in Bounds table. This store and load is done Instructions, bndldx and bndstx. The instructions bndldx and bndstx are used to load/store from/to a memory location using the pointer address. Figure 3.2 shows how a pointer address is stored.

  Storing pointer address to BTs table requires two level-address translation which is an expensive process. In the first level, the corresponding BD entry has to be loaded. For that, CPU extracts offset from BD entry, shifts it, loads the base address of BD from BNDCFGx register and finally, adds the base and the offset. This resultant address provides the BD entry to be loaded.
  In the second stage, the CPU extracts the offset of BT entry from the pointer address and shifts it. It then shifts the loaded entry—which corresponds to the base of BT. Then it adds the base to the offset. The resulting address provides the BT entry to be loaded [39].

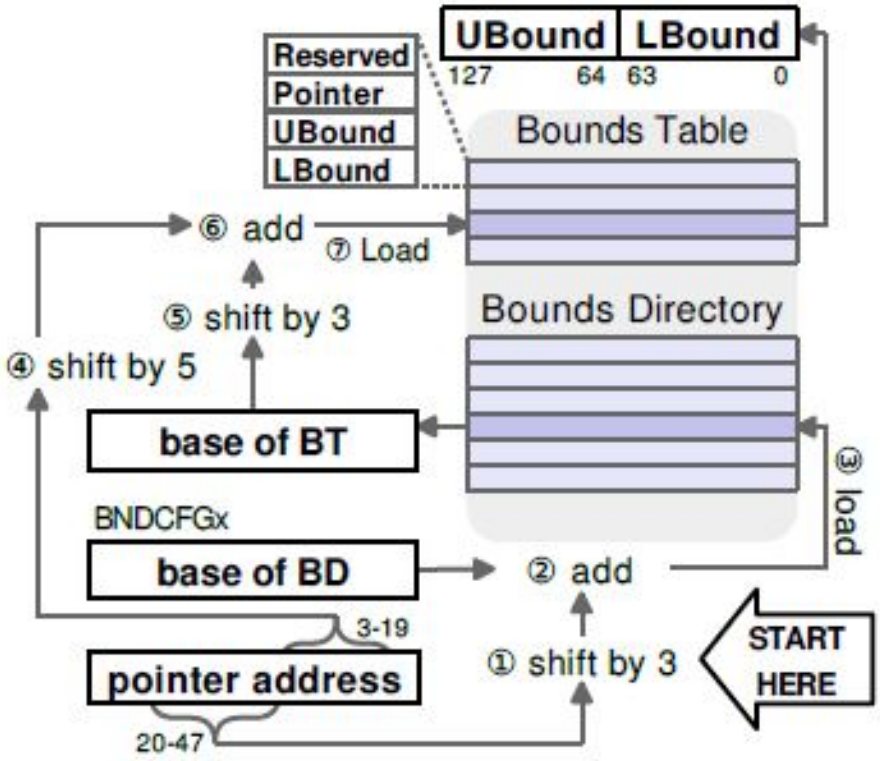


Figure 3.2: Two Level Address Translation [39]

- **OS level.** A new #BR exception handler is added that handles bounds violations and manages the Bounds Tables. If the CPU detects that the referenced pointer is outside of its checked bounds, it raises the BR exception and traps it to the kernel (in case of Linux). The kernel extracts the violating address and the violated bounds from the instruction. The extracted information is stored in the siginfo structure. The application is now informed that a violation has happened by SIGSEGV. The responsibility now lies with the application developer whether to handle this issue or ignore it.

  The OS also does the Bounds table management i.e. creates and deletes them. The Bounds Directory is allocated at runtime but BTs have to be created dynamically on demand. Bounds Table(BT) is required for two levels of bounds address translation.

OS creates BTs dynamically on-demand. [39].

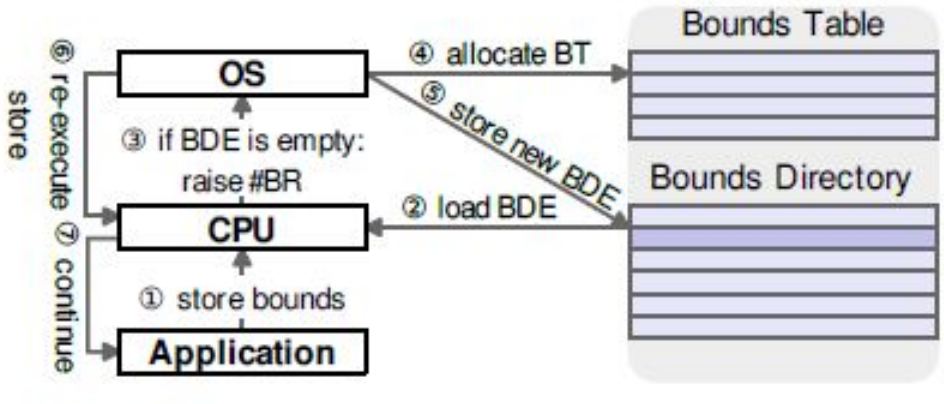


Figure 3.3: Creation of Bounds table and storage of an entry [39]

Figure 3.3 shows how BTs are created and entries are stored in them. Every time an application tries to store pointer bounds ,the CPU first verifies if it is valid. For that, CPU loads the corresponding entry from the Bounds Directory(BD) and verifies. The #BR exception is raised if it is invalid (that is, an empty BD entry). Then, the kernel has to create two entities: a new BT (to store new bounds) and Bounds directory entry (to store the new address). After this, the normal execution is continued in the user space. OS also has to free these tables as a part of ”garbage removal” of memory object that is freed. The OS frees the unused BTs and removes their entries in the BD.

- **Compiler level.** New Intel MPX transformation passes are added to insert MPX instructions. These passes are the one which impose bound checking along with their creation, propagation and storage. Also, runtime support is added. The new versions of GCC (5.0+) and ICC (15.0+) are able to support MPX. Both of these compilers introduced a new compiler pass called Pointer(s) Checker.

  Runtime support was provided with MPX specific libraries. The GCC has libmpx and libmpxwrappers whereas ICC has libchkp. These libraries enable MPX support in hardware and OS.

At the application level, the MPX support is not full-fledged. It is observed to not support several widely-used C programming idioms. The support is not there because of some design and implementation choices. Therefore, some applications need so much manual changes to enable MPX that it is not advisable to use it all. Furthermore, there is no support for the multi-threaded programs.

Compared to previous work, MPX has several advantages. Since MPX is a hardware-assisted protection, it fares well compared to software-only approaches. Because of the introduction of bounds register to so the bound checking, the pressure is taken away on the GPR file.
Second, Intel's MPX modifies the x86-64 calling convention. Before a function call, the bounds associated with pointer arguments are put in registers bnd0–bnd3. Before the return from the function, the bounds associated with the pointer return value are put in bnd0. Whereas, software-based rely on function cloning when a set of function arguments is extended to include pointer bounds. This causes complications with interoperability with legacy uninstrumented libraries because of cumbersome caller/callee code.
Finally, in software-based approaches, "compare and branch" instruction consumes one cycle and also, involves the branch predictor. However in MPX, the bndcl and bndcu instructions act as economical substitutes for these expensive instructions.

Despite these advantages, primary being the spatial memory security, there are performance overheads associated with MPX. While it provides significant improvements over other approaches, the high overheads discourage its adoption on a wide scale.

3.2 Performance Analysis of MPX in Prior Work

Oleksenko et. al [39] did an exhaustive empirical study of MPX to understand its advantages, performance overheads being one of the dimensions of their study. They reported

that MPX's performance overheads are as high as 50% on an average. The authors selected PARSEC 3.0 [26], Phoenix 2.0 [41] and SPEC CPU 2006 [45] benchmarks for their performance study. They studied two implementations of MPX (that is, GCC and ICC) and compared it with SAFECode [27], SoftBound [34] and AddressSanitizer [42], which are software-based approaches. For the scope of our research, we will be listing performance of SPEC benchmarks out of all the major findings of this paper.

- **Runtime overhead.** It was observed that ICC version performed better than GCC. But it was less usable as compared to GCC, it compiled fewer benchmarks without error. AddressSanitizer, a software based approach, performed as good as MPX-ICC and better than MPX-GCC. This results reveals that the complicated design of MPX offset the hardware-assisted performance improvements of MPX. Figure 3.4 shows runtime overhead in different SPEC CPU 2006 benchmarks.

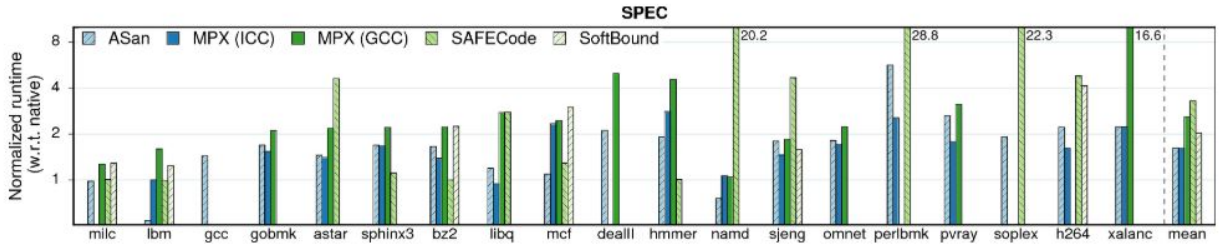


Figure 3.4: Runtime overheads in SPEC benchmarks [2]

- **Instruction overhead.** The authors pointed out a strong correlation between performance overheads and instruction overheads. They also found that ICC had low instruction overhead due to its HW assistance. Figure 3.5 shows the instructions overhead in different SPEC CPU 2006 benchmarks.

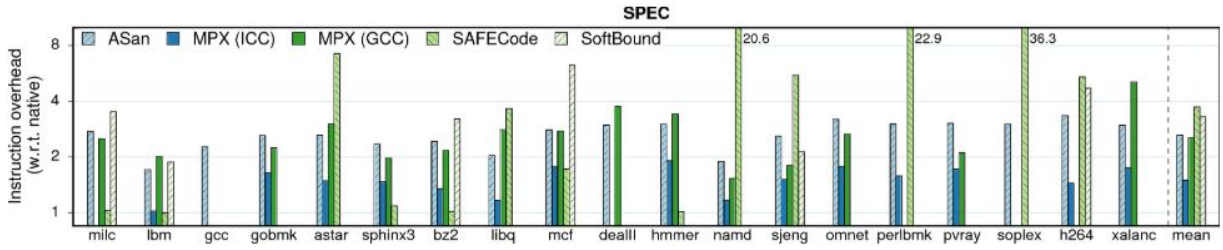


Figure 3.5: Instruction Overhead in SPEC benchmarks [2]

- **IPC.** It was found that the MPX does not increase Instructions per cycle. However, IPC increased significantly in software-based approaches, partially hiding the performance overheads. Some programs were found to have very low IPC, indicating that they are memory intensive and therefore, their CPU utilization would reveal more findings. Figure 3.6 shows the IPC overheads in different SPEC CPU 2006 benchmarks.

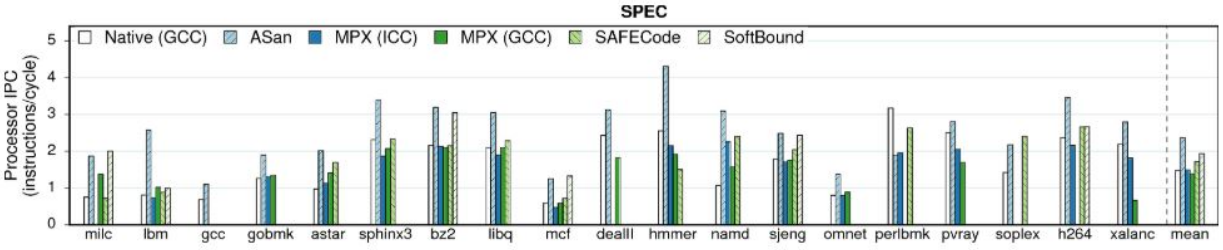


Figure 3.6: IPC Overheads in SPEC benchmarks [2]

- **Cache Utilization.** It was found that most of the programs had good cache locality with most of the accesses in L1 cache. As reported in previous point, programs having low IPC had high cache misses. Figure 3.7 shows the cache utilization in different SPEC CPU 2006 benchmarks.

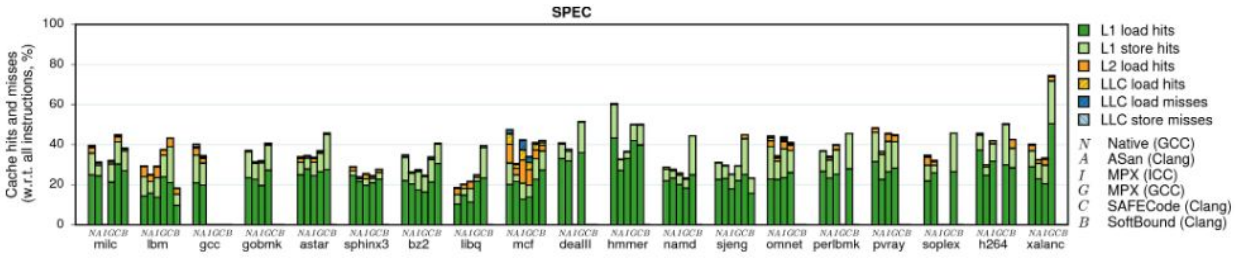


Figure 3.7: Cache Utilization [2]

- **Intel MPX instructions.** As expected, the bounds checking instructions bndcl and bndcu form the major chunk of all the MPX instructions. There is a strong correlation between proportion of expensive instructions: bndldx and bndstx and performance overheads. Figure 3.8 shows the MPX instructions' distribution SPEC CPU 2006 benchmarks.

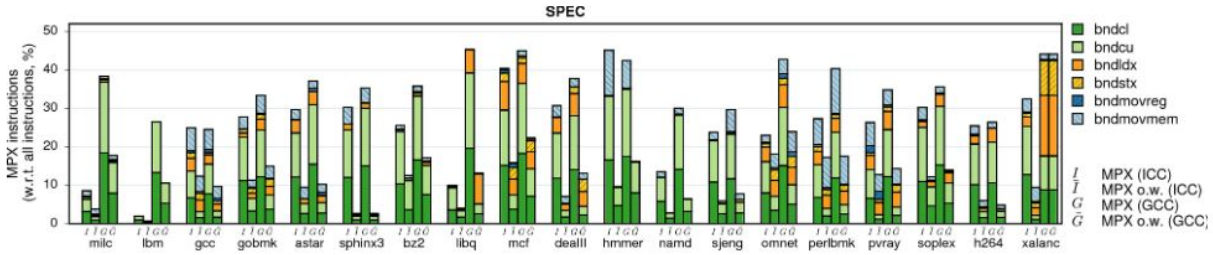


Figure 3.8: MPX instructions' distribution [2]

- **Memory Consumption.** Compared to AddressSanitizer, both MPX-ICC and MPX-GCC reported significantly lower memory consumption as it only needs space for pointer-related metadata. The presence of intermediate Bounds Directory means longer access time. This hides the lower memory consumption.

3.3 Motivation

The way MPX support is enabled in the hardware stack, that results in poor cache locality. As we see in section 3.1, the bounds storage and loading requires two level address translation. For instance, a pointer bounds loading requires loading corresponding entries from both bounds directory and bounds table. However, these two loadings are non-contiguous and therefore, the cache locality worsens. It is also well known that the cache misses negatively impact the runtime performance of programs.

Therefore, we were motivated to focus on increase in cache misses when MPX support is enabled. We can expect that if we see performance degradation, we should see an increase in cache misses in the MPX version compared to the native version. Therefore, the major research question that we want to answer is: **If there is a correlation between increase in cache misses and degraded runtime performance in applications compiled with MPX?**

We performed the evaluations for two different input sizes: train(medium) and ref(large). We quantified the average performance overhead for different input sizes. We did our analysis for different workload sizes based on our hypothesis that as the input sizes increase, it is logical to assume that more pointer accesses might happen. Also, more MPX instructions would be added that enforce spatial security. As the pointer accesses increase, more bounds metadata would need to be stored in bounds table. Hence, we should expect to make two observations. First, we should see instructions overheads increase as the corresponding input sizes increases. Second, there should be increase in cache misses as we go from the medium input size(train) to the large size(ref) for a benchmark.

3.4 Contributions

We make the following contributions in our research:

- We find the instructions overhead in an attempt to explain the runtime overheads trends as the input size increases. In prior work by Oleksenko et al., the impact of instruction overheads has been studied for only the ref size. We explore this impact for one more input size-train.
- We find cache misses and attempt to find a correlation between increase in cache misses and performance degradation after enabling the MPX support. In prior research, cache misses were found but no comparison has been done with the native (non-MPX) versions. We have done a detailed analysis to see when cache misses are increased wrt the native versions, if that is correlated with the runtime overheads. We calculated correlation coefficients to understand the relationship between these two.
- Along with cache misses, we also found TLB misses for all the benchmark programs. Unlike the prior work that just shows the % TLB misses, we find and compare the increase in TLB misses in MPX version with misses in the native versions.
- The prior work is restricted to only the 2006 version of SPEC CPU benchmarks. In our research, we also included 7 benchmarks from the latest 2017 version. These benchmarks are considered to be more stable and have more kilo lines of code.
- We have done our analysis with ICC compiler on windows platform. Windows became our choice of platform as it was the only OS where we could make the ICC compiler work with MPX. In the prior work, the evaluations are done and reported with only Linux platform. To the best of our knowledge, our work on Windows is one of the firsts.

Chapter 4

Evaluation

In this section, we discuss the experimental setup and the methodology that we used for our evaluations. We will discuss the benchmarks and setup for the performance overhead evaluation. Then, we discuss the procedure for collecting and processing the hardware statistics. These statistics were found to find cache misses per kilo instructions, TLB misses kilo instructions and instructions overheads.

4.1 Methodology

4.1.1 Workloads

We chose SPEC benchmarks: SPEC CPU2006 [5, 45] and CPU2017 [6, 23] as the benchmark suites. These suites make a good choice for benchmarking as they are designed to provide performance measurements that can be used to compare compute-intensive workloads on different computer systems. The workloads in these suites are developed from real user applications. Table 4.1 shows the workloads selected by us for performance overhead evaluation. As the table shows, we have chosen 15 benchmarks of both Floating Point and Integer type.

Table 4.1: List of SPEC CPU benchmarks used.

Workload	**Type**	**Language**	**Category**
2017:LBM_S	Speed, Floating Point	C	Fluid Dynamics
2017:EXCHANGE2_S	Speed, Integer	Fortran	Artificial Intelligence: recursive solution generator (Sudoku)
2017:BWAVES_S	Speed, Floating Point	Fortran	Explosion modeling
2017:FOTONIKS3D_S	Speed, Floating Point	Fortran	Computational Electromagnetics
2017:ROMS_S	Speed, Floating Point	Fortran	Regional ocean modeling
2017:MCF_S	Speed, Integer	C	Route planning
2017:LEELA_S	Speed, Integer	C++	Artificial Intelligence: Monte Carlo tree search (Go)
2006:BZIP	Integer	ANSI C	Compression
2006:HMMER	Integer	C	Search Gene Sequence
2006:SJENG	Integer	ANSI C	Artificial Intelligence
2006:LIBQUANTUM	Integer	C99	Physics/Quantum Computing
2006:ASTAR	Integer	C++	Path-finding Algorithms
2006:MILC	Floating Point	C	Physics: Quantum Chromodynamics
2006:NAMD	Floating Point	C++	Biology / Molecular Dynamics
2006:POVRAY	Floating Point	ISO C++	Computer Visualization

4.1.2 Processor

We conducted our experiments on Intel Core i7 8700K processor on Windows OS. Table 4.2 shows the processor micro-architecture.

Table 4.2: Processor microarchitecture.

Core	6 cores, 12 Threads, 3.7 GHz
L1 d-cache, i-cache	8 way, 32kB
L2 cache	8-way, 256kB
L3 cache	8-way, 12MB

We changed several parameters for the processor, before compiling and running the benchmarks. These settings are specific to a Windows Operating system. Following are the settings we did prior to running the experiments. :

Affinity. Our primary concern is ensuring benchmark stability. That is, if we run our benchmarks repeatedly, we should have consistent numbers. In a multi-core processor, the operating system can schedule a process or a thread on any CPU core at any time. Therefore, for every different run, the process/thread can move to a different CPU core. This affects benchmark stability as the new core does not have the same data in its L1 cache as the old core. And when the process/thread is scheduled back to the old core, that core's cache may already have been polluted by other work that was scheduled on that core [32]. Hence, we have to make sure that our process is restricted to a certain core during the entire run. This process is called setting the affinity of the process.

We achieved this by writing a batch script that runs to set the affinity in such a way that the benchmark program runs on only CPU0. This means that the experiments run on only one core. Listing 4.1 shows the batch script that sets affinity:

Listing 4.1: Setting up affinity of a process

```
1    @echo off
2    start "" /wait /affinity 1 "allRuns.bat"
```

Here, batch script allRun.bat is started with affinity set to 1. An affinity 1 corresponds to CPU0 (that is, on just one core).

Priority. A benchmark process running at normal priority can get preempted/interrupted by another process which has higher priority. This interrupting process ends up consuming the maximum amount of CPU time. To avoid this problem, we assign a higher priority to the benchmark programs.

In windows, we have six levels of priorities: idle(64), below normal(16384), normal(32), above normal(32768), high priority(128) and real time (256). We chose "above normal" priority for our runs. Setting the priority as highest, which is, "real time" is not recommended because when set to "real time" it leaves no cycles for any other process [24]. Then we moved to second highest level- "high". We started off with setting the priority as "high" but also caused our system to freeze. Therefore, "above normal" became the best choice for us. Most of processes that were running had "normal" priority and therefore, our process is still set at a higher priority with this choice. We wrote a script that runs in the background all the time. This script keeps on looking for our benchmark binary and as soon as it finds it, it sets its priority to "above normal" using wmic command[18]. Listing 4.2 shows the batch script code that sets the priority of the binary.

Listing 4.2: Setting priority of a process

```
1    @echo off
2    :loop
3    tasklist | findstr /i "brun.exe"
4    if "%ERRORLEVEL%"=="0" (
```

```
5       wmic process where "name like '%%brun.exe%%'" CALL setpriority
            32768
6   )
7   if "%ERRORLEVEL%"=="1" (
8       goto:loop
9   )
```

Disabled Turbo-boost. Modern processors employ Turbo-boost that changes their frequencies to maximize performance when needed [10]. This means the frequency keeps changing according to the workload that is run. The frequency can also change because of increased temperature and number of active cores [46]. For the purpose of our benchmarking where we want to have repeated evaluations and reproducible results, random frequency changes due to turbo-boosting can introduce noises and impact our results. The result could be different numbers for every run and therefore, we will not have stability. We disabled turbo-boosting by changing the advanced power settings [9, 21]. For the SPEC benchmark runs, we set both the minimum and maximum power set at 75%. This set our CPU frequency at a conservative frequency of around 2.8 GHz with very small variations throughout the runs. For some benchmarks with input size train, we had to set at even lower value of 25% which resulted in an approximate frequency value of 0.89 GHz. Figure 4.1 shows the power settings we did to disable turbo-boosting.

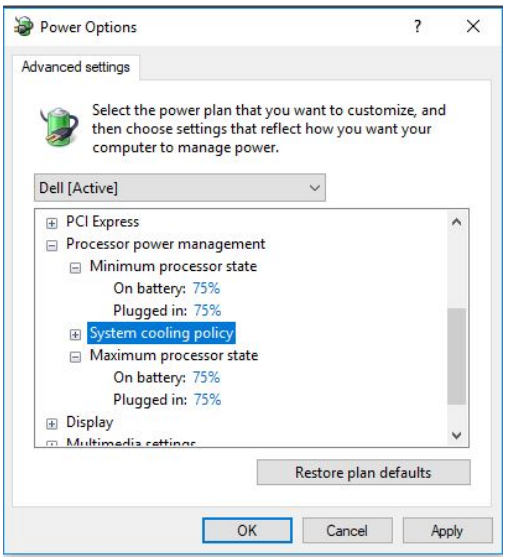


Figure 4.1: Change in processor states to disable Turboboosting

Settings like affinity and priority can be set manually with task manager but that would mean keeping track of each run and then, setting them in the task Manager. Also, smaller benchmarks can finish running in few seconds, leaving no time to manually do these settings for them. Therefore, we wrote scripts to automate this process of setting affinity and priority of the running benchmark process.

4.1.3 Enabling MPX support in applications

To enable MPX support in applications running on windows platform, we executed a series of steps. First, we downloaded and installed an MPX driver for the Windows platform. MPX support can be simply enabled by adding Pointer-Check flags when compiling the programs. Listing 4.3 shows the flags that we used to enable MPX in windows:

Listing 4.3: Flags added to enable MPX in Windows

```
-Qcheck-pointers-mpx=rw -Qcheck-pointers-undimensioned-
```

To verify that the spatial memory security has been imposed by MPX, we tested the same array out-of-bounds bounds program as in section 2.1 in which we observed an array with 10 elements. Initially, we compiled the binary without any flags and ran it. During compilation, it showed a garbage value which was lying out of bounds. On running it, we see the garbage value for arr[10] as shown in Listing 4.4.

Listing 4.4: Output of binary compiled without MPX support

```
arr [0] is 1
arr[10] is 0
```

Then, we compiled the program with appropriate MPX flags as shown in listing 4.5.

Listing 4.5: Flags added during compilation flags to enable MPX on Windows

```
icl -Qcheck-pointers-mpx=rw -Qcheck-pointers-undimensioned- -o MPX.
    exe mpxtest.c
```

After running the MPX.exe binary, we could verify that MPX support has been enabled.

Listing 4.6: Output of binary compiled with MPX flags

```
arr [0] is 1
```

As seen in Listing 4.6, the garbage values are not printed as before. This ensures that bounds-checking were in place.

Enabling MPX support in SPEC CPU benchmark applications

To automate the build and run process of the benchmarks, we created a configuration(.cfg) file. This is the file where all the variables and configurations were set, primarily the ICC compiler (icl in windows), pointer-check flags to enable MPX. Listing 4.7 shows the ICC(icl)

compiler and MPX flags that we set in the SPEC config file.

Listing 4.7: Setting variables in config file to enable MPX support in benchmark applications

```
CC = icl -Qvc14.1 -Qm64 -Qstd=c99 -TC -Qcheck-pointers-mpx=rw -Qcheck-
    pointers-undimensioned-
CXX = icl -Qvc14.1 -Qm64 -Qstd=c++11 -TP -Qcheck-pointers-mpx=rw -
    Qcheck-pointers-undimensioned-
```

Here, CC and CXX variables indicate how to invoke the C and C++ compilers respectively. Flags -Qcheck-pointers-mpx=rw and -Qcheck-pointers-undimensioned- are the ones that we discussed in Listing 4.5 that help enable MPX support in applications. The "Qcheck-pointers-mpx=rw" flag checks bounds for reads and writes through pointers. The "-Qcheck-pointers-undimensioned-" flag enables "Bounds checking" for memory access through arrays that are declared without dimensions.

The ICC compiler is not available by default in the command line. Therefore, compiling a program with ICC will throw errors. In the next section 4.1.4, we discuss how we get access to ICC compiler to resolve this issue.

4.1.4 Intel Parallel Studio XE

Intel's Parallel Studio XE is an ensemble of different applications, libraries, compilers and tools that help in software development, code analysis and other computing tasks. It provides access to the ICC compiler(from Visual Studio) and profiling tools like VTune Amplifier that we used for collecting hardware statistics. We downloaded the academic version of Parallel Studio . The Visual Studio needs to be pre-installed before installing the Parallel Studio to integrate them with each other.

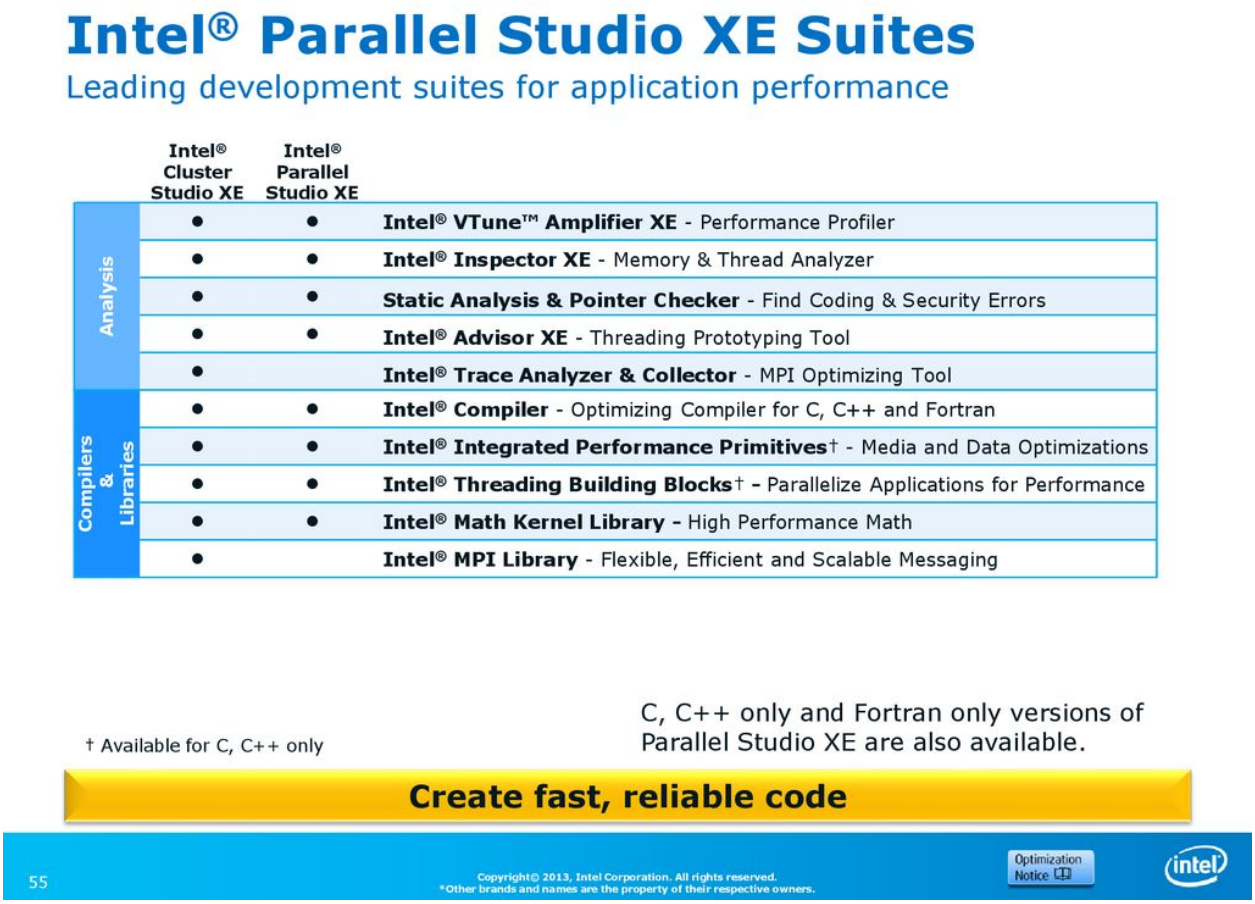


Figure 4.2: Intel's Parallel Studio XE [8]

Visual Studio

We navigated to Parallel Studio XE folder and selected compile command line for Intel64 as our terminal under that. This opens up the Visual Studio Developer Command Prompt with environment initialized for x64 in our case. With this command line, we get access to Intel's ICC compiler corresponding to the version of Parallel Studio XE that is downloaded. For our Parallel Studio version 19 (2019), we had ICC compiler version 19.0.

Intel's VTune Amplifier

The tools that can help in taking hardware measurements are limited on Windows. We found that the Intel's VTune Amplifier, also shipped with Parallel Studio XE, is one of the most powerful tools to profile the benchmarks and collect important hardware statistics like number of instructions, L1/L2/LLC cache misses and hits among others. Following are the VTune Amplifier's performance analysis configuration:

- **Microarchitecture analysis group.**
  - **Microarchitecture Exploration.** It is used to identify the CPU pipeline stage (front-end, back-end, etc.) and hardware units responsible for hardware bottlenecks.
  - **Memory Access.** It can be used to identify which memory hierarchy level is impacting our memory-bound app. It reviews the CPU cache and main memory usage to identify memory access.
- **Hotspots analysis group.** This group has two configurations:
  - **Hotspots.** It is used to analyze call paths to find spots where the code spends more execution time. Finding hotspots can help developers to tune their algorithms for optimal performance.
  - **Memory Consumption.** Used to analyze memory consumption by the application, its distinct memory objects, and their allocation stacks.
- **Parallelism group.**
  - **Threading.** It is used to visualize thread parallelism on available cores, locating causes of low concurrency, and identifying serial bottlenecks in the code.

  - **HPC Performance Characterization.** It helps in understanding how your compute-intensive OpenMP* or MPI app is using the CPU, memory, and floating point unit (FPU) resources.

- **Platform analysis group.** This helps in analyzing specific use cases like GPU, I/O, IRQ analysis and so on.

These analysis groups help in profiling and performance analysis. However, our measurements involved cache MPKIs. That required us to find raw number of cache misses and instructions count. Table 4.3 lists the events for which we found the raw numbers for the benchmark run. We then found MPKIs using these numbers. Since none of these metrics helped find all these numbers, we utilized another group: Custom Analysis. We used the command line interface since that helped us automate the process of statistics collection [14]. We simply used a Windows command prompt and navigated to the VTune Amplifier (version) folder under IntelSWTools. These IntelSWTools come along with the Parallel Studio XE. Now, we set up the tool by executing the amplxe-vars.bat file A response shown in listing 4.8 shows that we are ready to use VTune Amplifier after running the amplxe-vars.bat file which sets the environment for running this tool.

Listing 4.8: Starting VTune Amplifier with amplxe-vars.bat file

```
Copyright (C) 2009-2018 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier 2019 (build 579888)
```

Command tool "amplxe-cl" was used for collection of statistics [17]. When used with collect-with, it enables custom hardware event-based sampling. We used "runsa" data collector with "collect-with" action for our custom analysis. The hardware event-based sampling collector of the VTune Amplifier profiles the application using the counter overflow feature of the Performance Monitoring Unit (PMU) [15]. Listing 4.9 shows the generic format of amplxe-cl

command with the runsa collector:

Listing 4.9: amplxe-cl command with runsa collector

```
$ amplxe-cl -collect-with runsa -knob event-config=CPU_CLK_UNHALTED
    .REF,INST_RETIRED.ANY -- /home/test/sample
```

For the hardware event based data collection, we worked with the default sampling interval of 1 ms, which was frequent enough to collect samples. This value can lie between 0.01 ms to 1000 ms, depending on how many samples we want to collect. For a high value of sampling interval, the number of samples would decrease and that will also result in smaller size of report that is generated. This high value helps collect values for applications that have a high runtime and generate large size reports.

In our custom analysis, we found several statistics that are described in Table 4.3. VTune Amplifier finds the events with their microarchitectural names and these are what we list in the table.

4.1.5 Performance analysis of SPEC benchmarks compiled with MPX

Now that we have discussed all the tools and compilers that we used in our methodology, we will now discuss the performance analysis in detail. The first evaluation is focused on finding runtime overheads when MPX is enabled on Windows. As discussed previously, we compiled these benchmarks with Intel's ICC compiler. Table 4.1 lists the benchmarks we considered. As it is known that MPX support on application-level is not full-fledged, some applications failed to compile with MPX.

We created several scripts to automate the benchmark run. Since we ran for two sizes:train

Table 4.3: List of hardware event for which we collected statistics using VTune Amplifier.

Event Name	**Meaning**
INST_RETIRED.PREC_DIST	INST_RETIRED shows how many instructions were completely executed between two clocktick event samples [12]. PRED_DIST is a version of INST_RETIRED that allows for a more unbiased distribution of samples across instructions retired. It utilizes the Precise Distribution of Instructions Retired (PDIR) feature to mitigate some bias in how retired instructions get sampled.
MEM_LOAD_RETIRED.L1_MISS	Counts retired load instructions with at least one uop that missed in the L1 cache [4].
MEM_LOAD_RETIRED.L2_MISS	Counts retired load instructions with at least one uop that missed in the L2 cache [4].
MEM_LOAD_RETIRED.L3_MISS	Counts retired load instructions with at least one uop that missed in the L3 cache [4].
DTLB_LOAD_MISSES.WALK_COMPLETED	Counts demand data loads that caused a completed page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels [4].

and ref, we carefully created scripts that run several benchmarks in sequence for each size. We keep the priority-setting script running in the background. First, we started the native run with no MPX enabled. After the runs were completed for each size, we collected the reports generated by SPEC to gather the runtime numbers. After that, we did a second run for benchmarks compiled with MPX flags for all the three sizes. We collected the runtimes for the second run as well.

After that, we divided the runtime of MPX enabled run with that of native run to get the runtime overheads for all the sizes using this formula:

$$\text{Runtime overhead} = \frac{\text{Runtime(MPX)}}{\text{Runtime (native)}} \times 100$$

We then created bar graphs to observe how runtime overheads were incurred for the 15 benchmarks in our dataset. We also created individual bar graphs for each size which we used for finding correlation with the cache MPKIs calculated in section 4.1.6.

4.1.6 Hardware events collection using VTune Amplifier

Once again, we created a batch script to automate the process of collecting the statistics listed in Table 4.3. Our script runs in the background and keeps on looking for a benchmark process to appear and run. We provided a unique tag to the benchmarks in the config file. For SPEC 2006 runs, all the benchmark binaries get this name "benchmark_name.brun.exe" where "brun" is the tag we gave. This "brun" is unique and no other windows process has this same name. As soon as the benchmark runs, the VTune script "latches" on this binary and starts collecting statistics. Given that the name of benchmarks vary, we used wildcards to match the latter part,"brun.exe", of the running process to identify and latch amplxe-cl command to it. Similarly in SPEC CPU 2017, we used "myTest" as an identifier for the benchmark processes among all the windows processes.

We ensured that when running multiple benchmarks, we run them in sequence and not in parallel. This was because during parallel runs of 2 or more benchmarks, the VTune amplifier script would give error as it can latch to only one benchmark's .exe file.

After the collection is complete, the report is generated in a format which can only be accessed via graphical interface of VTune Amplifier. Figure 4.3 shows the hardware events collected by our VTune Amplifier using amplxe-cl with runsa collector. Along with this custom analysis, the report also shows elapsed time for the collection and the platform information like CPU microarchitecture etc.

Figure 4.4 shows a high level overview of our methodology.

Hardware Events

Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample
CPU_CLK_UNHALTED.REF_TSC	22,512,033,768	11,256	2000003
INST_RETIRED.PREC_DIST	11,196,016,794	1,866	2000003
MEM_INST_RETIRED.ALL_LOADS_PS	2,778,004,167	463	2000003
MEM_INST_RETIRED.ALL_STORES	948,001,422	158	2000003
MEM_LOAD_RETIRED.FB_HIT	65,704,599	219	100007
MEM_LOAD_RETIRED.L1_HIT	2,412,003,618	402	2000003
MEM_LOAD_RETIRED.L1_MISS	78,902,367	263	100003
MEM_LOAD_RETIRED.L2_HIT	17,700,531	59	100003
MEM_LOAD_RETIRED.L2_MISS	60,925,578	406	50021
MEM_LOAD_RETIRED.L3_HIT	60,325,326	402	50021
MEM_LOAD_RETIRED.L3_MISS	300,021	1	100007

*N/A is applied to non-summable metrics.

Figure 4.3: Hardware events collected by VTune Amplifier

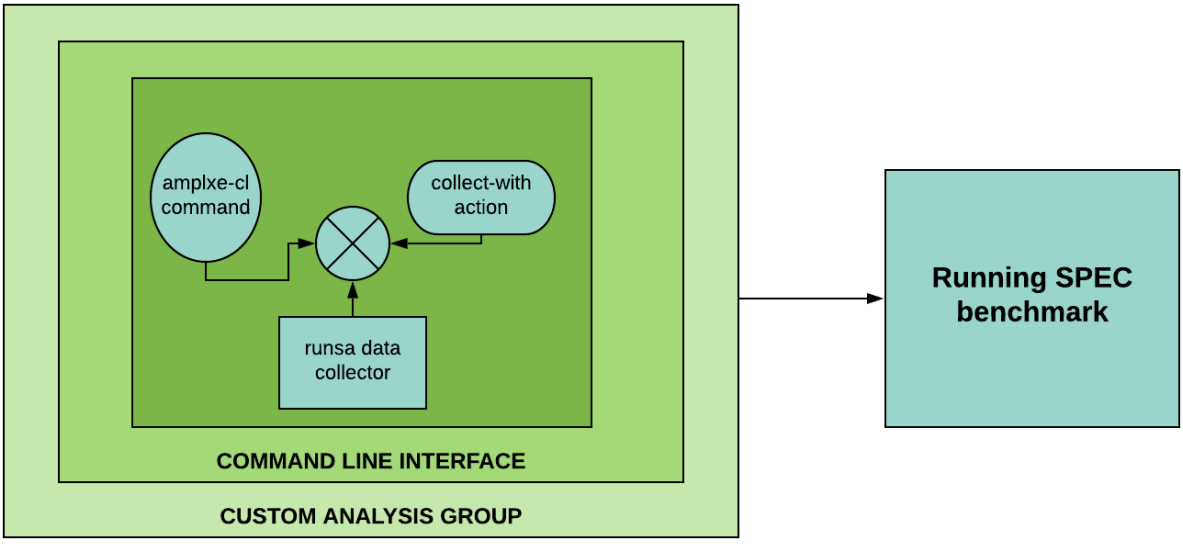


Figure 4.4: Custom Analysis using VTune Amplifier

When we run the benchmarks, we run for both native and MPX-enabled versions. In both the cases, we collect the hardware event counts to calculate L1, L2, L3(LLC), TLB cache misses and number of instructions for both the sizes of the benchmarks. Our main aim is

to explain the performance degradation that occurs because of MPX. To understand cache utilization, we find cache Misses per Kilo Instructions (MPKIs) for both MPX-enabled and native benchmarks using this formulae:

$$\text{Cache MPKI(native)} = \frac{\text{Number of cache misses(native)}}{\text{Total Number of Instructions(native)}} \times 1000$$

$$\text{Cache MPKI(MPX)} = \frac{\text{Number of cache misses(MPX)}}{\text{Total Number of Instructions(native)}} \times 1000$$

Similarly, we found the TLB misses per kilo instructions using the similar formulae:

$$\text{TLB MPKI(native)} = \frac{\text{Number of TLB misses(native)}}{\text{Total Number of Instructions(native)}} \times 1000$$

$$\text{TLB MPKI(MPX)} = \frac{\text{Number of TLB misses(MPX)}}{\text{Total Number of Instructions(native)}} \times 1000$$

These cache misses were plotted on bar charts. These bar charts helped us in identifying the relationship between increased cache misses and runtime overheads that we do in Chapter 5.

Along with these misses, we also find instructions overheads using this formula:

$$\text{TLB MPKI(MPX)} = \frac{\text{Number of Instructions (MPX)}}{\text{Number of Instructions(native)}} \times 100$$

Our whole methodology can be summed up with figure 4.5.

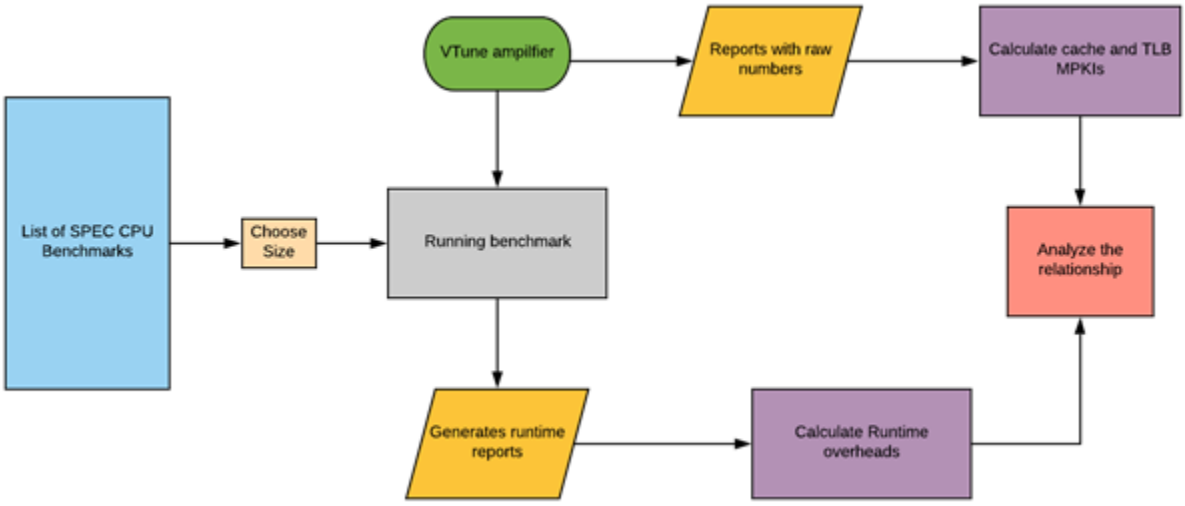


Figure 4.5: Overview of Methodology

4.2 Analysis of correlation between two statistics

We start with qualitative analysis of our results to gain important insights. Along with that, we also use a statistical measure, correlation factor, to see the relationships between different statistics that we measure. We find the Pearson coefficient [1] to quantify the relationships between different overheads.

$$\rho = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}$$

Here ρ means the correlation coefficient, cov(X,Y) means the covariance between variables X and Y and σ_X and σ_Y correspond to the variance of a variable X and Y respectively.

Based on the following values of ρ as shown in Table 4.4, we categorised relationships between two variables:

Table 4.4: Correlation coefficient values and their meanings.

Value of ρ	**Meaning**
+.70 or higher	Very strong positive relationship.
+.40 to +.69	Strong positive correlation.
+.30 to +.39	Moderate positive relationship.
+.20 to +.29	weak positive relationship.
+.01 to +.19	No or negligible relationship.
0	No relationship [zero correlation.
-.01 to -.19	No or negligible relationship.
-.20 to -.29	weak negative relationship.
-.30 to -.39	Moderate negative relationship.
-.40 to -.69	Strong negative relationship.
-.70 or higher	Very strong negative relationship.

Using this scale, we find correlations between runtime overheads and instructions, cache misses and TLB misses overheads.

Chapter 5

Results and Discussion

In this chapter, we present our results and discuss our learnings from them. Section 5.1 shows the results of runtime analysis where we present the runtime overheads of the benchmarks. We start with analysing the relationship between runtime overheads and instructions overhead in section 5.2. In section 5.3, we examine the relationship between cache misses and runtime overheads for both the input sizes. In section 5.4, we analyze the impact of TLB misses on benchmarks' performance. And finally in section 5.5, we find correlation coefficients between runtime overheads and different overheads-instructions, cache misses and TLB misses.

5.1 Runtime Analysis

As explained in section 4.1.5, we calculated runtime overheads for the two input sizes for the SPEC CPU benchmarks. Figure 5.1 shows the the runtime overheads in a bar graph. As seen in the figure, all the benchmarks have overheads across both the sizes except five SPEC CPU 2017 benchmarks.

For each benchmark program, the overheads are with respect to the native runs. Each benchmark is independent of each other. We made some important observations from Figure 5.1:

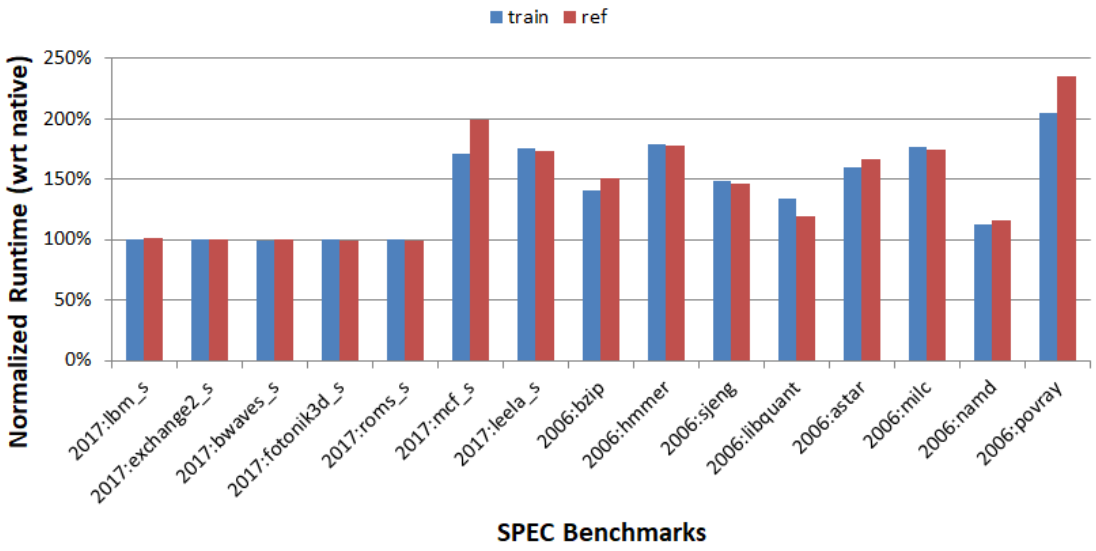


Figure 5.1: Runtime overheads in input size ref and train

- The 15 benchmarks have an average overhead of around 145% for size ref. Whereas, the average overhead for size train is around 140%.

- We had expected that the runtime overheads would increase as the input size increases. That is, moving from left to right for a benchmark in the bar graph, we should see an increasing trend. Our reasoning behind this was as the input sizes increase, we should see an increase in operations related to pointer accesses. As these operations increase because of size increase, we should expect more non-contiguous pointer loading in the bounds directory and tables. With that cache misses should increase and therefore, runtimes should increase. However, except for mcf, bzip, astar and povray, we observe no such trend in the benchmarks.

- We observe that for 5 out of 7 SPEC CPU 2017, there are negligible runtime overheads with MPX for any input size. These 5 benchmarks are lbm, exchange2, bwaves, fotonik3d and roms. It seems that even after compiling with MPX, there might not be any increase in the pointer-bounds check operations for these benchmarks. Even if

there are any such operations, they might be handled by the bounds registers. Therefore, there might not be any increase in cache misses and hence, no runtime overheads. Also, for benchmark 2006:namd, although there is some overhead (10-15%) but it is still small compared to the other benchmarks. Also, the overheads remain almost the same across both the sizes(with standard deviation less than 0.5 across the three sizes).

- Benchmarks 2017:mcf, 2017:leela, 2006:astar, 2006:povray and 2006:milc have very high overheads. The overheads range from 166% to 235% for input size ref.

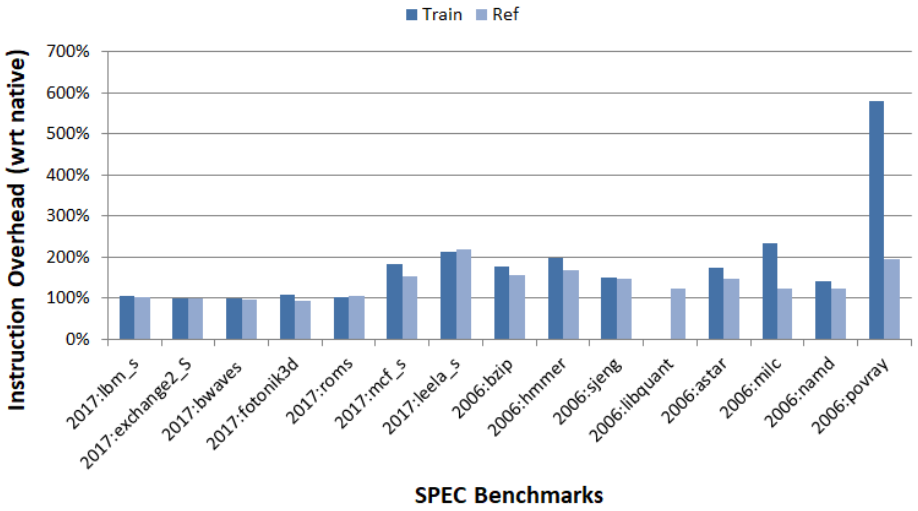
- For benchmarks 2017:leela, 2006:bzip and 2006:sjeng also have high runtime overheads. These overheads do not change much across the different input sizes as it can be seen with the standard deviations which are less or equal than 5.

5.2 Relationship between Instructions overheads and Runtime Overheads

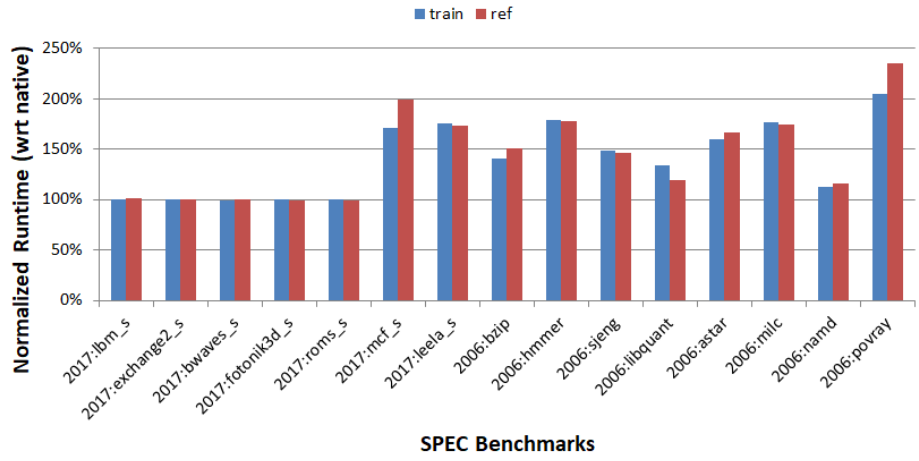
To explain the runtime overhead trends observed in 5.1, we first started by looking at instructions overheads. We speculate that the instruction overhead trends should follow the runtime overheads trends. That is, if we observe a benchmark having its runtime overhead increase as input size is increased, the instruction overheads should also increase in a similar way. Figure 5.2(a) shows the instructions overheads for the benchmarks across all the three sizes. Figure 5.2(b) shows the runtime overheads.

Looking at Figure 5.2, we make some important observations. We see that for 5 SPEC CPU 2017 benchmarks- lbm, exchange2, bwaves, fotonik3d and roms, we see little to no instructions overheads for any size. When compared with their corresponding runtimes in 5.2(b), we do not see any runtime overheads either. It seems that that when compiled with

MPX, no additional MPX instructions were added. Therefore, both the MPX and native versions would have same number of instructions executed for their respective sizes.



(a)



(b)

Figure 5.2: (a) Instructions overheads across all the sizes (b) Runtime overheads across all the sizes

We see that for mcf, astar, bzip and povray, the instructions overheads decrease as the workload size increases. The runtime trends, however, do not follow the same as the corrospoding runtimes overheads increase with an increase in input size. We can observe that

the increase in instructions overheads is not in proportion to runtime overheads. However, we can still see that the instructions overheads do change as the runtime overheads change indicating that there is indeed a relationship between these two. Therefore, we found the correlation coefficient between runtime overheads and instructions overheads in section 5.5 to understand this relationship.

5.3 Relationship between Cache Misses and Runtime Overheads

Increased cache misses can degrade runtime performance. To carefully analyze the cause of increased runtimes, we looked at the cache misses per kilo instructions in this section. Using VTune Amplifier XE, we found the total cache misses and number of instructions for every benchmark for both the input sizes. Then, we processed them to find cache MPKIs and plotted the bar graphs. The y-axes of these bar graphs are plotted in log scale to the base 4 for a better readability.

5.3.1 Cache MPKI and runtime overheads for size ref

The average runtime overhead for size ref for these 15 benchmarks is around 144%. Figure 5.4 shows the L1, L2 and L3 cache misses per kilo instructions for both native and MPX-enabled versions of benchmarks.

As seen in Figure 5.3, we found that the cache misses for lbm, exchange2 and bwaves are very low for both the native and MPX versions. Moreover, these misses remain almost the same at all the three cache levels. We further observe that benchmarks fotonik3d and roms also do not see much change in their cache misses at any cache level, when compiled with MPX.

It seems that even after compiling with MPX, the pointer-bounds check operations for these benchmarks might not have increased. Or they are so less that they are simply handled by the bounds registers. Therefore, there would have been no increase in cache misses. As we can see for these benchmarks, the runtime overheads are almost 100%, signifying no impact of MPX.

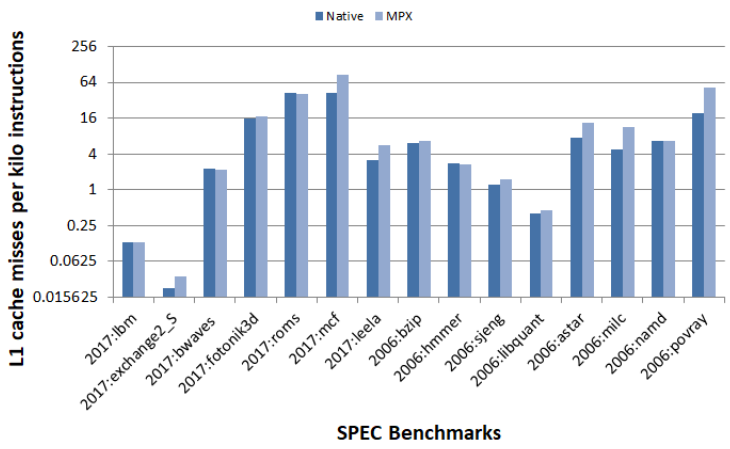
In Figure 5.1, 2006:povray ref had a very high runtime overhead of 235%. For povray, the L1 cache misses overhead is 270% when compiled with MPX. These overheads are even higher at L2(1716%) and L3(325%) cache levels. This hints at a strong correlation relationship between runtime and cache MPKI overheads for povray.

Benchmarks 2017:mcf (200%), 2006:milc(174%) and 2006:astar (166%) which were among the benchmarks with high runtime overheads also have high L1 cache MPKI overheads at 200%, 231% and 178% respectively. For mcf, the number of misses in L2 and L3 remains high throughout. The high number of cache misses increase at all the levels justify the high runtime overhead for mcf. The cache misses double for astar and the increase is even more in milc, resulting in very high cache MPKI overheads and can explain the performance degradation.

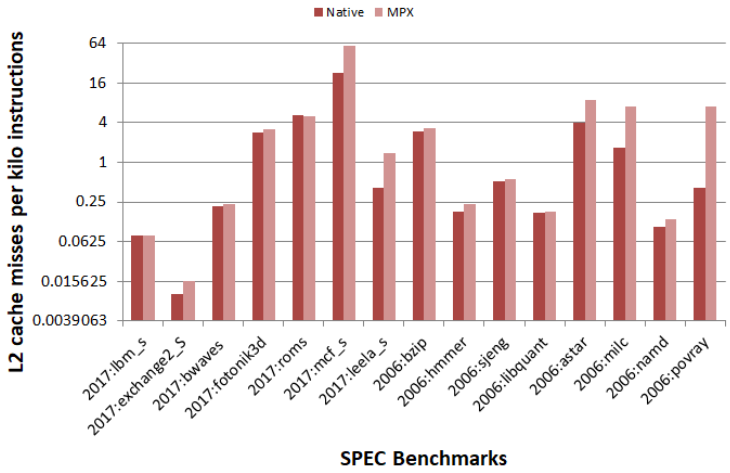
In 2017:leela, the L1 cache MPKI increase from 3 to 5.5. This increase of 2.5 misses per 1000 instructions translates to an overhead of around 176%. Similar observation can be made for L2 and L3 cache misses for this benchmark.

For 2006:namd, the cache misses do not vary much from the native version and the cache misses overheads remain at around 100%. However, at L2 and L3, the cache overheads increase which can explain its slight runtime overhead (118%). For 2006:sjeng, 2006:bzip and 2006:hmmer, the cache misses are very small in both native and MPX versions. However, the misses increase and have high runtime overheads. Therefore, we observe that while some

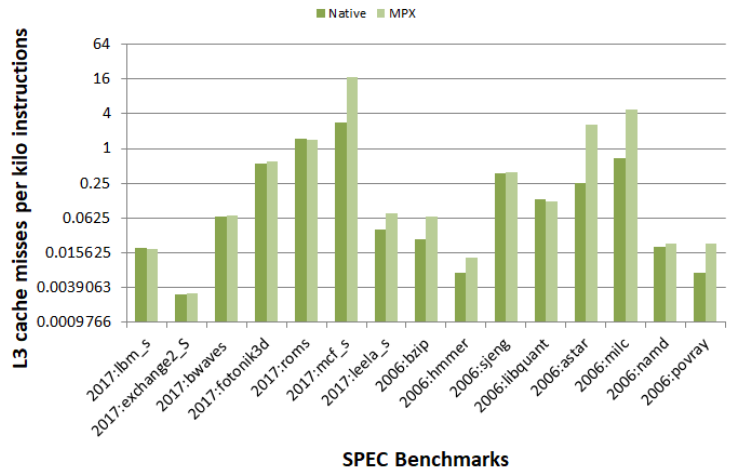
benchmarks have no runtime and cache misses overheads, we see some benchmarks which have very high cache misses overheads and high runtime overheads too. We will quantify the correlation between cache misses overheads and runtime overheads for size ref in section 5.5.



(a)



(b)



(c)

Figure 5.3: (a) L1 cache misses per kilo instructions for size ref(b) L2 cache misses per kilo instructions for size ref(c) L3 cache misses per kilo instructions for size ref

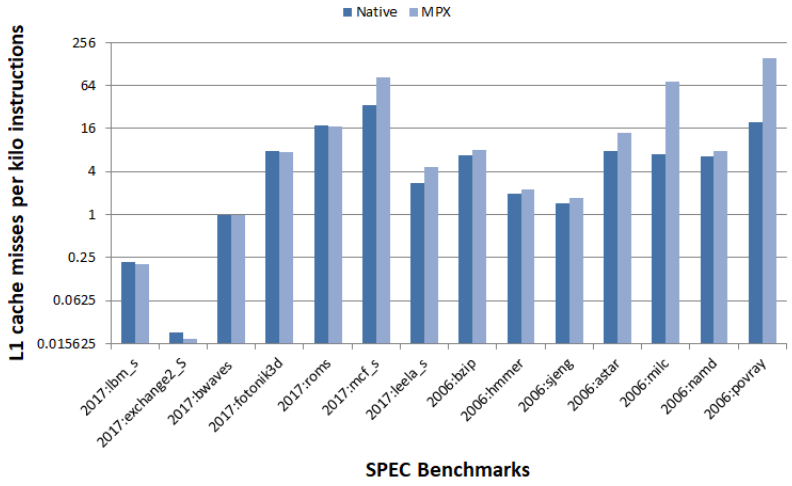
5.3.2 Cache MPKIs and runtime overheads for size train

The average runtime overhead for our 15 benchmarks with input size train is around 140%. Figure 5.4 shows the L1 cache MPKIs(a), L2 cache MPKIs(b) and L3 cache MPKIs(c). In this figure, we chose not to include libquantum statistics as VTune Amplifier could not latch to train size benchmark which ran for less than 0.5 second. Even after reducing the frequency to less than 1 GHz, we couldn't make it run for a time long enough for VTune Amplifier to latch and collect any statistics.

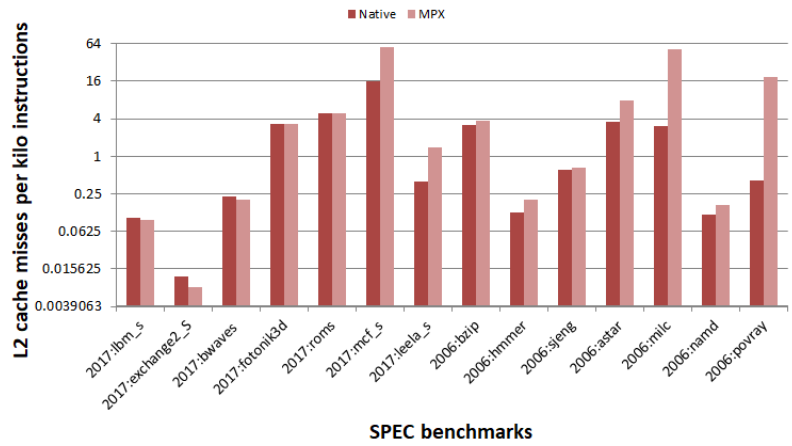
For input size train also, we observe that 2017:mcf, 2006:povray and 2006:milc show the highest cache misses at almost all the three levels. The benchmark 2006:astar sees the cache misses doubled up. As seen in Fig 5.1, these benchmarks have very high runtime overheads (>150%) as well. Just like input size ref, 2017:lbm, 207:bwaves, 2017:fotonik3d and 2017:roms do not have any increase in cache misses with MPX, at any cache level. Also, 2017:exchange2 show a very minor change in cache misses. As we can see in the section 5.1, these benchmarks do not show any runtime overhead when MPX is enabled for size train.

Benchmark 2006:namd shows some overheads in cache misses at L1 and L2 cache levels. It has a runtime overhead of 120% which can be justified by these cache misses overheads. Benchmarks like 2017:leela, 2006:hmmer, 2006:bzip and 2006:sjeng also show high overheads in both runtimes and cache misses.

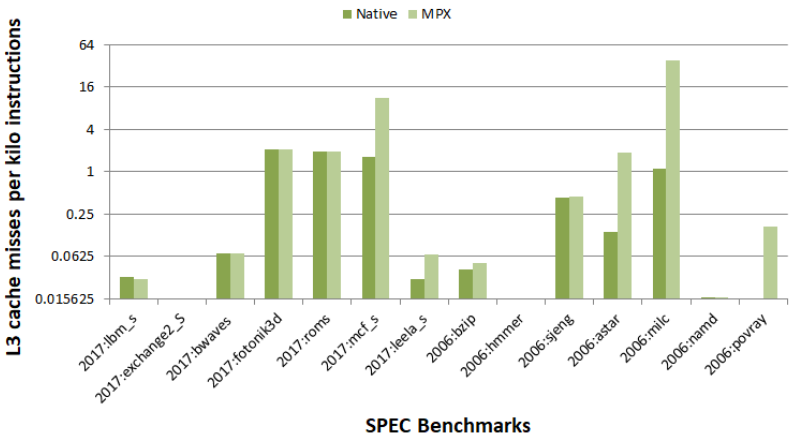
Therefore, for size train also, we see that benchmarks have a positive correlation with runtime overheads. We find the correlation coefficients to understand the relationships between these overheads in section 5.5.



(a)



(b)



(c)

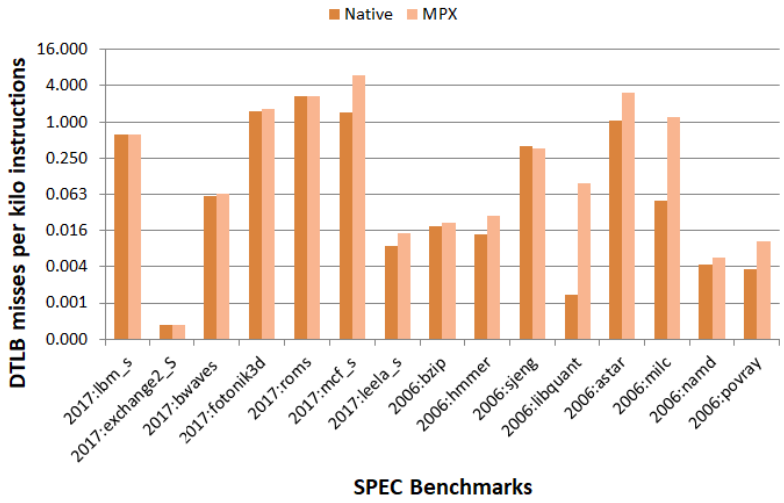
Figure 5.4: (a) L1 cache misses per kilo instructions for size train (b) L2 cache misses per kilo instructions for size train (c) L3 cache misses per kilo instructions for size train

5.4 Analysis of impact of Translation Look-aside buffer misses on performance

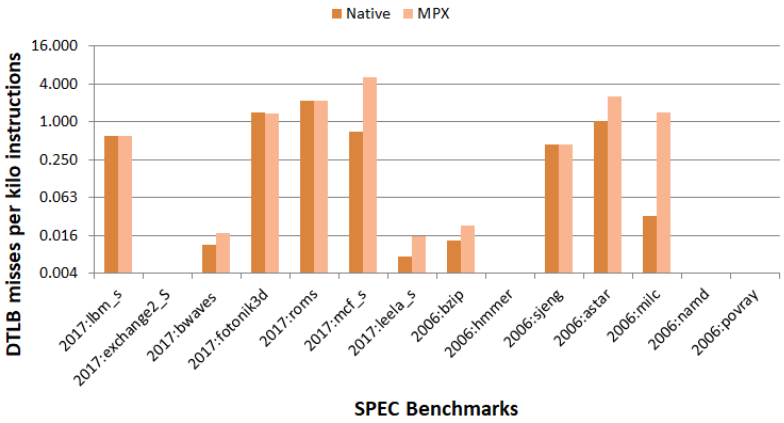
A TLB miss can be even more expensive than an instruction or a data cache miss. In our results, we have included number of TLB misses for which a page walk is completed. The misses correspond to misses at all the levels. Figure 5.5 shows the DTLB misses for size ref(a) and train(b). We could not gather statistics for 2006:libquantum for size train.

For both the sizes, the TLB misses overheads remain very high in benchmarks 2017:mcf, 2006:astar and 2006:milc, just like the cache MPKIs we observed in section 5.3. For 2017:lbm, 2017:exchange2, 2017:bwaves, 2017:fotonik3d and 2017:roms, we observe that there are little to no overheads in TLB misses for any size. Once again, these are the same benchmarks which have no runtime overheads.

For 2017:leela, 2006:sjeng, 2006:bzip the TLB misses overheads are high and we see high runtime overheads for these benchmarks as well. One additional observation we make is that for some benchmarks like leela, bzip, sjeng, the TLB misses overheads are higher in train compared to ref. In section 5.5, we found the correlation coefficients to quantify the relationships between TLB misses and runtime overheads.



(a)



(b)

Figure 5.5: (a) TLB misses per kilo instructions for size ref(b) TLB misses per kilo instructions for size train

5.5 Analysis of correlation between runtime and different overheads

So far, we have seen the instructions overhead, increase in cache misses and increase in TLB misses for our dataset of 15 benchmarks. In this section, we quantify our observations with

correlation coefficients. We used the Pearson coefficient that we described in section 4.2.

5.5.1 Correlation factors for size ref

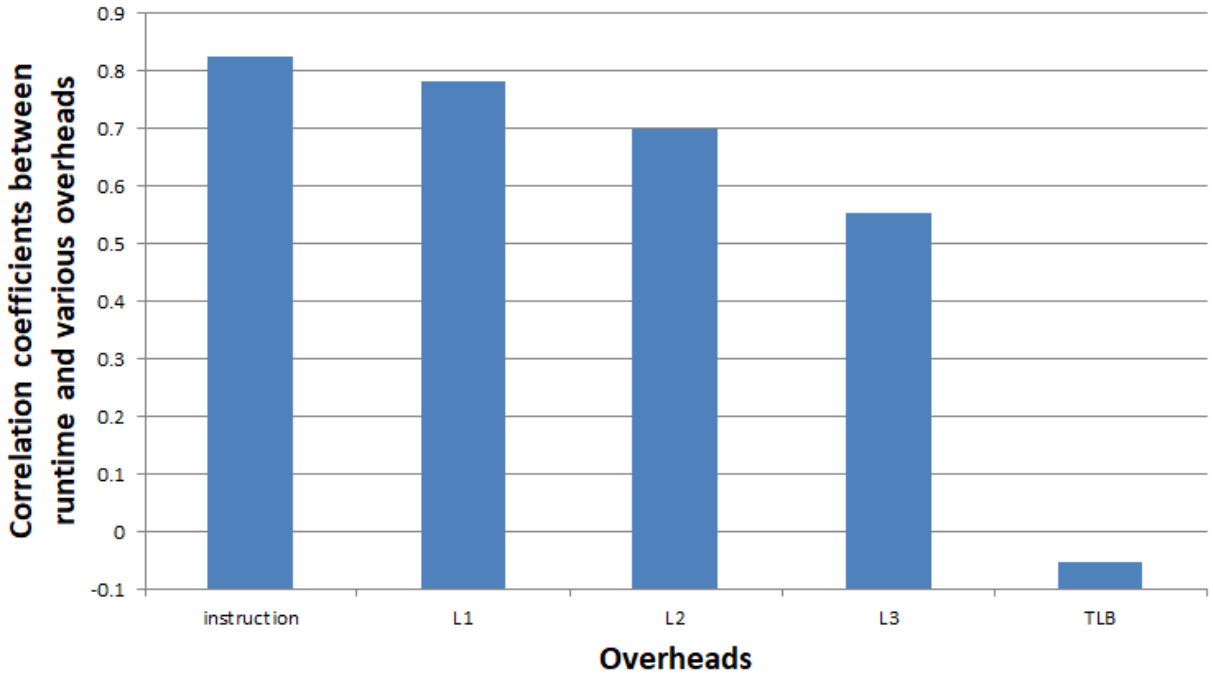


Figure 5.6: Correlation coefficient between runtime and various overheads for size ref

In figure 5.6, we have shown the correlation factors between runtime overheads and instruction overheads, L1/L2/L3 cache MPKIs overheads and TLB MPKIs overheads. This figure shows the correlation coefficients for all the 15 benchmarks. Looking at the benchmark, we see a strong positive correlation between instructions overheads and runtime overheads, with a correlation factor of greater than 0.8. This corroborates that as the number of instructions increase, the runtime also increases. The correlation is also strong between the L1/L2/L3 cache misses overheads and runtime overheads, with their correlation coefficients lying in the range of 0.55 to 0.78. This supports our hypothesis of correlation between the increase in cache misses and runtime overheads. However, it shows a very small negative correlation

factor of -0.05 between TLB misses overheads and runtime overheads. This gives us an idea that TLB misses are not correlated to the runtime overheads.

Looking at instructions and misses graphs in previous sections, we have seen that 5 out of 15 benchmarks do not have any overheads. That is, the number of instructions, cache misses and TLB misses remain the same in both native and MPX versions of these benchmark applications. These five benchmarks are lbm, exchange2, bwaves, fotonik3d and roms. Therefore, we excluded these 5 benchmarks and found correlation coefficients again, which we have shown in Figure 5.7.

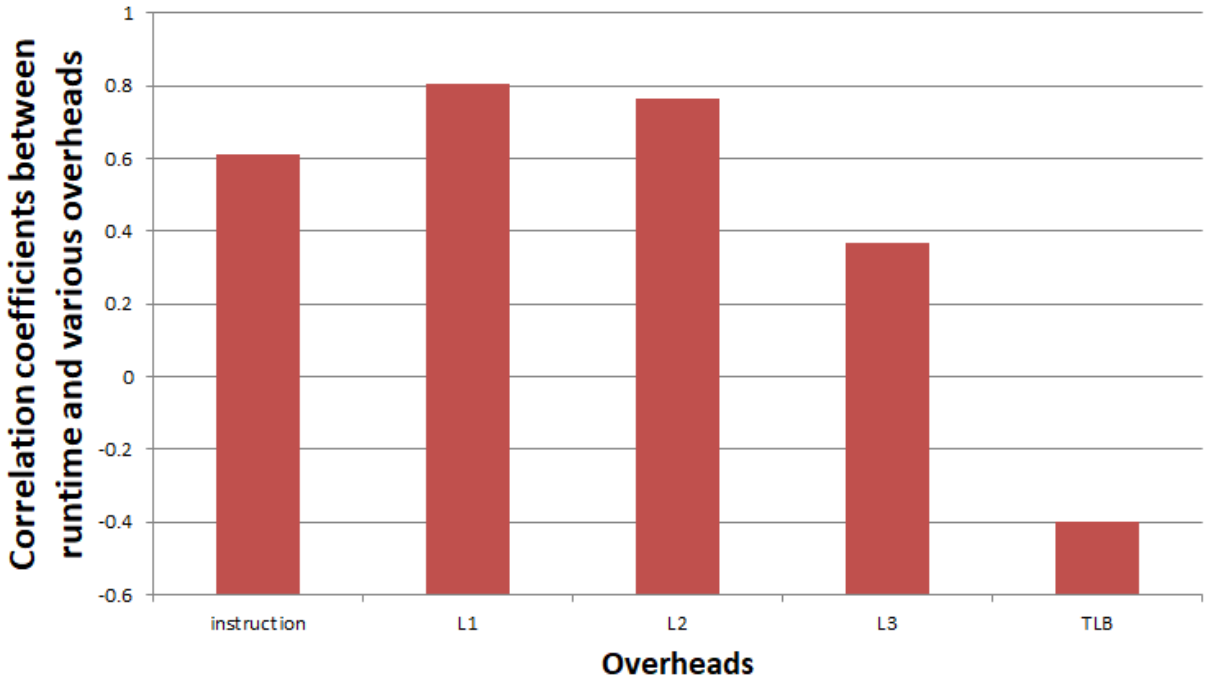


Figure 5.7: Correlation coefficient between runtime and various overheads for 10 benchmarks for size ref

In Figure 5.7, we can see that compared to overall dataset in figure 5.6, the correlation between instructions overhead and runtime overheads has decreased. However, the cache misses overheads' correlation coefficients have increased, suggesting strong positive relation-

ships. We see that the correlation coefficient between TLB misses overheads and runtime overheads is now -0.4 which means that as the TLB misses decrease in MPX versions, the runtime overheads are increasing for these 10 benchmarks.

5.5.2 Correlation factors for size train

In this section, we find the correlation coefficients between runtime overheads and instruction overheads, L1/L2/L3 cache MPKIs overheads and TLB MPKIs overheads for size train. We have shown the coefficients in Figure 5.8 for all the 15 benchmarks.

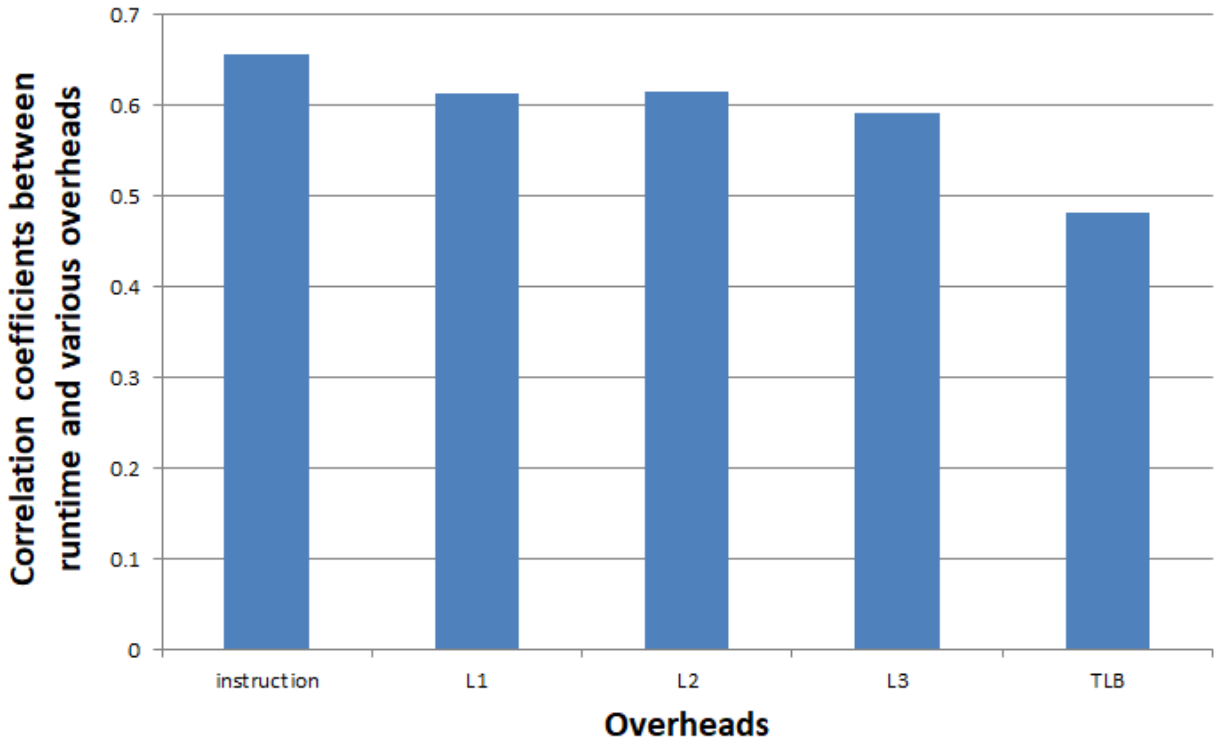


Figure 5.8: Correlation coefficient between runtime and various overheads for size train

For size train, we observe that the correlation coefficients decrease compared to size ref. However, instructions overheads and cache misses overheads still show a positive correla-

tion with runtime overheads. Here, we see that correlation coefficient between TLB misses overheads and runtime overheads has increased to 0.5. A value like this means a positive correlation between these two overheads. However, we observed almost zero correlation in case of size ref. We observe that for size train, the TLB misses overheads are higher than ref. These high TLB miss overheads could be the reason for us to observe a positive correlation coefficient when we were expecting it be like ref. The figure 5.8 shows the coefficients for all the 15 benchmarks. Figure 5.9 shows the correlation coefficients for the 10 benchmarks after removing the non-changing benchmarks.

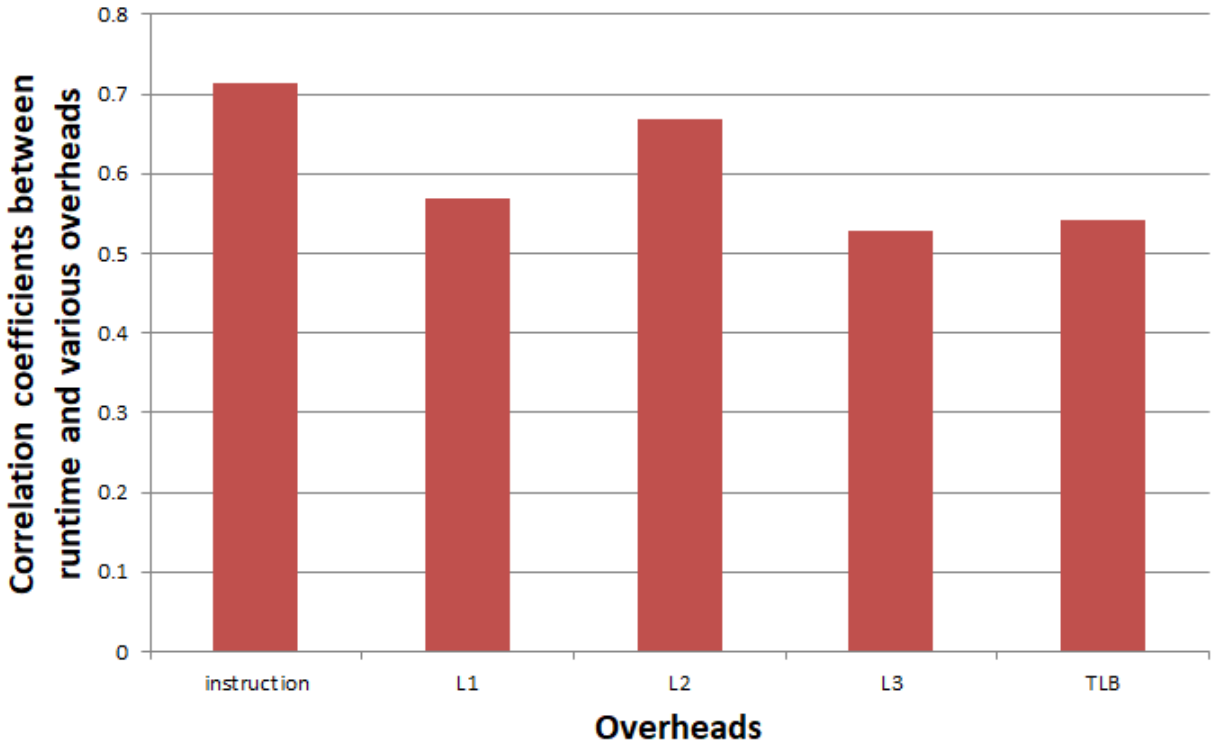


Figure 5.9: Correlation coefficient between runtime and various overheads for 10 benchmarks for size train

In this case, we still see a strong positive correlation between runtime overheads and instructions overhead. Compared to overall correlation in Figure 5.8, the correlation between

L1 cache misses overheads and runtime overhead decreases. However, we still see that there are positive correlations between cache misses and rutime overheads, proving an interdependence between them. The correlation coefficient between TLB misses overheads and runtime overheads has an even higher value than the one with whole dataset for size train.

Chapter 6

Conclusion

Intel's MPX provides effective protection against spatial memory errors by providing hardware assistance. The prior work has shown that runtime overheads associated with MPX are very high. There can be several reasons behind the performance degradation, cache misses being one of them. The design of MPX introduces a set of bound registers, seven new instructions among other changes in the hardware stack. These changes ensure pointer-bounds checking to enable spatial memory safety. However, the two-way address translation is also responsible for poor cache locality.

In our research, we attempt to discover the correlation between increase in cache misses due to MPX and runtime performance. We expect to see an increase in runtimes if there is an increase in cache misses in applications compiled with MPX. To test this hypothesis, we perform our evaluations on SPEC CPU 2006 and 2017 benchmark programs. We conduct our analysis on Windows platform and programs are compiled with ICC compiler. Using VTune Amplifier XE, we find L1/L2/L3 cache misses, number of instructions and TLB misses.

In our analysis on 15 SPEC CPU benchmarks for different input sizes, we observe an average of 144% slowdown for size ref(large) and 140% slowdown for size train(medium). We observe that 5 out of 15 benchmark (one third of total) do not show any runtime overheads for both the sizes. These benchmarks show no overheads in instructions after compiled with MPX. Also, their cache misses and TLB misses remain almost the same as native for both the input sizes.

We start with observing the relationship between runtime overheads and instructions overheads. While we observe that the instructions overhead do not follow the same trends as runtime overheads as the input size is increased from train to ref, we still observe that these two overheads are dependent on each other. On finding correlation coefficients, we see a positive relationship with coefficient value of 0.8 in ref and 0.65 in train. After removing the benchmarks with no runtime and instructions overheads, we see the coefficients changing to 0.6 in ref and 0.72 in train.

After that, we explore the relationships between increase in cache misses and runtime overheads. We find that memory-intensive benchmarks like 2017:mcf have high increase in cache misses at all the levels for both the sizes. Benchmarks like povray, astar and milc also show that their cache misses increase when compiled with MPX. For other benchmarks, we observe that their number of cache misses remain small in both native and MPX versions. However, we observe that these benchmarks have increase in cache misses when compiled with MPX resulting in high cache misses overheads. We calculate correlation coefficients to see the extent of correlation between cache misses overheads and runtime overheads. We find coefficient values ranging from 0.78 to 0.55 for size ref showing a strong correlation. We also find correlation coefficients after removing 5 benchmarks that showed almost no runtime and cache misses overheads. After that, we find even stronger relationships as the coefficients increase to around 0.8 for both L1 and L2 level misses overheads. For size train, we find the values to be around 0.6. These numbers again support the positive correlation between cache misses and runtime overheads for the benchmarks.

After instructions overheads and cache misses, we move on to find the correlation relationship between TLB misses and runtime overheads. For size ref, we find that benchmarks like mcf, astar, milc and povray have very high TLB misses overheads. Also, these benchmarks are found to have high runtime overheads. For benchmarks like leela, bzip, sjeng, again the runtime overheads are high. For overall overheads for size ref, we find the correlation

coefficient and it comes out to be very low signifying almost no correlation. However, for size train, we see that these sjeng, bzip and leela have high TLB misses overheads. For this size, we see a higher correlation factor of around 0.5.

Therefore, we come to the conclusion that both instructions overheads and cache misses have a strong correlations with the runtime performance. That is, if applications compiled with MPX are found to have runtime overheads, we would see an increase in number of instructions and cache misses.

Bibliography

[1] Correlation coefficient: Simple definition, formula, easy calculation steps. URL https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/correlation-coefficient-formula/.

[2] Performance evaluation. URL https://intel-mpx.github.io/performance/.

[3] National institute of standards and technology. national vulnerability database. URL https://web.nvd.nist.gov./.

[4] URL https://download.01.org/perfmon/index/skylake.html.

[5] Standard performance evaluation corporation, . URL https://www.spec.org/cpu2006/.

[6] Standard performance evaluation corporation, . URL https://www.spec.org/cpu2017/.

[7] Why do people use c if it is so dangerous? URL https://softwareengineering.stackexchange.com/questions/321547/why-do-people-use-c-if-it-is-so-dangerous.

[8] Intel® parallel studio xe 2013 sp1 intel® cluster studio xe 2013 sp1 - ppt download. URL https://slideplayer.com/slide/14117687/.

[9] Tautvidas sipavičius. URL https://www.tautvidas.com/blog/2011/04/disabling-intel-turbo-boost/.

[10] Intel® turbo boost technology 2.0. URL https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[11] What is system software? - definition from whatis.com. URL https://whatis.techtarget.com/definition/system-software.

[12] What does "instruction retired" mean exactly?, Apr 2003. URL https://software.intel.com/en-us/forums/intel-vtune-amplifier/topic/311170.

[13] Hardware-enforced comprehensive memory safety, May 2013. URL https://dl.acm.org/citation.cfm?id=2516415.

[14] amplxe-cl command syntax, Feb 2019. URL https://software.intel.com/en-us/vtune-amplifier-help-amplxe-cl-command-syntax.

[15] runsa/runss custom command line analysis, Feb 2019. URL https://software.intel.com/en-us/vtune-amplifier-help-running-runsa-runss-custom-analysis-from-the-command-line.

[16] What is a buffer overflow? learn about buffer overrun vulnerabilities, exploits attacks, Feb 2019. URL https://www.veracode.com/security/buffer-overflow.

[17] amplxe-cl actions, Feb 2019. URL https://software.intel.com/en-us/vtune-amplifier-help-amplxe-cl-actions.

[18] Bobs 1. Changing windows process priority via command line. URL https://superuser.com/a/620725.

[19] Amlendra, Amlendra, and Amlendra. Dangling, void , null and wild pointer, Mar 2019. URL https://aticleworld.com/dangling-void-null-wild-pointers/.

[20] Karthik AMR. Programming: How to improve application performance by understanding the cpu cache levels, Aug 2018. URL https://hackernoon.com/programming-how-to-improve-application-performance-by-understanding-the-cpu-cache-l

[21] David BalažicDavid Balažic. How to fix cpu clock (for benchmarking purposes)? URL https://superuser.com/questions/934749/how-to-fix-cpu-clock-for-benchmarking-purposes.

[22] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000. URL http://doi.acm.org/10.1145/1133981.1134000.

[23] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5629-9. doi: 10.1145/3185768.3185771. URL http://doi.acm.org.ezproxy.lib.vt.edu/10.1145/3185768.3185771.

[24] Raymond Chen. When you set a 100% cpu program to real-time priority, you get what you asked for, Jun 2010. URL https://devblogs.microsoft.com/oldnewthing/?p=13753.

[25] C. Cowan, F. Wagle, , S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, Jan 2000. doi: 10.1109/DISCEX.2000.821514.

[26] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, and Massimo Torquati. P3arsec: Towards parallel patterns benchmarking. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1582–1589, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4486-9. doi: 10.1145/3019612.3019745. URL http://doi.acm.org/10.1145/3019612.3019745.

[27] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Not.*, 41(6):144–157, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1133999. URL http://doi.acm.org/10.1145/1133255.1133999.

[28] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, May 2003. ISSN 0362-1340. doi: 10.1145/780822.781149. URL http://doi.acm.org/10.1145/780822.781149.

[29] Syed Sohaib Karim. Dangling pointers exploitation, Dec 2014. URL https://programmingexploitation.wordpress.com/2014/12/24/dangling-pointers-exploitation.

[30] Andrew Koenig. How c makes it hard to check array bounds. URL http://www.drdobbs.com/cpp/how-c-makes-it-hard-to-check-array-bound/240168083.

[31] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Jr., and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 721–732, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516713. URL http://doi.acm.org/10.1145/2508859.2516713.

[32] Hongli Lai. Understanding your benchmarks and easy tips for fixing them, Aug 2017. URL https://blog.phusion.nl/2017/07/13/understanding-your-benchmarks-and-easy-tips-for-fixing-them/.

[33] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 115–124, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346296. URL http://doi.acm.org/10.1145/1346281.1346296.

[34] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542504. URL http://doi.acm.org/10.1145/1542476.1542504.

[35] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 175:175–175:184, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2581122.2544147. URL http://doi.acm.org/10.1145/2581122.2544147.

[36] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz*

International Proceedings in Informatics (LIPIcs), pages 190–208, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.190. URL http://drops.dagstuhl.de/opus/volltexte/2015/5026.

[37] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, December 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409382. URL http://doi.acm.org/10.1145/1409360.1409382.

[38] Oleksii Oleksenko, Pascal Felber, Dmitrii Kuvaiskii, Christof Fetzer, and Pramod Bhatotia. Intel mpx explained. URL https://se.inf.tu-dresden.de/pubs/slides/Oleksenko18mpx.pdf.

[39] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, June 2018. ISSN 2476-1249. doi: 10.1145/3224423. URL http://doi.acm.org/10.1145/3224423.

[40] Mathias Payer. Memory safety. URL https://nebelwelt.net/teaching/17-527-SoftSec/slides/02-memory_safety.pdf.

[41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-0804-0. doi: 10.1109/HPCA.2007.346181. URL https://doi.org/10.1109/HPCA.2007.346181.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342849.

[43] Akash Sharan. Buffer overflow attack with example, May 2017. URL https://www.geeksforgeeks.org/buffer-overflow-attack-with-example/.

[44] Y. Solihin, M. Prvulovic, G. Venkataramani, and B. Roemer. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 273–284, Los Alamitos, CA, USA, feb 2007. IEEE Computer Society. doi: 10.1109/HPCA.2007.346205. URL https://doi.ieeecomputersociety.org/10.1109/HPCA.2007.346205.

[45] Cloyce D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, March 2007. ISSN 0163-5964. doi: 10.1145/1241601.1241625. URL http://doi.acm.org.ezproxy.lib.vt.edu/10.1145/1241601.1241625.

[46] Victor Stinner. How to run a stable benchmark. URL https://speakerdeck.com/haypo/how-to-run-a-stable-benchmark?slide=20.

[47] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.13. URL http://dx.doi.org/10.1109/SP.2013.13.

[48] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM*

International Conference on Computer-aided Design, ICCAD '94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-89791-690-5. URL http://dl.acm.org/citation.cfm?id=191326.191500.

[49] Hacking Tutorials. Buffer overflow explained: The basics, Dec 2017. URL https://www.hackingtutorials.org/exploit-tutorials/buffer-overflow-explained-basics/.

[50] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future, Sep 2012. URL https://link.springer.com/chapter/10.1007/978-3-642-33338-5_5.

[51] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL http://dl.acm.org/citation.cfm?id=2665671.2665740.

[52] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, September 2003. ISSN 0163-5948. doi: 10.1145/949952.940115. URL http://doi.acm.org/10.1145/949952.940115.