SPEC CPU2017: PERFORMANCE, ENERGY AND EVENT CHARACTERIZATION ON MODERN PROCESSORS

by

RANJAN HEBBAR SEETHUR RAVIRAJ

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical & Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2018

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____ _____

(student signature) (date)

THESIS APPROVAL FORM

Submitted by Ranjan Hebbar Seethur Raviraj in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering.

(Dr. Aleksandar Milenkovic) (date)	Committee Chair
(Dr. Rhonda Gaede) (date)	
(Dr. B. Earl Wells) (date)	
(Dr. Ravi Gorur) (date)	Department Chair
(Dr. Shankar Mahalingam) (date)	College Dean
(Dr. David Berkowitz) (date)	Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering
College/Dept. Engineering/Electrical & Computer Engineering
Name of Candidate Ranjan Hebbar Seethur Raviraj
Title SPEC 2017: Performance, Energy and Event Characterization on Modern Processors

Computer engineers in both academia and industry rely on a standardized set of benchmarks to quantitatively evaluate the performance of modern computer systems and research prototypes. The SPEC CPU2017 benchmark suites are the most recent incarnation of standard benchmarks designed to stress a system's processor, memory subsystem, and compiler. This thesis describes the results of measurement-based studies focusing on performance and energy-efficiency of modern Intel processors using SPEC CPU2017. The studies utilize SPEC CPU2017 run utilities as well as modern Linux tools for profiling that interface on-chip performance monitoring units. The thesis encompasses the following aspects of performance evaluation: (a) top-view characterization of individual benchmarks; (b) analysis of scalability in the context of speed and throughput metrics, while varying the number of threads for speed benchmarks and copies for rate benchmarks, respectively; (c) analysis using Intel's Top-down Microarchitectural Analysis Method, (d) comparative performance study of different computers with Intel's Core i7 and Xeon processors, and (e) analysis of performance impact of hardware prefetching in modern processors.

Abstract Approval: Committee Chair _____
Department Chair _____
Graduate Dean _____

ACKNOWLEDGMENTS

The work presented in this thesis would be incomplete without thanking all the people who helped me directly and indirectly. First, I would like to express my sincere gratitude to my advisor, Dr. Aleksandar Milenkovic for his constant support at every stage of this work for creating an inspirational work environment in the LaCASA laboratory. He inspired me personally and professionally with his patience and his interest towards student learning.

I will be always grateful to Dr. Ravi Gorur, Chair of the Electrical and Computer Engineering Department, for encouraging me to pursue my thesis studies, and for providing me with financial support through the teaching assistantship during Fall-2017, Spring-2018 and Summer-2018 semesters.

I would like to thank Mrs. Mounika Ponugoti, Mr. Prawar Poudel and Dr. Armen Dzhagaryan for their constant support and for helping me to get started in the laboratory.

I would like to thank Dr. Rhonda Gaede and Dr. Buren Wells for teaching me valuable skills and serving on my committee. I would also like to thank all the professors and staff members who helped me during my time at the University of Alabama in Huntsville.

Finally, I would like to express my deepest gratitude to my parents, Raviraj Hebbar and Jyothi Hebbar, for their unconditional love and support. I would like to thank my grandparents, Subramanya T S and Jayashree T S, for providing continuous support and encouragement for higher studies.

Dedicated to the memory of my grandmother, T S Jayashree,
who will forever be in our hearts.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Computing has been constantly evolving as major forces shaping it, technology, applications, and markets, continue to change and advance. Semiconductor technology processes that have been improving exponentially for almost six decades are approaching their limits. Two observations, known as Moore's Law and Dennard's Law, that succinctly describe exponential semiconductor scaling and that were true for many decades no longer hold. Moore's Law states that the number of transistors doubles with every 18-24 months. Dennard's Law states that the power density is constant for a given area of silicon – a larger number of transistors is compensated by their smaller size. Unfortunately, these "golden" years of semiconductor scaling are now over. Changes in markets and applications have been perhaps even more dramatic. Applications people could only dream of a decade ago are now a reality. Mobile and cloud computing have emerged as dominant computing models in the last decade. Internet-of-Things (IoT) promises to be a major driver for innovation in the years to come. In these conditions, complexity, and sophistication of the software stack continue to dramatically increase. Five distinct classes of computing have emerged: IoT/Embedded, Personal Mobile, Desktop, Server, and Cluster/Warehouse. Each is

characterized by its unique application sets, performance requirements, prices, form factors, and operating conditions.

Modern processors that power contemporary laptop, desktop, and server computers remain one of the most important components in computing ecosystems. Understanding their performance and limitations is important for application developers, system analysts, and computer designers alike. Even regular consumers may want to be able to quantitatively assess processor performance when making informed decisions about what computer to buy. Evaluating new systems by comparing their performance while executing relevant workloads – a mixture of real programs that users would run – is often not possible or practical. Instead, benchmark programs – well-established programs with well-defined inputs – are typically used to establish the relative performance of systems under test. Thus, developing standardized benchmark suites is particularly important to enable fair and through performance analysis.

SPEC (Standardized Performance Evaluation Corporation) is one of the most successful efforts in standardizing benchmark suites. Its origins arose from an effort to create a standardized set of benchmarks for comparing processor performance, but in the meantime, SPEC has evolved to developing benchmark suites applicable areas beyond processor performance analysis. There have been six generations of SPEC CPU benchmarks, starting from CPU89, via CPU92, CPU95, CPU2000, and CPU2006 to the most recent SPEC CPU2017. SPEC CPU benchmarks are portable across many machines and operating systems and they are designed to stress processor performance and minimize the impact of I/O on performance. Though SPEC CPU is perceived as a CPU intensive benchmark suit, it does emphasize memory and compiler

efficiency as well. The most recent SPEC CPU, CPU2017, has been in development for many years and is expected to be a cornerstone in performance evaluation of desktop and server computers in the years to come.

1.2 Scope of This Thesis

This thesis focuses on performance evaluation of modern processors using the SPEC CPU2017 benchmark suites. First, the history and evolution of SPEC CPU benchmarks are given, and then the composition and metrics of SPEC CPU2017 are discussed. SPEC CPU2017 includes four groups of benchmarks that differ in types of data processed (integer vs. floating point) and types of performance metrics of interest (speed vs. throughput). These four groups are:

- SPECspeed2017 Integer (*int_speed* for short)
- SPECspeed2017 Floating Point (*fp_speed* for short)
- SPECrate2017 Integer (*int_rate*), and
- SPECrate2017 Floating Point (*fp_rate*).

The measurement-based studies performed in this thesis rely on SPEC utilities to report execution times and SPEC CPU composite performance metrics. In addition, a set of modern tools for event-based sampling and profiling is used, including Linux utilities *perf* and *likwid*, and *Intel's VTune Amplifier*. These tools interface and gather information from on-chip performance monitoring units (PMU) that are part of modern processors' fabric. Statistics collected by PMU registers during benchmark execution include the number of clock cycles, the number of instructions executed, as well

as a myriad of microarchitecture-specific events that capture the behavior of a processor's front-end and back-end resources, such as branch predictors, functional units, and memory hierarchy.

This thesis includes several aspects of performance evaluation of modern processors using the SPEC CPU2017 benchmark suites as described below.

- The SPEC CPU2017 suites are run on a set of desktop and server computers featuring recent generations of Intel's Core i7 (Core i7-4700 and Core i7-8700K) and Xeon processors (Xeon E3-1240 V2 and Xeon E5-2643 V3). Performance of individual benchmarks and benchmark suites, in general, are analyzed and reported using SPEC CPU2017 metrics.
- Comparative performance analysis of multiple test systems is performed using SPEC CPU2017 metrics.
- The *Intel VTune Amplifier*'s Top-down Microarchitectural Analysis Method is used to identify performance bottlenecks for each benchmark, including both single-threaded and multi-threaded variants when possible.
- SPEC CPU2017 includes a number of benchmarks that can execute multiple threads. Thus, to understand the scalability of these parallel benchmarks, they are run while varying the number of threads. In throughput-oriented computing, multiple copies of the same benchmark are run, and corresponding performance metrics are analyzed.
- The effectiveness of hardware prefetching is quantitatively assessed for both single-threaded and multi-threaded benchmark runs.

1.3 Contributions

The main contributions of this work are as follows.

- Characterization of the SPEC CPU2017 benchmark suites executed on the latest 8th generation of Intel's Core i7-8700K.
- A Top-down analysis of the SPEC CPU2017 benchmark suites using *Intel VTune Amplifier* aimed at characterizing the performance of the individual processor components when running a diverse set of benchmarks.
- Experimental evaluation of the scalability of the SPEC CPU2017 benchmarks when varying the number of threads for the speed benchmarks and the number of copies for the rate benchmarks.
- Comparative performance analysis of the SPEC CPU2017 on four test machines, featuring different generations of Intel's Core i7 and Xeon processors.
- Performance evaluation of the effectiveness of hardware prefetching on the latest 8th generation of Intel's Core i7-8700K.

1.4 Findings

- For each of 43 benchmarks in the speed and rate suites, a top view with important performance and power metrics is given.
- The findings from the *Top-down Microarchitectural Analysis Method* are as follows. In general, single-threaded *fp_speed* benchmarks are mainly bound by stalls in the processor back-end. The number of instructions retired per clock cycle ranges from less than 1 to ~3.5. Single-threaded *int_speed* benchmarks are mainly bound by stalls in the front-end and bad

speculation. By increasing the number of threads in *fp_speed* benchmarks, the back-end stalls from memory hierarchy become an even more dominant source of stalls.

- The results from the scalability analysis show that several *fp_speed* benchmarks scale well, as long as the number of threads does not exceed the number of physical cores. However, other parallelized benchmarks showed little or no performance improvements. In all cases, the impact of hyper-threading on performance improvements has been limited.
- The *fp_rate* and *int_rate* benchmarks scale well as the number of copies increase up to the number of physical cores. The results indicate that the *fp_rate* benchmarks do not benefit from hyper-threading, whereas *int_rate* benchmarks see some performance improvements when the number of copies exceeds the number of physical cores but does not exceed the number of logical cores.
- Analyzing performance across different machines, the following observations can be made. (a) Clock frequency is the biggest driving force in performance improvements. The *fp_speed* benchmarks show performance gains of up to ~30% for the newest generation Core processor, whereas *int_speed* gains are marginal at ~8%. (b) The performance of the rate benchmarks is heavily affected by the size of last level cache (L3 in this case).
- Hardware-supported prefetching is very effective in the single-threaded speed and rate benchmarks. Disabling hardware prefetching significantly reduces the overall performance. Its impact on performance decreases as

the number of threads increases in the speed benchmarks or the number of copies in the rate benchmarks.

1.5 Outline

The rest of the thesis is organized as follows, CHAPTER 2 introduces the SPEC CPU benchmark suits, its history, and evolution and finally gives a detailed view of the latest iteration, SPEC CPU2017. CHAPTER 3 establishes experimental goals, gives a brief introduction to the Intel microarchitectures and their evolution, describes test systems, and finally describes the tools used in the experimental evaluation. CHAPTER 4 introduces the Top-down Microarchitectural Analysis Method, the concept of thread affinity, and discusses the results of the baseline SPEC CPU2017 evaluation. CHAPTER 5 describes an in-depth performance and power analysis of the SPEC CPU2017 individual suites using a test system with the latest 8th generation of Intel Core i7-8700K processor. In addition, events from the processor's performance monitoring unit are gathered to characterize the behavior of individual processor components. CHAPTER 6 describes the results of a study that compares scalability and performance of SPEC CPU2017 for different test machines and running conditions. Finally, CHAPTER 7 concludes the thesis and discusses future directions.

CHAPTER 2

SPEC CPU2017

This chapter gives a more detailed view of the SPEC CPU benchmark suits and the implications of their usage in this research. Section 2.1 gives a brief background on the need to have real application benchmarks for performance analysis and architectural research. Further discussion in Section 2.2 revolves around the history and evolution of SPEC CPU over many generations. Section 2.3 explores the latest iteration of SPEC CPU and gives a breakdown of its organization.

2.1 Background

With each new generation of modern processors, semiconductor technology nodes get smaller and more refined, resulting in an exponential increase in the number of transistors on a single chip. Unable to extract more parallelism from single-threaded programs and to cope with thermal issues of high-frequency deeply pipelined superscalar designs, chip manufacturers turned their focus to designing multicore and many-core processors that integrate multiple processor cores, DRAM controllers, cache hierarchies and even graphics controllers on a single chip [1]. However, recent trends have shown that there is a slowdown in further technology node reduction and that we are nearing the end of Moore's prediction (commonly known as Moore's Law) in semiconductors as we have known it for the last 50 years. Other performance improvements need to be explored. Though design complexity has increased rapidly over the years in the quest for higher performance, productivity and hardware utilization

have not been able to keep up with Moore's Law. Finding ways to utilize all the hardware resources available is one of the biggest challenges in computing.

Performance and energy driven designs in modern processors make it ever so complicated to profile and evaluate computing platforms. As architectural enhancements evolve, the method of comparing performance based on mere hardware specifications becomes obsolete. Benchmarking is the most widely used technique for measuring and comparing performance across different architectures. Benchmarks are designed to mimic a specific type of workload on a system. Synthetic benchmarks can be created to pinpoint workloads to a particular component. An application benchmark is a real-world program that could be used for performing system-wide analysis. Benchmarking has many uses. It acts as a useful tool when it comes to application-specific hardware purchases. An informed hardware purchase, focusing on application requirements saves time, money and energy. Benchmarking is also of great importance in CPU design. It allows the architects and designers to evaluate performance and make trade-offs in micro-architectural decisions. The architectural research relies on benchmarking to evaluate current systems for bottlenecks and evaluate enhancements proposed designs of future systems. It is thus of utmost importance to have standardized benchmarks that are representative of real-life applications.

With workloads and computing needs varying and evolving over time with the never-ending demand to process more data in less time, the world of performance evaluation is always evolving. Performance evaluations play a major role in determining bottlenecks and hotspots during program execution. Hence, it becomes extremely

important to have benchmarks that adapt and stay relevant even with shifts in computing needs.

The use of benchmarks is widespread and is accepted throughout the computing industry. For the above-stated reason, there are various suites available for different platforms and architectures. In the past, manufacturers have been known to use tricks to get better results for benchmarks which in other cases would not result in meaningful performance improvements. Hence it is preferable to have an unbiased benchmark suite that is portable across architectures and platforms. One such suite is SPEC CPU. SPEC CPU benchmarks, known to be uniform and considered a standard, offer CPU intensive workloads for measuring and comparing performance using the metrics of speed and throughput across different architectures.

2.2 History and Evolution of SPEC CPU

The System Performance Evaluation Cooperative, now named the Standard Performance Evaluation Corporation (SPEC), was founded in 1988. It is a non-profit organization that aims to "establish, maintain and endorse standardized benchmarks and tools to evaluate performance and energy efficiency" for computing systems [2]. SPEC initially consisted of a small number of workstation vendors who understood the need for standardized performance tests. Over time, SPEC has grown into a performance standardization entity with more than 60 member companies. Defining metrics which are fair and meaningful in the analysis helps differentiate systems under test. The path chosen by SPEC is an attempt to balance requiring strict compliance and allowing vendors to showcase engineering advantage. For comparisons to be interpreted as fair, the source codes are distributed so that the user can compile it in

the test environment. A pre-compiled set of binaries (executables) from a trusted source can also be used.

There have been six iterations of the SPEC CPU benchmarks namely CPU89, CPU92, CPU95, CPU2000, CPU2006, and CPU2017. With each new iteration, the benchmarks were updated in terms of complexity and workloads to keep up with advances in software and hardware. The first official set of CPU benchmarks was released in 1989 called SPEC Benchmark Release 1 suite, later identified as CPU89. CPU89 provided a standardized measure of compute-intensive microprocessor performance. This product replaced the vague and confusing MIPS and MFLOPS ratings then used in the computer industry [3]. These benchmarks were derived from real applications and were provided as source code to allow compilation on various workstations. Various running utilities ensured that all platforms would perform precisely the same task, providing comparability and uniformity across different architectures. CPU89 metrics did capture performance of the memory subsystem and the compiler too. As time progressed, SPEC CPU gained recognition and was adopted widely. But during this period compiler and processor technology improved drastically, requiring new benchmarks. Recognizing the shift, SPEC obsoleted CPU89 and released CPU92 in January 1992. In a similar trend, CPU95 was released in the summer of 1995. CPU95 incorporated subcomponents CINT95 and CFP95, which focused on integer and floating-point operation, respectively. CPU95, like its predecessor, was rolled out in the form of source code to be utilized in a wide range of computer platforms. Baseline results were introduced for the integer and floating-point benchmarks to maintain uniformity in test condition, in terms of compilers.

Though SPEC CPU is perceived as a CPU intensive benchmark suite, it does emphasize memory and compiler efficiency as well. In 2000, CPU2000 was released which consisted of 26 benchmarks divided into two subcomponents namely CINT2000 and CFP2000, like its predecessor. The next iteration, SPEC CPU2006, came out in 2006 and included two subcomponents namely, the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 and the SPECfp 2006 contains 19 benchmarks. SPEC CPU2006 gives an option to test both speed and throughput metrics. The SPECspeed metrics (e.g., the SPECint 2006 benchmark) are used for comparing the ability of a computer to complete a single task. The SPECrate metrics (e.g., the SPECint_rate 2006 benchmark) are used for comparing the ability of a computer to complete multiple tasks. CPU2006 was retired in 2017 with the release of its successor CPU2017.

2.3 CPU2017

The SPEC CPU2017 benchmark suites contains SPEC's latest, industry-standardized, CPU intensive suites for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem, and compiler. Building on CPU2006, CPU2017 contains 43 benchmarks, organized into four suites. A suite consists of a set of benchmarks that are run as a group and whose performance can be captured by a single number. Of the four suites in CPU2017, two are speed benchmarks namely SPECspeed2017 Integer (*int_speed* for short) and SPECspeed2017 Floating Point (*fp_speed* for short). For calculating SPECspeed metrics, one copy of the benchmark in a suite is run. The tester has the option of choosing the number of OpenMP threads for testing in hyper-threaded and multicore systems. A high score

means that less time is required to complete the suite. For each speed benchmark, SBi, after a successful run, a performance ratio, $Perf.Ratio(SBi)$ is calculated as shown in Eq.2.1, where $ExeTime(ref)$ is the execution time on the reference machine and $ExeTime(sut)$ is the execution time on the system under test.

$$Perf.Ratio(SBi) = \frac{ExeTime(ref)}{ExeTime(sut)} \qquad Eq.2.1$$

The remaining two suites of the SPEC CPU2017 are rate benchmarks namely SPECrate 2017 Integer (*int_rate* for short) and SPECrate 2017 Floating Point (*fp_rate* for short). For SPECrate metrics calculation, OpenMP is disabled. The tester has the option to select the number of concurrent copies of the same benchmark to run on the system. A high score means that more work is done per unit of time. For each rate benchmark, RBi, after a successful run, a performance ratio, $Perf.Ratio(RBi)$ is calculated as shown in Eq.2.2, where $ExeTime(ref)$ is the execution time on the reference machine, N is the number of copies executed and $ExeTime(sut)$ is the execution time on the system under test.

$$Perf.Ratio(RBi) = \frac{N * ExeTime(ref)}{ExeTime(sut)} \qquad Eq.2.2$$

A reference machine is used to normalize the performance metrics used in all the CPU2017 suites. Each benchmark is run on the reference machine and a measure of execution time is taken and is used as the reference time. The reference machine is a historical Sun Microsystems server, the Sun Fire V490 with 2100 MHz UltraSPARC-IV+ chips. The UltraSPARC-IV+ was introduced in 2006 and is newer than the processor used in the CPU2000 and CPU2006 reference machines (the 300 MHz

1997 UltraSPARC II) [4]. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. These times are then used in the SPEC calculations. The benchmark suites in SPEC CPU2017 are shown in Table 2.1.

Table 2.1 Benchmark Suites in SPEC CPU2017 [4]

Short Tag	Suite	Contents	Metrics	# Threads/Copies
int_speed	SPECspeed 2017 Integer	10 integer benchmarks	SPECspeed2017_int_base SPECspeed2017_int_peak	SPECspeed suites always run one copy of each benchmark. The user can set the number of threads.
fp_speed	SPECspeed 2017 Floating Point	10 floating point benchmarks	SPECspeed2017_fp_base SPECspeed2017_fp_peak	
int_rate	SPECrate 2017 Integer	10 integer benchmarks	SPECrate2017_int_base SPECrate2017_int_peak	SPECrate suites run all the benchmarks single threaded. The tester selects how many concurrent copies to run.
fp_rate	SPECrate 2017 Floating Point	13 floating point benchmarks	SPECrate2017_fp_base SPECrate2017_fp_peak	

The nominal memory requirements for SPEC CPU2017 are as follows:

- SPECrate: 2GB of memory per copy, if compiled for a 64-bit address space.
- SPECspeed: 16GB of main memory on the system, process limits must allow large stacks.

The integer benchmarks in the CPU2017 suites are derived from a wide variety of application fields (Table 2.2). The use of different programming languages offers a better range of compiler optimization techniques to be explored. The speed benchmarks focus on how fast a system's functional unit can execute instructions using all the available hardware resources. The speed benchmarks and rate benchmarks within the same pair (5nn benchmark for rate and 6nn, the benchmark for speed) are like

each other. Differences can be found in compile flags, run rules and size of the input workloads; generally, speed benchmarks require more memory than their rate counterparts. Table 2.2 lists the integer benchmarks in both the speed and rate suites, describes their source code type and size expressed in Kilo Lines of Code (KLOC), and gives their application field.

Table 2.2 Integer Benchmark Details [4]

SPECrate 2017 Integer	SPECspeed 2017 Integer	Language	KLOC	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

The floating-point benchmarks in the CPU2017 suites are also derived from a wide variety of application fields. Similar to the integer benchmarks, the floating-point benchmarks also have similar differences between SPECfpspeed and SPECfprate benchmarks. The SPECspeed benchmarks need large stacks, both for the

main process and the OpenMP threads. Table 2.3 lists the floating-point benchmarks in both the speed and rate suites, describes their source code type and size expressed in Kilo Lines of Code (KLOC), and gives their application field. Some of these benchmarks can be run with several inputs. The different inputs are explored in further sections.

Table 2.3 Floating-point Benchmark Details [4]

SPECrate 2017 Floating Point	SPECspeed 2017 Floating Point	Language	KLOC	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

The SPEC CPU2017 suites as a whole gives metrics for both speed and throughput using real-life applications to be used as a strong reference point for system evaluations. The large inputs representing modern workloads pose the issue of long runtimes. While experimenting with various parameters, it is challenging to evaluate multiple changes as runtimes can be extremely long. So, it becomes extremely important to choose metrics that uncover major bottlenecks and avoid getting diminishing results out of optimization. To do so efficiently, a baseline run is carried out and the results are analyzed using various tools to determine true bottlenecks and hotspots, providing insights so that an informed decision to minimize all known bottlenecks to get the best possible results can be obtained [5].

CHAPTER 3

TEST ENVIRONMENT & PROFILING TOOLS

This chapter gives a detailed overview of the objective of this study, experimental setup, the tools used for profiling, and the metrics used for evaluation. Section 3.1 describes the experimental goals of this research. Section 3.2 explains the micro-architectures and hardware changes over different generations. Section 3.3 gives information about the systems used for measurements and the conditions used during profiling. Section 3.4 covers all the tools used for the study. All the measurements were carried out on the test systems sponsored by the LaCASA Laboratory at UAH.

3.1 Experimental Goals

The measurement-based studies performed in this thesis rely on SPEC utilities to report execution times and SPEC CPU composite performance metrics. In addition, a set of modern tools for event-based sampling and profiling is used, including Linux utilities *perf* and *likwid*, and *Intel VTune Amplifier*. These tools interface and gather information from on-chip performance monitoring units (PMU) that are part of modern processors' fabric. Statistics collected by PMU registers during benchmark execution include the number of clock cycles, the number of instructions executed, as well as a myriad of microarchitecture-specific events that capture the behavior of a processor's front-end and back-end resources, such as branch predictors, functional units, and memory hierarchy.

This thesis includes several aspects of performance evaluation of modern processors using SPEC CPU2017 benchmark suite as described below.

- The SPEC CPU2017 suites are run on a set of desktop and server computers featuring recent generations of Intel's Core i7 (Core i7-4700 and Core i7-8700K) and Xeon processors (Xeon E3-1240 V2 and Xeon E5-2643 V3). Performance of individual benchmarks and benchmarks suites, in general, are analyzed and reported using SPEC CPU2017 metrics.
- Comparative performance analysis of multiple computers is performed using SPEC CPU2017 metrics.
- The *Intel VTune Amplifier*'s Top-down Microarchitectural Analysis Method is used to identify performance bottlenecks for each benchmark, including both single-threaded and multi-threaded variants when possible.
- SPEC CPU2017 includes a number of benchmarks that can execute multiple threads. Thus, to understand the scalability of these parallel benchmarks, they are run while varying the number of threads. In throughput-oriented computing, multiple copies of the same benchmark are run, and corresponding performance metrics are analyzed.
- The effectiveness of hardware prefetching is quantitatively assessed for both single-threaded and multi-threaded benchmark runs.

3.2 Intel Microarchitectures: An Overview

This section gives a bird's eye view of the different microarchitectures produced by Intel over the years. The discussion starts from the Ivy Bridge Microarchitecture released in 2012 and looks at all further improvements over the years and ends with

the Coffee-Lake Microarchitecture released in 2017. Intel processors are based on a "tick-tock" development process. At first, a "tock" comes with a new microarchitecture that uses the same technology node as before. The next generation is followed by a "tick" which comes with a new smaller technology node but the same microarchitecture. This type of development allows both sources of improvements to mature and cuts cost. The "tick-tock" representation is as shown in Figure 3.1 below. It should also be noted that the conventional notation of representing a technology node, the distance between drain and source, does not specifically apply for a three-dimensional MOSFET. Conventionally, transistor size reduction has played a key role in speed and energy improvements. But for four full generations of the Intel lineup, the same technology node was used with refinements. This shows a break from the "tick-tock" approach. Though size has lately remained the same, other forms of performance enhancements such as better parallelization, faster memory interconnect, and larger caches have maintained a nearly 30% improvement in performance and 15-20% power reduction each generation. All the information about the internal structure is obtained from the Intel Optimization Reference manual [6].

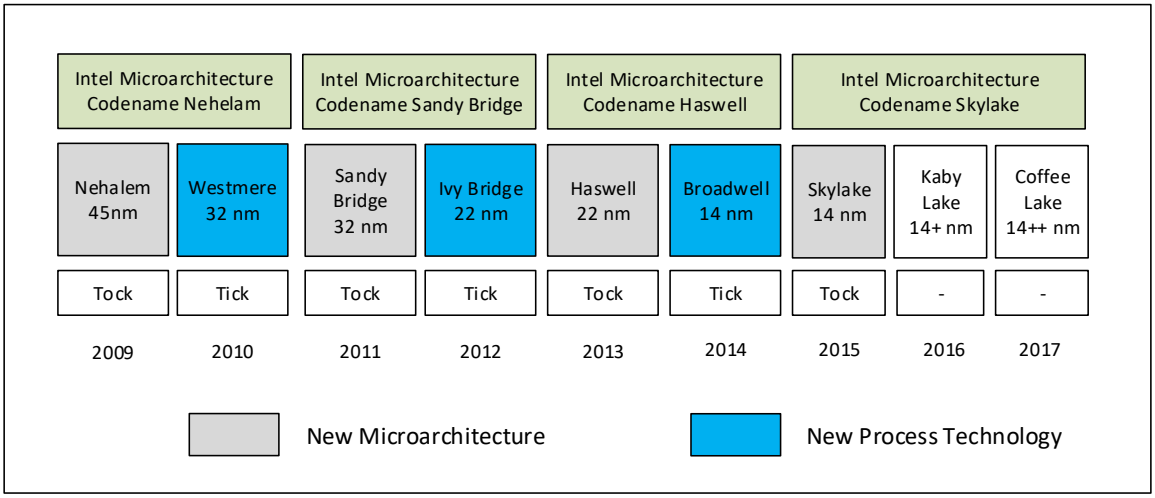

Figure 3.1 Tick-Tock Model Representation [7].

Each generation of Intel microarchitecture contains two variants, the microarchitecture for the Core processors and for the Xeon processors. Though the internal core architecture is similar, the design of the Xeon is oriented towards prolonged usage and lower power consumption. To achieve that, a Xeon is usually clocked at a lower clock frequency to have a lower operating temperature, whereas a Core i7 is usually clocked at a higher clock frequency and operates at higher temperatures. Most Xeons also do not come with an on-chip graphics processing unit, thus requiring discrete graphics cards. With larger cache memory, the Xeons are better suited for server applications. Xeon processors are qualified to handle heavier, more intensive loads day in and day out. For the serious workstation user, this can translate to better longevity over their Core i7 counterparts. Error Checking and Correction (ECC) RAM detects and corrects most common data corruption before it occurs, eliminating the cause of many systems crashes and translating to more stable overall performance. Only Xeon processors support ECC RAM.

3.2.1 Intel Ivy Bridge

The Intel Ivy Bridge was the upgrade to the Sandy Bridge microarchitecture moving to a new 22-nm technology node from the previous 32-nm technology node. It was released in the second quarter of 2012. As it was a "tick" in Intel's processor cycle, the internal microarchitecture remained the same as Sandy Bridge with minor tweaks. A Core i7-4770 has about 1.4 billion transistors in about 160 mm^2 [8]. Processors based on Ivy Bridge are commonly known as the third-generation core processors.

Figure 3.2 shows an annotated die map of a quad-core Ivy Bridge processor. A similar die map can be seen across all quad-cores in that generation, apart from some of the version 2 Xeon lineup that does not include the onboard graphics processor. A

significant increase in on-chip graphics area for the core processors results in faster graphics control.

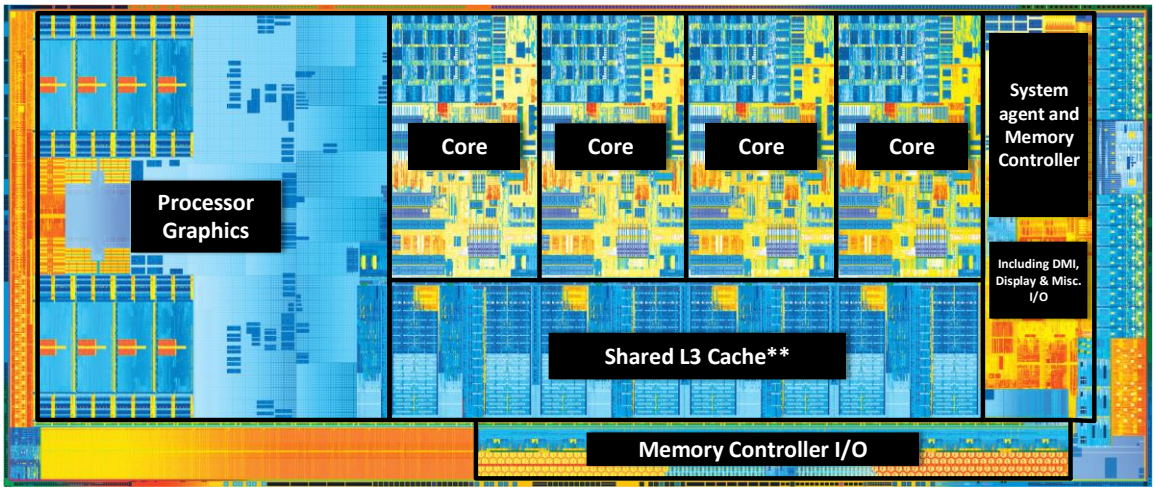

Figure 3.2 Die Map of a Quad-Core Ivy Bridge Processor [9]

As the Intel Ivy Bridge processors are based on the Intel Sandy Bridge micro-architecture, an internal look at the Sandy Bridge microarchitecture is given in this section. The Intel Sandy Bridge microarchitecture specifies an out of order superscalar design which can dispatch up to six micro-instructions to execution units per CPU clock cycle. It has a 14-stage pipeline (16 with fetch/retire). The dispatched instructions are reordered to "dataflow" order so that they can execute as their respective sources are available. Instructions with no data dependencies will generally execute as they come using first in first out (FIFO) policy. The data cache is non-blocking and can handle multiple simultaneous misses [6]. Each processor with enabled hyper-threading will have two logic cores. Figure 3.3 depicts the internal CPU block diagram of a Sandy Bridge processor. The internal functional units can be segregated into the front-end and the back-end.

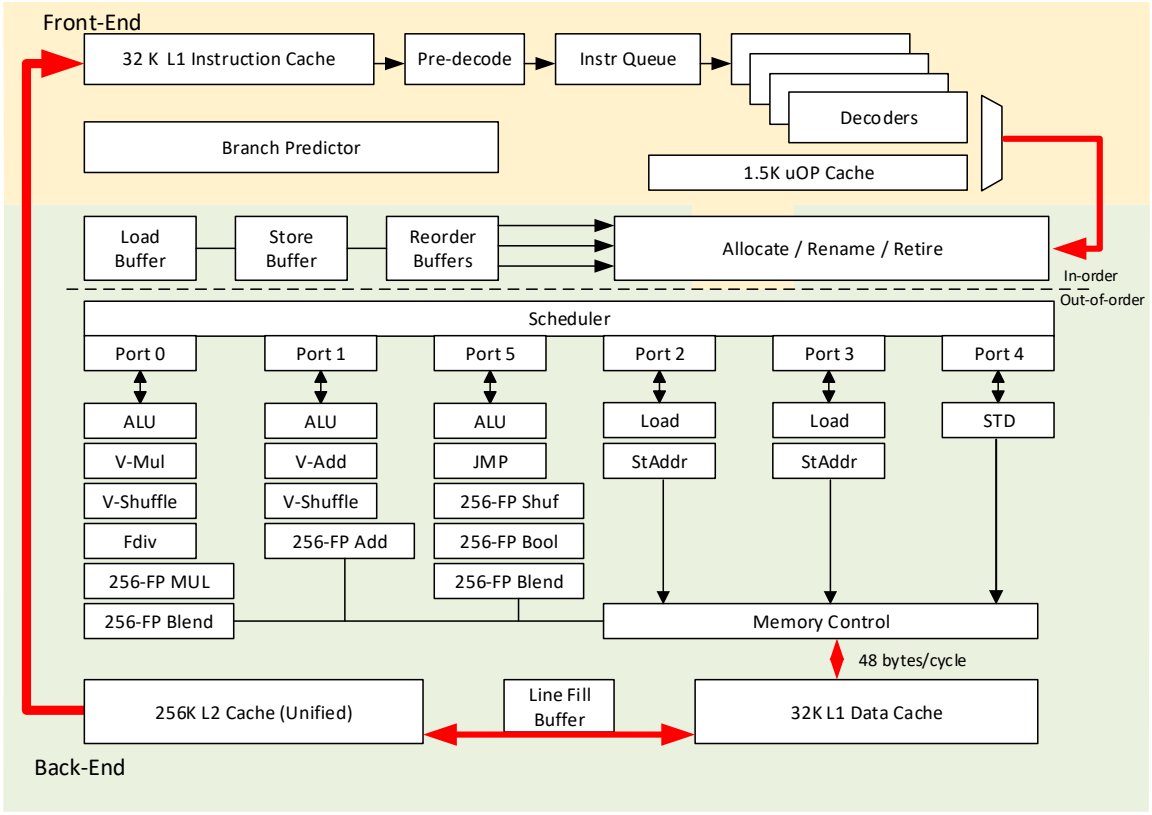

Figure 3.3 Sandy Bridge/Ivy Bridge CPU Core Block Diagram [6]

The front-end of the processor is responsible for fetching instructions from memory and translating them into micro-operations. These translated micro-operations are then fed to the back-end of the processor. The back-end handles scheduling, execution and retiring of instructions. The flow of an instruction through the pipeline can be illustrated as follows. Initially, the branch prediction unit (BPU) chooses the next 16-byte block of instructions for execution. The processor then searches for instructions in the Decode ICache, Instruction Cache, L2 cache, last level cache (LLC) and memory in that order, as necessary. The instructions fetched from the L1I cache or above are then converted into micro-operations and sent to the rename/retirement block. They enter the scheduler in program-order but execute out of

order. Branch mispredictions are found at branch executions and they redirect the front-end as necessary. Memory operations are parallelized for maximum performance. Exceptions are signaled at retirement of the faulting instruction. The common components of the front-end are as shown in Table 3.1.

Table 3.1 Components of the Front-End of Intel Ivy Bridge [6]

Component	Function
Instruction Cache	32-Kbyte backing store of instruction bytes.
Legacy Decode Pipeline	Decode instructions to micro-operations, delivered to the micro-operation queue and the Decoded ICache.
Decoded ICache	Provide a stream of micro-operations to the micro-operation queue.
MSROM	Complex instruction micro-operation flow store, accessible from both Legacy Decode Pipeline and Decoded ICache
Branch Prediction Unit (BPU)	Determine next block of code to be executed and drive lookup of Decoded ICache and legacy decode pipelines.
Micro-op queue	Queues micro-operations from the Decoded ICache and the legacy decode pipeline.

The Legacy Decode Pipeline is comprised of the instruction translation lookaside buffer (ITLB), instruction cache (ICache), instruction pre-decode, and instruction decode units. An instruction fetch is a 16-byte aligned lookup through the ITLB and into the instruction cache. The pre-decode unit accepts the 16 bytes for each cycle from the instruction cache and determines the length of the instructions. There are four decoding units that decode instructions into micro-operations. The first unit can decode all IA-32 and Intel 64 instructions, producing up to four micro-operations. The remaining three decoding units handle single micro-operation instructions.

Micro-fusion fuses multiple micro-operations from the same instruction into a single complex micro-operation. The complex micro-operation is dispatched in the out-of-order execution core as many times as it would if it were not micro-fused. Micro-fusion enables the use of memory-to-register operations, also known as the complex instruction set computer (CISC) instruction set, to express the actual program operation without worrying about a loss of decode bandwidth.

Macro-fusion allows the processor to merge common x86 instruction pairs into one micro-operation that can be executed in a single clock on a single ALU. At first, the instructions are read from the instruction queue. A fusible pair of instructions is sent to a single decoder. The obtained single micro-operation represents two instructions.

Micro-operations emitted by the decoders are directed to the micro-operation queue and to the Decoded ICache. Instructions longer than four micro-operations generate their micro-operations from the MSROM. The Decoded ICache is essentially an accelerator of the legacy decode pipeline. By storing decoded instructions, the Decoded ICache enables reduced latency on branch mispredictions, increased micro-operation delivery bandwidth to the out-of-order engine, and reduced front-end power consumption.

Branch prediction predicts the branch target and enables the processor to begin executing instructions long before its true execution path is known. All branches utilize the branch prediction unit (BPU). The BPU predicts the target address not only based on the EIP (EIP refers to the next instruction to be executed) of the branch but also based on the execution path through which execution reached this EIP.

The BPU can efficiently predict the following branch types:

- Conditional branches.
- Direct calls and jumps.
- Indirect calls and jumps.
- Returns.

The dynamic branch prediction unit consists of two major parts: a branch target buffer (BTB) for the prediction of branch targets, and an outcome predictor for the prediction of branch outcomes. The BTB is a cache structure, where a part of the branch address is used as the cache index, and the last target address of that branch is the cache data [10]. Unfortunately, the branch predictor organization and operation are not disclosed by the manufacturer. With the use of experimental reverse engineering, it is found that the branch predictor unit used in the Intel processors is a 4096-entry bimodal predictor [11].

The micro-operation queue decouples the front-end and the out of order engine. It stays between the micro-operation generation and the Renamer. The micro-operation queue provides post-decode functionality for certain instruction types.

The back-end, also known as the out-of-order (OOO) engine, can detect dependency chains and sends those chain of instructions for execution while maintaining data-flow. If a dependency chain is waiting for resources, micro-instructions from a secondary dependency chain are sent for execution to increase the instruction per cycle (IPC). The major components of the back-end are the Renamer, Scheduler and the Retirement unit. The Renamer component moves up to four micro-operations every cycle from the front-end to the execution core. It eliminates false dependencies among micro-operations, thereby enabling out-of-order execution of micro-operations. The

Scheduler component queues micro-operations until all source operands are ready and schedules and dispatches ready micro-operations to the available execution units in as close to a first in first out (FIFO) order as possible. Depending on the availability of dispatch ports and write-back buses, and the priority of ready micro-operations, the scheduler selects which micro-operations are dispatched every cycle. The Retirement component retires instructions and micro-operations in order and handles faults and exceptions.

The execution core is superscalar and can process instructions out-of-order. When a dependency chain causes the machine to wait for a resource (such as a second-level data cache line), the execution core executes other instructions, increasing the overall rate of instructions executed per cycle (IPC). The out-of-order core consists of three execution stacks, where each stack encapsulates a certain type of data: a general-purpose integer, an SIMD integer and floating-point, and an X87. The execution core also contains connections to and from the cache hierarchy. The loaded data is fetched from the caches and written back into one of the stacks.

The scheduler can dispatch up to six micro-operations every cycle, one on each port. Table 3.2 summarizes which operations can be dispatched on which port. After execution, the data is written back on a write-back bus corresponding to the dispatch port and the data type of the result. When a source of a micro-operation executed in one stack comes from a micro-operation executed in another stack, a one or two-cycle delay can occur. The cache hierarchy contains a first level instruction cache, a first level data cache (L1 DCache) and a second level (L2) cache, that is private to each core.

The caches may be shared by two logical processors if the processor is hyper-threaded. The L2 cache is shared by instructions and data. All cores in a physical processor package connect to a shared last level cache (LLC) via a ring connection. L2 is not inclusive of the data in L1. Only the LLC is inclusive of all the levels above it. Real delay is a factor of how far the required data is from the core. Each cache line in the LLC holds an indication of the cores that may have this line in their L2 and L1 caches. If there is an indication in the LLC that other cores may hold the line of interest and its state might have to modify, there is a lookup into the L1 DCache and L2 of these cores too.

Table 3.2 Dispatch Port and Execution Stacks [6]

	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
Integer	ALU, Shift	ALU, Fast LEA, Slow LEA, MUL	Load_Addr Store_addr	Load_Addr Store_addr	Store_data	ALU, Shift, Branch, Fast LEA
SSE-Int, AVX-Int, MMX	Mul, Shift, STTNI, Int-Div, 128b-Mov	ALU, Shuf, Blend, 128b-Mov			Store_data	ALU, Shuf, Shift, Blend, 128b-Mov
SSE-EP, AVX-FP_low	Mul,Div, Blend, 256b- Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov
X87, AVX-FP_High	Mul,Div, Blend, 256b- Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov

Table 3.3 shows the best-case cache latency in the correct lookup order. Real delay is a factor of how far the required data is from the core.

Table 3.3 Best Case Cache Latency/ Load Latency [6]

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1 Data	4	2 x16 bytes
L2 (Unified)	12	1 x 32 bytes
Third Level (LLC)	26-31	1 x 32 bytes
L2 and L1 DCache in other cores if applicable	43 - clean hit; 60 - dirty hit	

The L1 DCache is the first level data cache. It manages all load and store requests from all types through its internal data structures. The L1 DCache enables loads and stores to issue speculatively and out-of-order ensures that retired loads and stores have the correct data upon retirement and ensures that loads and stores follow the memory ordering rules of the IA-32 and Intel 64 instruction set architecture. The common load latency for L1 DCache is five cycles. Table 3.4 shows the components of the L1 Data Cache.

Table 3.4 L1 Data Cache Components [6]

Component	Intel microarchitecture code name Sandy Bridge
Data Cache Unit (DCU)	32KB, 8 ways
Load buffers	64 entries
Store buffers	36 entries
Line fill buffers (LFB)	10 entries

The L1 DCache architecture can service two loads per cycle, each of which can be up to 16 bytes. The LLC consists of multiple cache slices. The number of slices is equal to the number of IA cores. Each slice has both logic and data array portions. The logic portion handles data coherency, memory ordering, access to the data array portion, LLC misses, writeback to memory, and more. The data array portion stores cache lines. Each slice contains a full cache port that can supply 32 bytes/cycle. The physical

addresses of the data kept in the LLC data arrays are distributed among the cache slices by a hash function, such that addresses are uniformly distributed. The data array in a cache block may have 4/8/12/16 ways corresponding to the 0.5M/1M/1.5M/2M block size.

Data can be speculatively loaded into the L1 DCache using software prefetching, hardware prefetching, or any combination of the two. The various hardware prefetching mechanisms provided by the Intel microarchitecture code name Sandy Bridge and their improvement over previous processors are discussed further. The goal of the prefetchers is to automatically predict which data the program is about to consume. If this data is not close-by to the execution core or inner cache, the prefetchers bring it from the next levels of cache hierarchy and memory. Two hardware prefetchers load data to the L1 DCache. The first one is a data cache unit (DCU) prefetcher. This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line. The second one is the instruction pointer (IP) based stride prefetcher. This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to 2K bytes.

3.2.2 Intel Haswell Microarchitecture

The Intel Haswell microarchitecture was the successor to the Intel Ivy Bridge. As it was a “tock” in Intel’s processor cycle, the technology node remained the same as

the previous generation (Ivy Bridge) but the microarchitecture went through a redesign. Processors based on the Haswell microarchitecture are commonly known as fourth-generation core processors. Figure 3.4 shows an annotated die map of a quad-core Haswell processor.

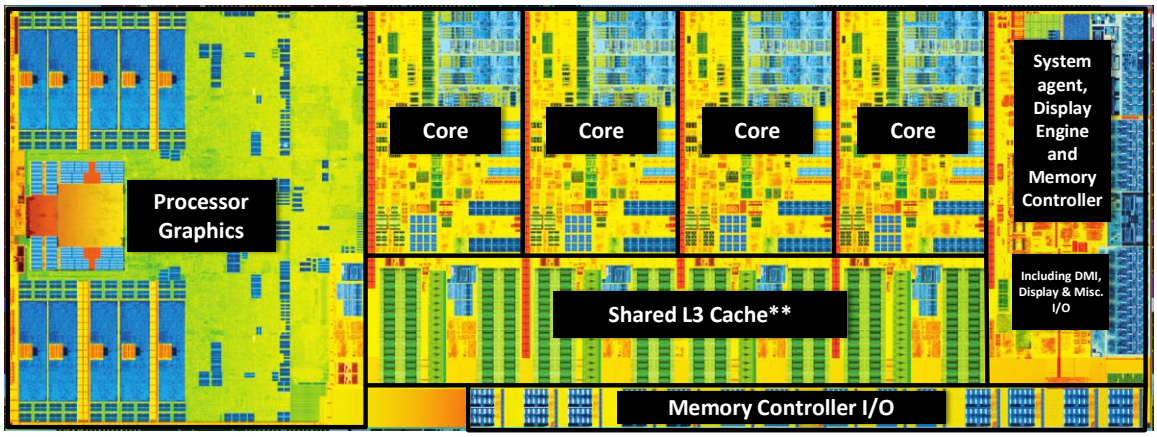

Figure 3.4 Die Map of a Quad-Core Haswell Processor[12]

Figure 3.5 shows the internal CPU Core Block Diagram of the Haswell microarchitecture. As a redesign from Ivy Bridge, multiple aspects have been improved in the Haswell architecture. It included added support to the Intel Advanced Vector Extension 2. The new microarchitecture can dispatch up to 8 micro-operations per cycle. It has two branch execution units.

The front-end of the Haswell microarchitecture builds on the Ivy Bridge. Additional enhancements are portrayed here. It has a 14-stage pipeline (16 with fetch/retire). The micro-operation cache (or decoded ICache) is partitioned equally between two logical processors. The instruction decoders will alternate between each active logical processor. If one sibling logical processor is idle, the active logical processor uses the decoders continuously. The loop stream detector (LSD) in the micro-op queue

(or IDQ) can detect small loops of up to 56 micro-operations. The 56-entry micro-operation queue is shared by two logical processors if hyper-threading technology is active (Ivy Bridge provides duplicated 28-entry micro-operation queues in each core).

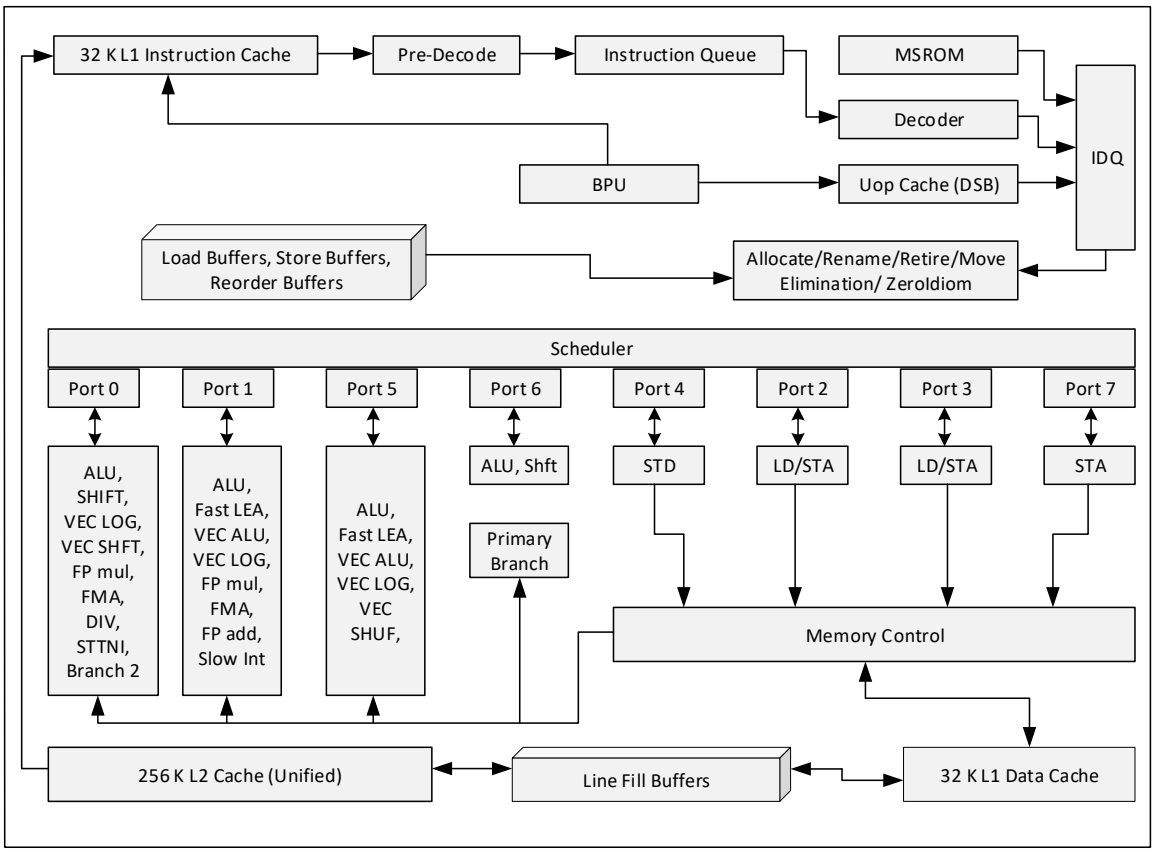

Figure 3.5 Haswell CPU Core Block Diagram [6]

The key components and significant improvements to the out-of-order engine are summarized as follows. The Renamer moves micro-operations from the micro-operation queue to bind to the dispatch ports in the Scheduler with execution resources.

The Scheduler controls the dispatch of micro-operations onto the dispatch ports. There are eight dispatch ports to support the out-of-order execution core. Four of the eight ports provide execution resources for computational operations. The other 4 ports support memory operations of up to two 256-bit load and one 256-bit store

operation in a cycle. The scheduler can dispatch up to eight micro-operations every cycle, one on each port. Of the four ports providing computational resources, each provides an ALU, two of these execution pipes provided dedicated fused multiply-add (FMA) units. With the exception of division/square-root, most floating-point and integer SIMD execution units are 256-bit wide. The four dispatch ports servicing memory operations includes two dual-use ports for load and store-address operation, a dedicated 3rd store-address port, and one dedicated store-data port. All memory ports can handle 256-bit memory micro-operations. Peak floating-point throughput, at 32 single-precision operations per cycle and 16 double-precision operations per cycle using FMA, is twice that of Sandy Bridge. The out-of-order engine can handle 192 micro-operations in flight compared to 168 in Sandy Bridge.

The cache hierarchy is like prior generations, including an instruction cache, a first-level data cache and a second-level unified cache in each core, and a third-level unified cache with size dependent on specific product configuration. The third-level cache is organized as multiple cache slices, the size of each slice may depend on product configurations, connected by a ring interconnect. The L1 data cache can handle two 256-bit load and one 256-bit store operations each cycle. The unified L2 can service one cache line (64 bytes) for each cycle. Additionally, there are 72 load buffers and 42 store buffers available to support micro-operation executions in-flight.

3.2.3 Intel Skylake Microarchitecture

The latest iteration of architectural improvement comes in the form of the Skylake architecture. Released in 2015, Skylake is the successor to Broadwell (successor of Haswell Microarchitecture) in terms of technology node and is built on the advancements in the Haswell architecture. Skylake was a “tock” in Intel’s cycle, hence it used

the same 14-nm technology node used in Broadwell with refinements. Skylake micro-architecture chips are commonly known as sixth-generation processors. Some of the major changes include larger internal buffers to enable deeper out-of-order (OOO) execution and higher cache bandwidth. It has improved front-end throughput, branch predictor, and lower power consumption. Figure 3.6 gives the CPU core block diagram of the Skylake microarchitecture.

The front-end in the Skylake microarchitecture has the following enhancements over previous generations. The Legacy Decode Pipeline delivers 5 micro-operations per cycle to the IDQ compared to 4 micro-operations delivered in previous generations. The distributed shared buffer (DSB) delivers 6 micro-operations to the IDQ instead of 4 micro-operations in previous generations. The IDQ can hold 64 micro-operations per logical processor vs. 28 micro-operations per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 micro-operations (64 vs. 56 micro-operations in ST operation). The LSD in the IDQ can detect loops up to 64 micro-operations per logical processor irrespective single-threaded (ST) or simultaneous multi-threaded (SMT) operation.

The out-of-order (OOO) and execution engine changes in Skylake microarchitecture include larger buffers to enable deeper OOO execution compared to previous generations, improved throughput and latency for divide/sqrt and approximate reciprocals, identical latency and throughput for all operations running on FMA units. Longer pause latency enables better power efficiency and better SMT performance resource utilization.

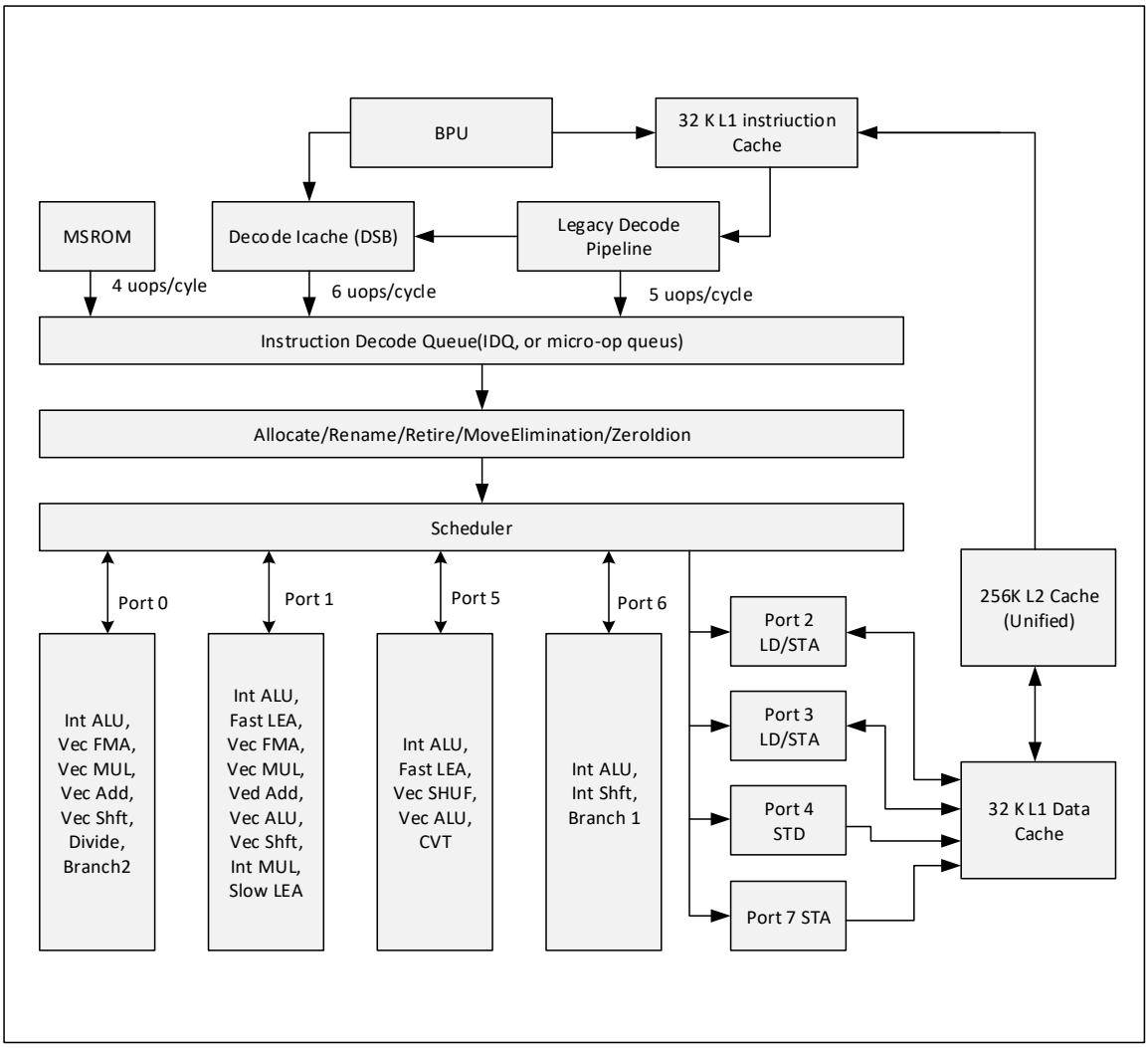

Figure 3.6 Skylake Microarchitecture CPU Core Block Diagram [6]

The cache hierarchy of the Skylake microarchitecture has the following enhancements; higher cache bandwidth compared to previous generations, simultaneous handling of more loads and stores enabled by enlarged buffers, L3 write bandwidth increased from 4 cycles per line in the previous generation to 2 per line, and L2 associativity changed from 8 way to 4 way.

3.2.4 Intel Kaby Lake

The Intel Kaby Lake processors were released in 2016 and are the successor to the Skylake microarchitecture. The Kaby Lake lineup is commonly known as seventh-generation core processors. As it was a “tick” it was expected to have a new technology node. However, due to complications in manufacturing, the old 14-nm technology node was used with refinements calling it “14-nm+”, signifying a break in the “tick-tock” convention. As the refined process resulted in the decrease in leakage current of 12%, it allowed the clock frequency of the cores to be pushed up by around 200 MHz in turbo mode. Overall, the internal architecture and memory system remained the same as Skylake.

3.2.5 Intel Coffee Lake

The Intel Coffee Lake processors were released in 2017 and are the successor to the Kaby Lake processors. The Coffee Lake lineup is commonly known as eighth-generation core processors. This is the second generation of processors that do not involve major changes in either microarchitecture or technology node. A further refined process called “14nm++” was used, resulting in a 21.5% decrease in leakage current compared to the 14nm process. As a result, the clock frequency was an impressive 4.70 GHz at single core turbo boost. The core count was also increased from four to six with parallelization in mind for the Core i7-8700K. The last level cache (L3) was bumped up to 12 MB to scale for the increase in core count. The general architecture and memory structure and the interconnect remain the same as the Skylake microarchitecture. Figure 3.7 shows the die map of a Coffee Lake i7-8700K processor.

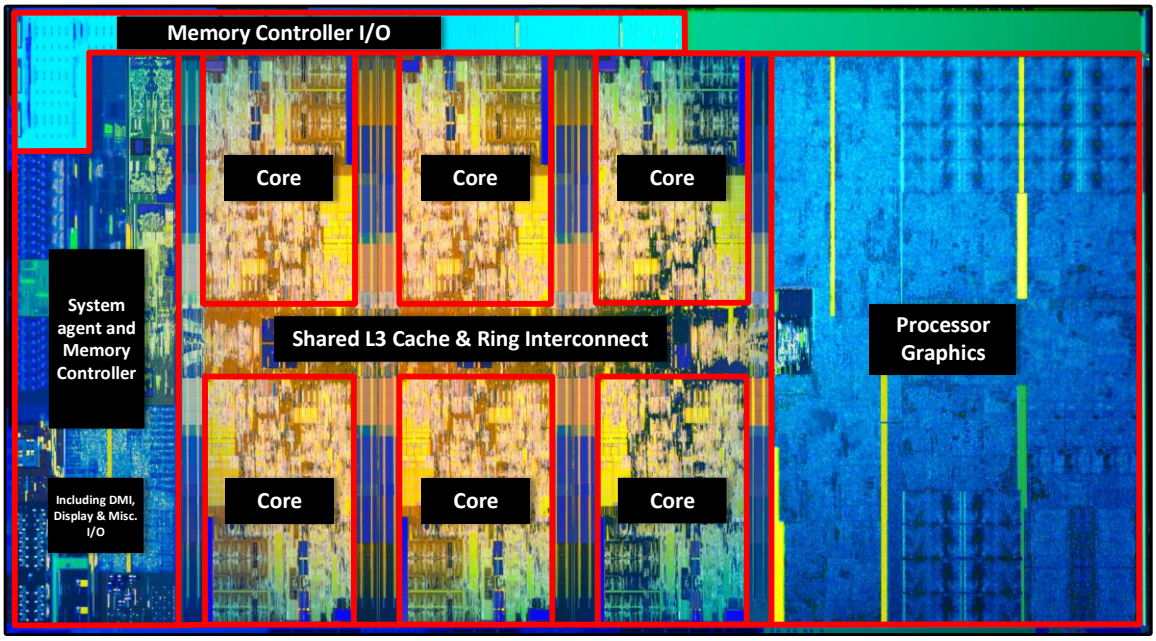

Figure 3.7 Die Map of a Hexa-Core Coffee Lake Processor

One of the major changes comes in the form of the Intel Turbo Boost Technology 2.0. Turbo mode accelerates processor and onboard graphics performance for peak loads, automatically allowing processor cores to run faster than the rated operating frequency if they're operating below power, current, and temperature specification limits. Whether the processor enters turbo mode and the amount of time the processor spends in that state depends on the workload and operating environment. With lower leakage current, the core frequency can be increased resulting in a larger frequency allocation for turbo mode. Depending on the workload, the turbo mode allocates turbo bins to one or multiple cores as shown in Figure 3.8.

Consider a Core i7-8700K which has 6 physical cores with a base frequency of 3.70 GHz. If the workload executes only on a single core, then all the turbo bins are allocated to that one core pushing the frequency to 4.7 GHz. If a workload occupies two physical cores, then the turbo bins are equally allocated resulting in 4.6 GHz for

those two cores. All 6 cores can go into turbo mode with core frequencies maxing out at about 4.3 GHz. Note that turbo mode is a function of the operating thermal range which determines the duration of turbo operation. As 8th generation processors are by default unlocked, it is possible to overclock frequency over the turbo mode.

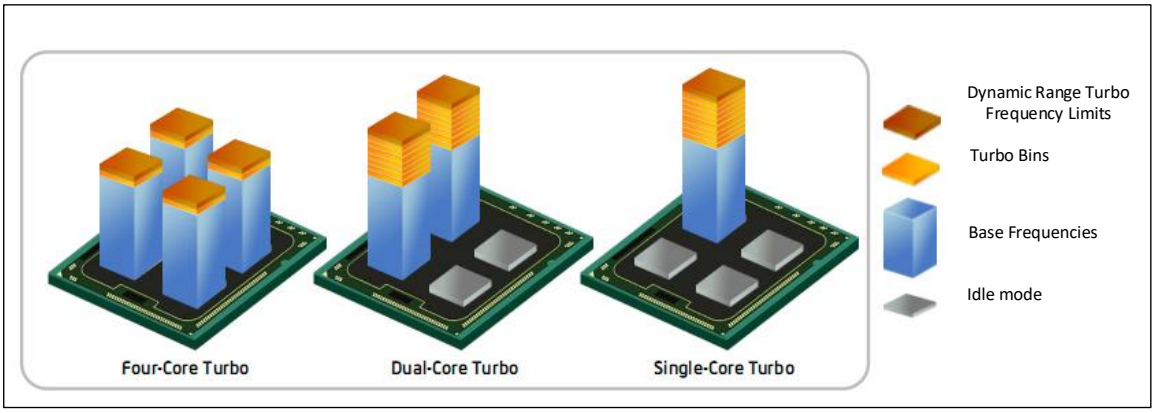

Figure 3.8 Turbo-Bin Allocation in Multi-Cores [13]

3.3 Systems under Test

The selection of test systems representing various architecture and computing class is discussed in this section. As the processor industry is dominated by the amd-x86 instruction set in desktop and server computing, this study is oriented towards analysis of x86 based systems. This study involves an Intel 4th generation, Haswell based Core-i7-4770 (LaCASA codename *mtsano*) and an Intel 8th generation, Coffee Lake based Core-i7-8700k (LaCASA codename *clingmansdome*) representing desktop computing. This pair of desktop machines gives us an opportunity to evaluate the performance improvements achieved over time. The second set consists of an entry-level version 2, Ivy Bridge based Xeon-E3-1240 (LaCASA codename *vidrak*) and a version

3, Haswell based Xeon-E5-2643 (LaCASA codename *mtleconte*) representing workstation/server platforms.

Table 3.5 shows the test systems chosen along with their hardware and software parameters. As the i7-4770 and Xeon-E3-1240 V2 are both quad cores with a comparable price point, the performance differences can be studied. The same kind of comparison can be made between the i7-8700K and the Xeon-E5-2643 V3 as both are hexa-cores with similar specifications. To maintain uniformity across the test platforms, factors such as the software versions and tool releases are set to the latest stable version on all the test systems, allowing for a hardware-oriented study. The same release of Linux is used across all devices used in this study. All the test systems have sufficient power and cooling requirements to ensure optimal performance capabilities. The SPEC CPU2017 benchmark suites are used to evaluate performance. The SPEC number acts as a reference point for all further insights. The choice of a compiler correlates with performance. A hardware-aware compiler performs better by optimizing the code to exploit all the hardware features such as deep pipelining, multi-level cache hierarchy, branch predictors, out-of-order execution engines, and advanced floating point and multimedia units [14]. The Intel compiler is chosen as the standard compiler across all test machines.

With respect to SPEC runs, a configuration file with similar performance optimization flags is used across all platforms. The benchmarks are compiled using the Intel-2018 compiler with required tuning flags. Benchmarks are built on each system and run according to system resources available. The native frequency of the chips is not altered, allowing the frequency governor to change frequency as required. Apart

from the operating system, no other tasks are run during testing to avoid discrepancies in benchmark runs.

Table 3.5 Systems Under Test [15] [16] [17] [10]

Sys. Code name	*mtsano*	*clingmansdome*	*vidrak*	*mtleconte*
Processor	Core i7-4770	Core i7-8700K	Xeon E3-1240 v2	Xeon E5-2643 v3
Lithography	22nm	14nm	22nm	22nm
Generation	4th Generation	8th Generation	3th Generation	4th Generation
Intel Codename	Haswell	Coffee-Lake	Ivy Bridge	Haswell
Year of release	Q2-2013	Q4-2017	Q2-2012	Q3-2014
Physical Cores	4	6	4	6*2
Threads / Core	2	2	2	2
Logical Cores	8	12	8	12*2
CPU Max Freq.	3.9 GHz	4.70 GHz	3.8 GHz	3.7 GHz
CPU Avg. Freq.	3.4 GHz	3.7 GHz	3.4 GHz	3.4 GHz
CPU Min Freq.	0.8 GHz	0.8 GHz	1.6 GHz	1.2 GHz
L1d cache	32K*4	32K*6	32K*4	32K*6*2
L1i cache	32K*4	32K*6	32K*4	32K*6*2
L2 Cache	256K*4	256K*6	256K*4	256K*6*2
L3 Cache (LLC)	8192K	12288K	8192K	20480K
RAM	16 GB	32 GB	16GB	64 GB
RAM Freq.	1600 MHz	2400 MHz	1600 MHz	2400 MHz
TDP (watts)	84 W	95 W	69 W	135 W
Operating System	Ubuntu 16.04.4 LTS	Ubuntu 16.04.4 LTS	Ubuntu 16.04.4 LTS	Ubuntu 16.04.4 LTS
OS Codename	Xenial	Xenial	Xenial	Xenial
Kernel	4.13.0-37-generic	4.4.0-116-generic	4.4.0-116-generic	4.4.0-116-generic
icc version	18.0.1	18.0.1	18.0.1	18.0.1
Perf version	4.13.13	4.4.98	4.4.98	4.4.98

3.4 Tools and Applications

A SPEC CPU2017 run generates results numbers that represent the total execution time relative to the baseline machine as a single number. These results give a quick representation of the system's performance but for an in-depth analysis, additional tools are required. The following subsection discusses additional tools used for analysis.

3.4.1 Linux perf

Modern processors have dedicated hardware counters for performance monitoring. Linux *perf* is a profiler tool present in all Linux-based systems after kernel version 2.6. It abstracts the hardware differences and provides a simple command line interface. *Perf* utilizes the *perf_events* interface exported by recent versions of the Linux kernel. It supports a list of measurable events. The tool and underlying kernel interface can measure events coming from various sources. Some events come from the processor itself and its Performance Monitoring Unit (PMU). *perf* provides a list of events to measure micro-architectural events such as the number of clock cycles, instructions retired, L1 cache misses and so on. Those events are called PMU hardware events or hardware events for short. They vary with each processor type and model. Other events are counted using Linux kernel counters, and they are thus called software events.

The *perf* tool can be used to count events on a per-thread, per-process, per-CPU or system-wide basis. In the per-thread mode, the counter only monitors the execution of a designated thread. When the thread is swapped out, monitoring stops. When a thread migrates from one processor to another, counters are saved for the current processor and are restored for the new one [19]. The per-process mode is a variant of

per-thread where all threads of the process are monitored. Counts and samples are aggregated at the process level. The *perf_events* interface allows for automatic inheritance on fork() and pthread_create(). By default, the Linux *perf* tool activates inheritance and counts all threads of a process and subsequent child processes and threads. *Perf Hardware Event* indicates one of the "generalized" hardware events provided by the kernel. *Perf Software Event* indicates one of the software-defined events provided by the kernel (even if no hardware support is available). *Perf Hardware Cache Event* indicates a hardware cache event that has a special encoding, described in the config field definition.

The *perf_events* interface also provides a small set of common hardware event monitors. On each processor, those events get mapped onto an actual event provided by the CPU, if they exist, otherwise, the event cannot be used. Every processor has hardware counters built into it and some are event-specific registers. The user can use the available general-purpose registers to count user defined-events. Most events are model specific except the architectural performance events shown in Table 3.6. The parameters shown in the table are common for all the test systems used for the study. Table 3.6 shows hardware perf events mapping details to actual Performance Monitoring Unit events, their function, and register numbers and umask. umask and event number together help map a particular event to the registers.

Furthermore, many events can be derived by software using available hardware counters. *perf_events* are an event-oriented observatory tool which is extremely useful in solving performance issues and in troubleshooting. Many tools further discussed use *perf* in the lower level to generate reports and observations.

Table 3.6 Architectural Performance Events [20]

Perf-event	Event Num	Umask Value	Event Mask Mnemonic	Definition	Description & Comment
instructions	C0H	00H	INST_RETIRED.ANY	The number of instructions at retirement.	Counts when the last micro-operation of an instruction retires.
cpu-cycles	3CH	00H	CPU_CLK_UNHALTED.THREAD_ P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it executes a HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
branch-instructions	C4H	00H	BR_INST_RETIRED.ALL_BRANC HES	Branch instructions at retirement.	Counts when the last micro-operation of a branch instruction retires.
branch-misses	C5H	00H	BR_MISP_RETIRED.ALL_BRANC HES	The number of mispredicted branch instructions at retirement.	Counts when the last micro-operation of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.
cache-misses	2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	Accesses to the LLC in which the data is not present (miss).
cache-references	2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	Accesses to the LLC, in which the data is present (hit) or not present (miss).
L1-dcache-loads	D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	All retired memory read instructions
L1-dcache-stores	D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	All retired memory writes instructions

3.4.2 Likwid

Likwid (Like I Knew What I'm Doing) is a set of lightweight performance tools that incorporates easy to use command line tools for Linux to help programmers in developing high-performance applications. *Likwid* has a set of tools for a specific purpose. Some of the tools are confined to x86 processors. The tools can be roughly grouped into three categories such as system information and control, performance and energy profiling and micro-benchmarking. Some of the tools used in this study are explained further. *likwid-features* can display and alter the state of the on-chip hardware prefetching units in Intel x86 processors. *likwid-topology* probes the hardware thread and cache topology in multicore, multilocked nodes. *likwid-perfctr* measures performance counter metrics over the complete runtime of an application or, with support from a simple application programming interface (API), between arbitrary points in the code. Counter multiplexing allows the concurrent measurement of many metrics, larger than the (usually small) number of available counters. Although it is possible to specify the full, hardware-dependent event names, some predefined event sets simplify matters when standard information like memory bandwidth or floating-point operations (FLOP) counts is needed. *likwid-pin* enforces thread-core affinity in a multithreaded application "from the outside," i.e., without changing the source code. It works with all threading models that are based on POSIX threads and is also compatible with hybrid "MPI+threads" programming. Sensible use of *likwid-pin* requires correct information about thread numbering and cache topology, which can be delivered by *likwid-topology* [21].

3.4.3 Intel VTune Amplifier

The *Intel VTune Amplifier* is a performance analysis tool that relies on the underlying hardware counters to get run-time parameters of the application under test. It can be used to locate or determine the following aspects of the code and system:

- The most time-consuming functions or hot-spots in the application.
- Sections of code that do not effectively utilize the available processor time.
- The best sections of code to optimize for sequential performance and for threaded performance.
- Synchronization objects that affect the application performance.
- Hardware-related issues in code such as data sharing, cache misses, branch misprediction, and others.
- The performance impact of different synchronization methods, different numbers of threads, or different algorithms.
- Thread activity and transitions such as migrations and context-switches.

For this study, four key features of *Intel VTune Amplifier* are used; Advanced Hotspots, HPC Performance Characterization, Memory Access Analysis and General Exploration. When the number of events exceeds the available counters, the tool multiplexes events and sampling is incorporated. The MUX reliability should be noted. If the reliability is less than 70%, then the results are not to be considered acceptable [22].

Advanced Hotspot analysis is a fast and straightforward way to identify performance-critical code sections in a given application. The periodic instruction pointer sampling performed by *Intel VTune Amplifier* identifies code locations where an application spends the most time. It creates a list of functions in the application ordered

by the amount of time spent in each function. By default, Advanced Hotspots analysis does not capture the function call stacks as the hotspots are collected, but it can be used to sample all processes on the system. This type of analysis uses event-based sampling collection and analyzes all the processes running on the system at the time, providing CPU time data on whole system performance.

HPC Performance Characterization analysis is used to identify how effectively a compute-intensive application uses CPU, memory, and floating-point operation hardware resources. The HPC Performance Characterization analysis type can be used as a starting point for understanding the performance aspects of an application. During HPC Performance Characterization analysis, the data collector profiles the application using event-based sampling collection.

Memory Access analysis is used to identify memory-related issues, like non-uniform memory access (NUMA) problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures). This attribution is possible due to instrumentation of memory allocations/de-allocations and getting static/global variables from the symbol information. Memory Access analysis type uses hardware event-based sampling to collect data.

General Exploration analysis is used to understand how efficiently the code passes through the core pipeline. During General Exploration analysis, the *Intel VTune Amplifier* collects a complete list of events for analyzing a typical client application. It calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems. The General Exploration analysis strategy varies by microarchitecture. For modern microarchitectures starting with Ivy Bridge, the General Exploration analysis is based on the Top-down Microarchitecture

Analysis Method (TMAM) using the Top-down Characterization (TCM) methodology. TCM is a hierarchical organization of event-based metrics that identify the dominant performance bottlenecks in an application. Superscalar processors can be conceptually divided into the front-end and the back-end. The front-end is where instructions are fetched and decoded into the operations that constitute them. The back-end is where the required computation is performed.

Each cycle, the front-end generates up to four of these operations. It places them into pipeline slots that then move through the back-end. Thus, for a given execution duration in clock cycles, it is easy to determine the maximum number of pipeline slots containing useful work that can be retired in that duration. The actual number of *Retiring* pipeline slots containing useful work and rarely equals this maximum.

Underutilization can be due to several factors. Pipeline slots may not be filled with useful work, either because the front-end could not fetch or decode instructions in time (*Front-End Bound*) or because the back-end was not prepared to accept more operations of a certain kind (*Back-End Bound*). Moreover, even pipeline slots that do contain useful work may not retire due to bad speculation. *Front-End Bound* stalls may be due to a large code working set, poor code layout, or microcode assists. *Back-End Bound* stalls may be due to long-latency operations or other contention for execution resources. *Bad Speculation* occurs most frequently due to branch misprediction.

Each cycle, each core can fill up to four of its pipeline slots with useful operations. Therefore, for any time interval, it is possible to determine the maximum number of pipeline slots that could have been filled in and issued. This analysis performs this estimate and breaks up all pipeline slots into four categories:

- Pipeline slots containing useful work that issued and retired (*Retired*).
- Pipeline slots containing useful work that issued and canceled (*Bad Speculation*).
- Pipeline slots that could not be filled with useful work due to problems in the front-end (*Front-End Bound*).
- Pipeline slots that could not be filled with useful work due to a backup in the back-end (*Back-End Bound*).

CHAPTER 4

BASELINE SPEC EVALUATION

This chapter gives a baseline SPEC evaluation and explains the conditions used. Section 4.1 explains the Top-down analysis method used by *Intel VTune Amplifier*. Section 4.2 gives a theoretical view of implementing thread affinity. Section 4.3 gives a contrast between assigned thread affinity and OS scheduling and the conditions for the final compiled executable that will be used for all further measurements.

4.1 Top-down Microarchitectural Analysis Method

This section describes Intel's *Top-down Microarchitecture Analysis Method*, or TMAM for short, for identifying performance bottlenecks in out-of-order processor cores. This method simplifies the process of identifying and quantifying performance bottlenecks. The use of TMAM abstracts the steep learning curve associated with each microarchitecture generation and a myriad of hardware events that can be observed through PMUs during program execution and replaces it with a bird's eye view of true performance limiters [23].

TMAM utilizes the concept of pipeline slots as shown in Figure 4.1. A pipeline slot represents a hardware resource needed to process one micro-operation. A CPU core offers multiple pipeline slots that can be utilized by micro-operations in each clock cycle. The number of slots is called pipeline width. Figure 4.1 illustrates a 4-wide CPU that executes code for 10 clock cycles. Thus, there are 40 pipeline slots in total (4 * 10). All green circles represent a micro-operation retiring in the given slots, thus the

utilization is 100%. On the other side, Figure 4.2 shows an example where 20 out of 40 slots are stalled, i.e., they did not retire any micro-operation. This means that the code efficiency is only 50%:

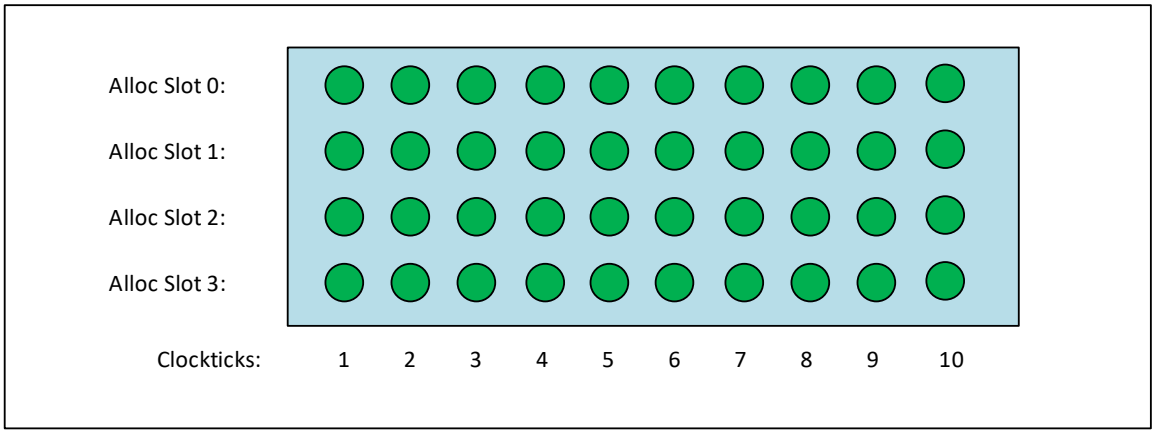

Figure 4.1 Pipeline Slots, 100% Utilization

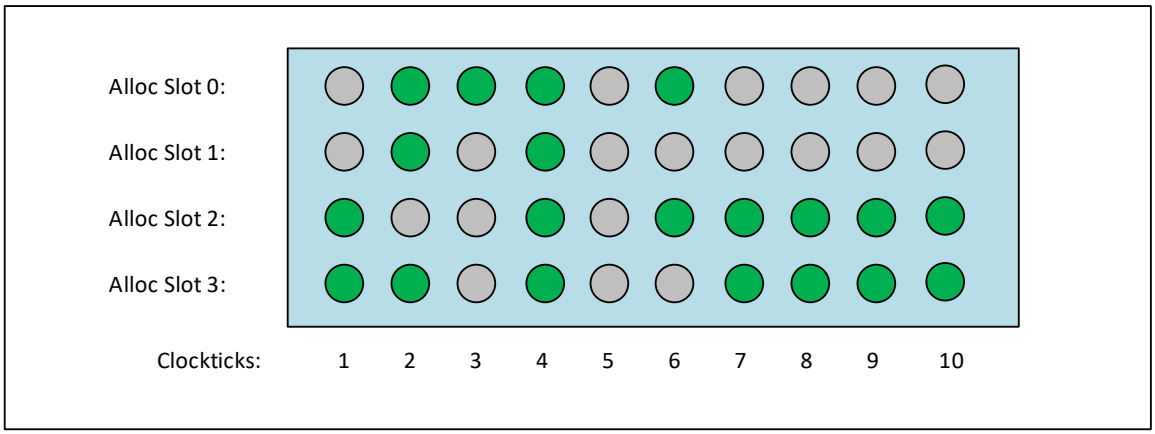

Figure 4.2 Pipeline Slots, 50% Utilization

Figure 4.3 shows the general hierarchical framework of TMAM. Figure 4.4 shows the top-level flowchart of top-down microarchitectural analysis. At the top level, pipeline slots are classified into four main categories: (a) *Front-End Bound*, (b) *Back-End Bound*, (c) *Bad Speculation* and (d) *Retiring*. If a slot is utilized by an operation, it will be classified as either *Retiring* or *Bad Speculation*, depending on whether the

micro-operation eventually gets retired (committed). Empty or stalled slots are classified as *Back-End Bound* if the back-end portion of the pipeline is unable to accept more operations (back-end stall), or *Front-End Bound* if there are no micro-operations to be delivered to the back-end.

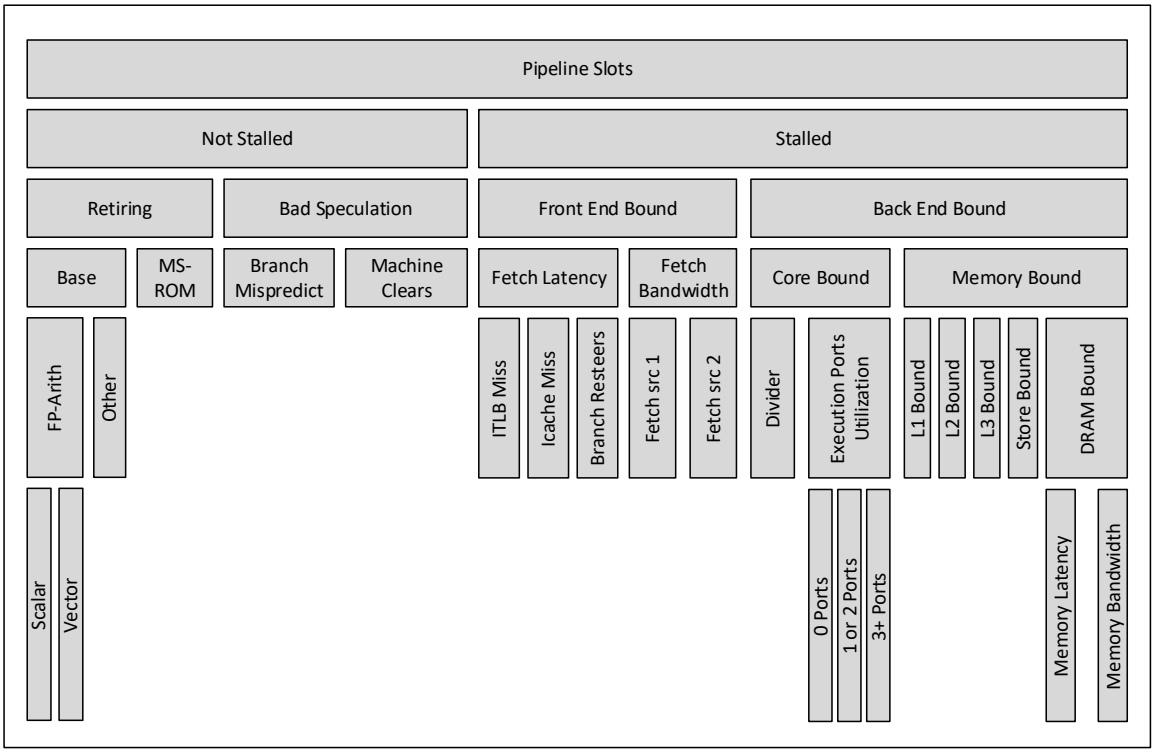

Figure 4.3 General Top-Down Microarchitecture Analysis Method [6]

Retiring denotes the slots utilized by useful micro-operations. The number of slots retiring in a given clock cycle directly correlates to the instructions per cycle (IPC) metric – the ideal performance is reached when 100% slots are retiring. Naturally, this is hard to achieve. If the percentage of *Retiring* slots reaches 50% in a 4-way superscalar processor, the average IPC=2. It is important to note that a high

retiring fraction does not mean there is no room for speedup. For example, the performance of a non-vectorized code with a high *Retiring* fraction can still be improved by using vectorized instructions.

Bad Speculation denotes the slots wasted due to all aspects of incorrect speculations. It includes (a) *Branch Mispredicts*, and (b) *Machine Clears*. *Branch Mispredicts* and *Machine Clears* cover control-flow and data-speculation, respectively.

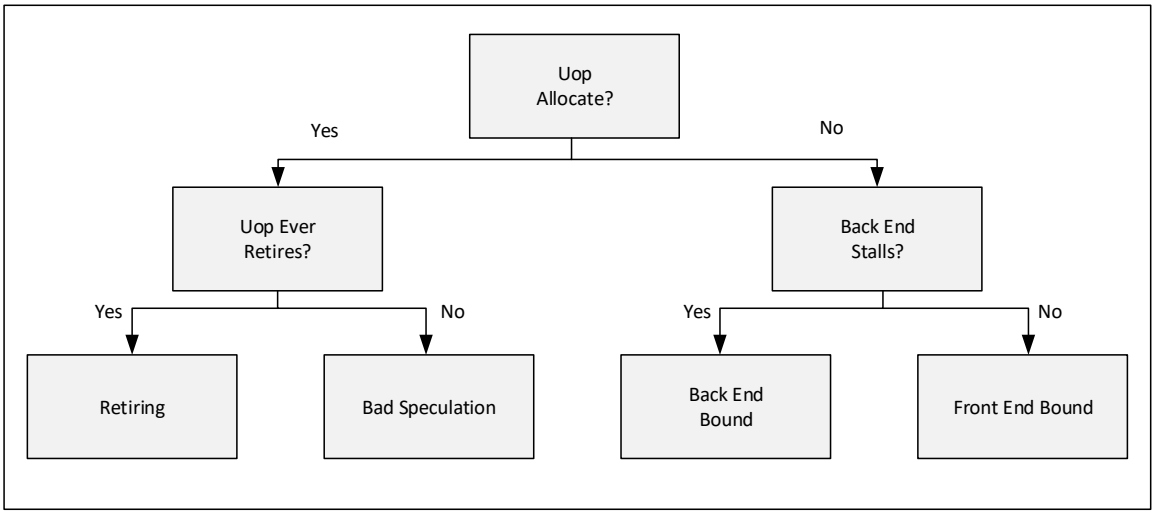

Figure 4.4 Top-Down Analysis Flowchart [6]

Front-End Bound denotes slots stalled due to the front-end's inability to supply the back-end. This category is further classified into Fetch Latency and Fetch Bandwidth. As the front-end stalls can have a detrimental impact on performance, many dedicated units are added to minimize them, including the Loop Stream Detector (LSD) and Decoded I-cache (DSB). Fetch Latency accounts for cases that lead to fetching starvation (the symptom of no micro-operation delivery) regardless of the cause and includes ITLB misses, misses in caches caused by instruction fetches or branch resteers.

Back-End Bound denotes slots stalled due to adverse events in the back-end when no micro-operations are being delivered to the pipeline due to lack of resources or lack of data. *Back-End Bound* are split into *Memory Bound* and *Core Bound*. This is achieved by breaking down back-end stalls based on execution units' occupation at every cycle. *Memory Bound* corresponds to execution stalls related to the cache and memory subsystems. These stalls usually manifest with execution units getting starved after a short while, as in the case of a load missing all caches. The out-of-order scheduler can dispatch micro-operations into multiple execution units for execution. While these micro-operations are executing in-flight, some of the memory access latency exposure for data can be hidden by keeping the execution units busy with useful micro-operations that do not depend on pending memory accesses. *Core Bound* corresponds to pressure on the execution units or lack of Instruction-Level-Parallelism (ILP) in the program. *Core Bound* stalls can manifest either with short execution starvation periods or with sub-optimal execution port utilization, which makes it more difficult to identify. *Core Bound* issues often can be mitigated with better code generation. For example, a sequence of dependent arithmetic operations would be classified as *Core Bound*. A compiler may relieve this stall with better instruction scheduling. Vectorization can mitigate *Core Bound* issues as well.

4.2 Thread Affinity

Thread affinity, or thread pinning, enables binding and unbinding of a process or thread to the processor core or a range of processor cores. Affinity can be user assigned or OS-scheduled. OS-scheduled processes tend to migrate between processors

depending on priority and resource availability. Thread affinity is used to avoid the migration overhead associated with OS-scheduling.

For thread affinity, two affinity types are used in the configuration files, accounting for different hardware parameters such as processor count. Specifying *compact* assigns OpenMP thread with index <*n*>+1 to a free thread context as close as possible to the thread context with index <*n*>. This is achieved using *KMP_AFFINITY=compact*, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized for applications exhibiting data sharing between consecutive iterations of loops. Figure 4.5 illustrates this strategy of using *KMP_AFFINITY=granularity=fine, compact,1,0* as a setting [24].

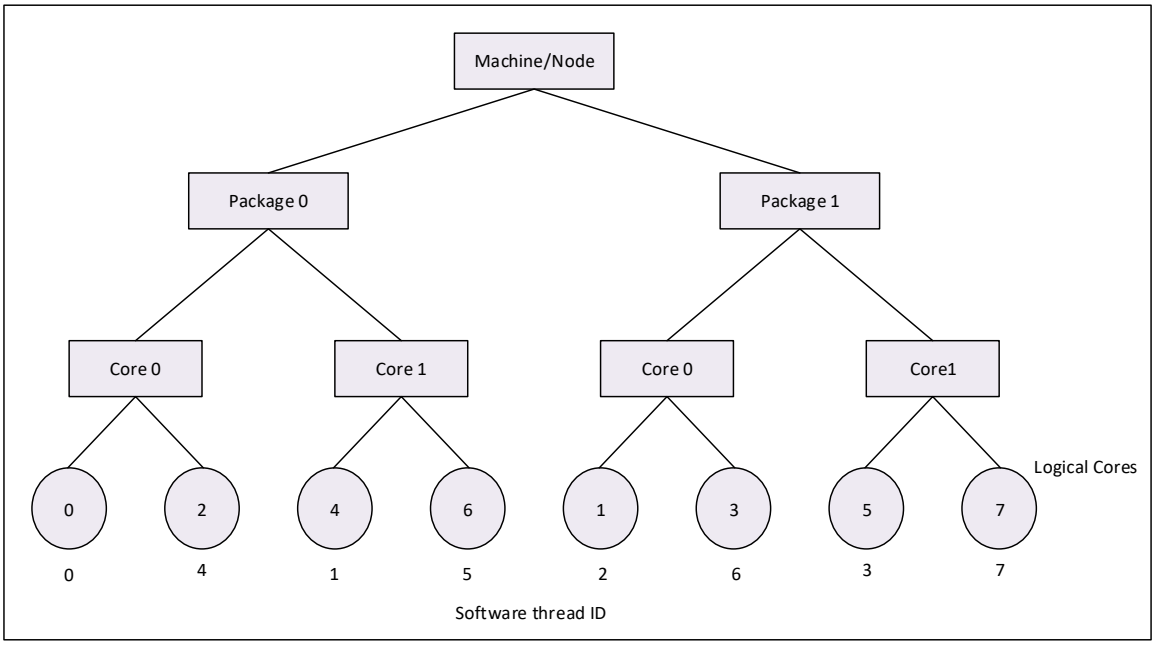

Figure 4.5 Software Thread Assignment on Logical Cores for *compact 1,0* [24]

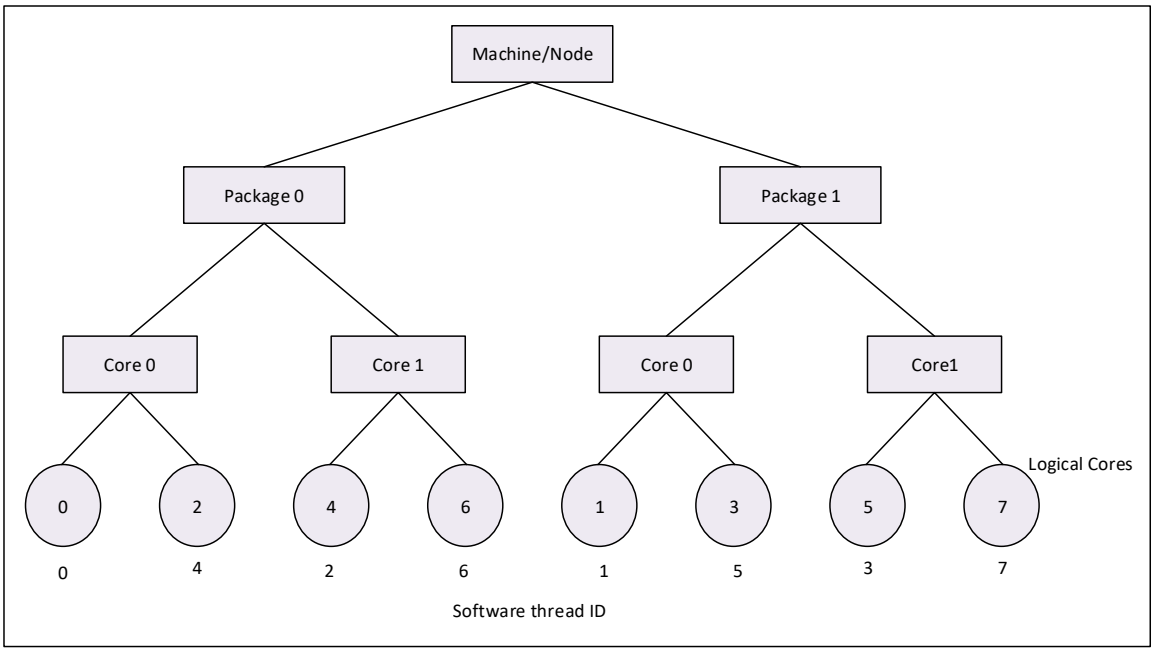

Figure 4.6 Software Thread Assignment on Logical Cores for *scatter* [24]

Specifying *scatter* distributes the threads as evenly as possible across the entire system which is the opposite of *compact*. Suppose the number of created software threads for an application is less than the logical cores available for the processor. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, as a thread would execute faster on a core with less competition for resources [24]. For independent threads with no communication overhead, the use of scatter improves utilization. Figure 4.6 illustrates thread affinity after specifying *KMP_AFFINITY=granularity=fine, scatter*. Note that "*fine*" is a switch for granularity which is interchangeable with "*thread*".

The implementation of thread affinity in speed benchmarks is done as shown in Figure 4.7. The integer speed benchmarks are set to the *fine*, *scatter*, as most of the benchmarks are not multithreaded and the floating-point speed benchmarks are set

to *fine*, *compact*, *1,0* for hyper-threaded systems. All the test systems in this study support hyper-threading and simultaneous multithreading is enabled by default.

```
%ifdef %{intspeedaffinity}
    preENV_KMP_AFFINITY    = granularity=fine, scatter
%else
    %if defined(%{smt-on})
        preENV_KMP_AFFINITY    = granularity=fine, compact,1,0
    %else
        preENV_KMP_AFFINITY    = granularity=fine, compact
    %endif
 %endif
```

Figure 4.7 Use of Affinity in Speed Benchmarks

The implementation of thread affinity in rate benchmarks is done as shown in Figure 4.8. As rate benchmarks run multiple independent copies/processes, there are not any data dependencies across them. With that in mind, copies are pinned based on topology. Different switches are used for NUMA machines.

```
default:
submit=numactl --localalloc --physcpubind=$SPECCOPYNUM --$command

%ifdef %{no-numa}
   submit = taskset -c $SPECCOPYNUM $command
%endif
```

Figure 4.8 Use of Affinity in Rate Benchmarks

4.3 Baseline Evaluation

To select one set of baseline executables for further analysis, two variants of SPEC runs are tested for this study. Keeping all other parameters, the same, one configuration file include thread affinity as described above and another configuration

file does not specify any affinity leaving the OS scheduler in charge of thread affinity. The SPEC run yielding better performance is chosen for all further analysis. Next, to determine the scalability of the benchmark suite, the speed benchmarks are run with multiple threads and rate benchmarks are run with multiple copies. To evaluate the difference in performance with thread affinity and OS-scheduling, all four suites of the SPEC CPU2017 were run on all the test systems. For quad-core machines 1, 4, 6 and 8 threads/copies were run. For hexa-core machines 1, 4, 6 and 12 threads/copies were run. For the machine with two hexa-cores 1, 4, 6, 8, 12, 16 and 24 threads/copies were run.

The SPEC CPU2017 run results obtained are discussed in this section. For simplicity in presentation, the results for Core processors and Xeon processors are shown separately.

4.3.1 SPEC CPU2017 Speed Benchmark Suites

Figure 4.9 shows the results for SPEC CPU2017 Speed benchmarks when run on machines with the Core i7 processors. The graph shows *fp_speed* and *int_speed* as a function of the number of threads. Initial observation suggests that the *fp_speed* benchmark suite does show good scalability – the performance increases as the thread count are increased. Peak performance is observed when the number of threads equals the number of physical cores, i.e., hyper-threading does not appear to increase overall performance. Using thread affinity shows no performance improvement for *fp_speed* benchmark suite; it actually degrades the performance for up to 25%. Hence OS-scheduling proves to be a better option than using thread affinity.

The *int_speed* benchmark suite shows minimal performance improvements as the thread count increases because the majority of the integer benchmarks are not

parallelizable. Consequently, it does not make much of a difference whether a single threaded application is pinned to one core or migrates to different cores.

Comparing the performance of the two Core processors, the 8th generation *cling-mansdome* (Core i7-8700K) outperforms the 4th generation *mtsano* (Core i7-4770) in every aspect. The performance improvements are significant for *fp_speed* but relatively small for *int_speed* benchmarks.

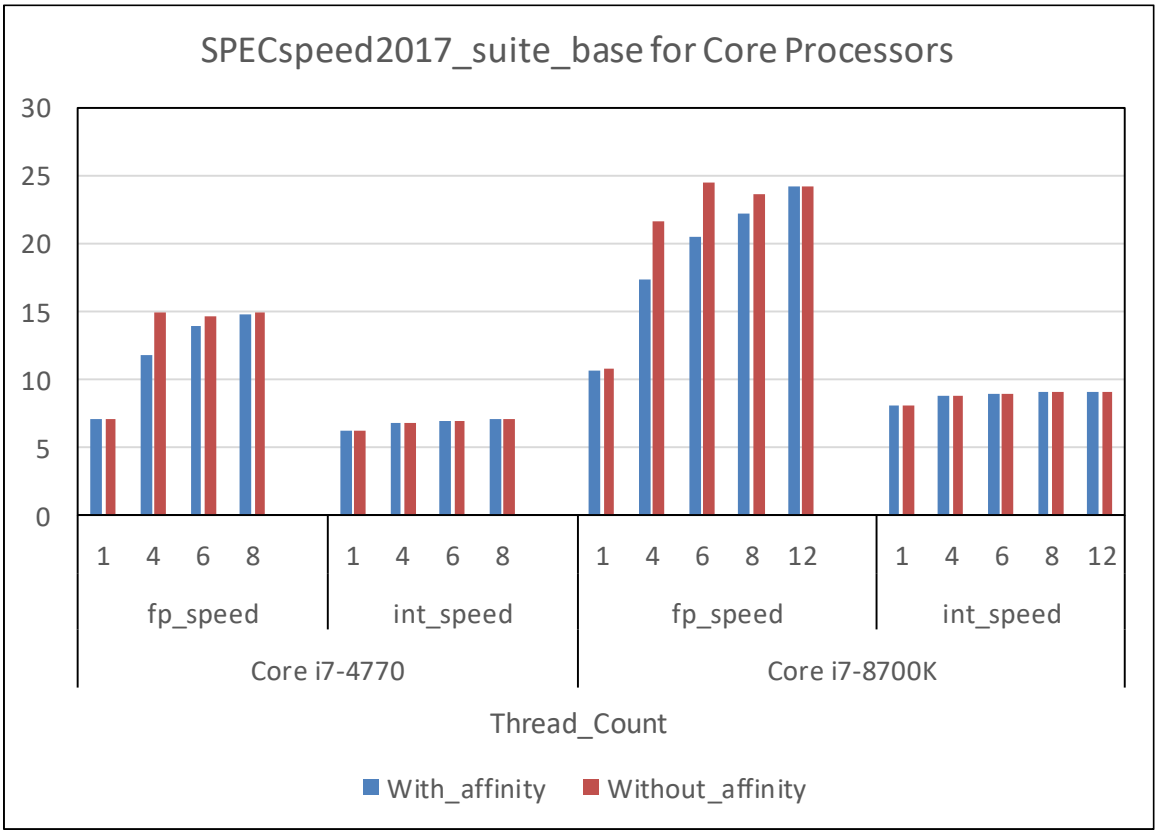

Figure 4.9 Baseline Evaluation of Speed benchmarks on Core Processors.

Figure 4.10 shows the results for SPEC CPU2017 Speed benchmarks when run on machines with Xeon processors. Similar to the trends observed on the Core processors, the performance scales well with the number of threads for *fp_speed* and it does

not scale for *int_speed*. Thread affinity plays a bigger role for multithreaded applications as inter-thread communication impacts performance.

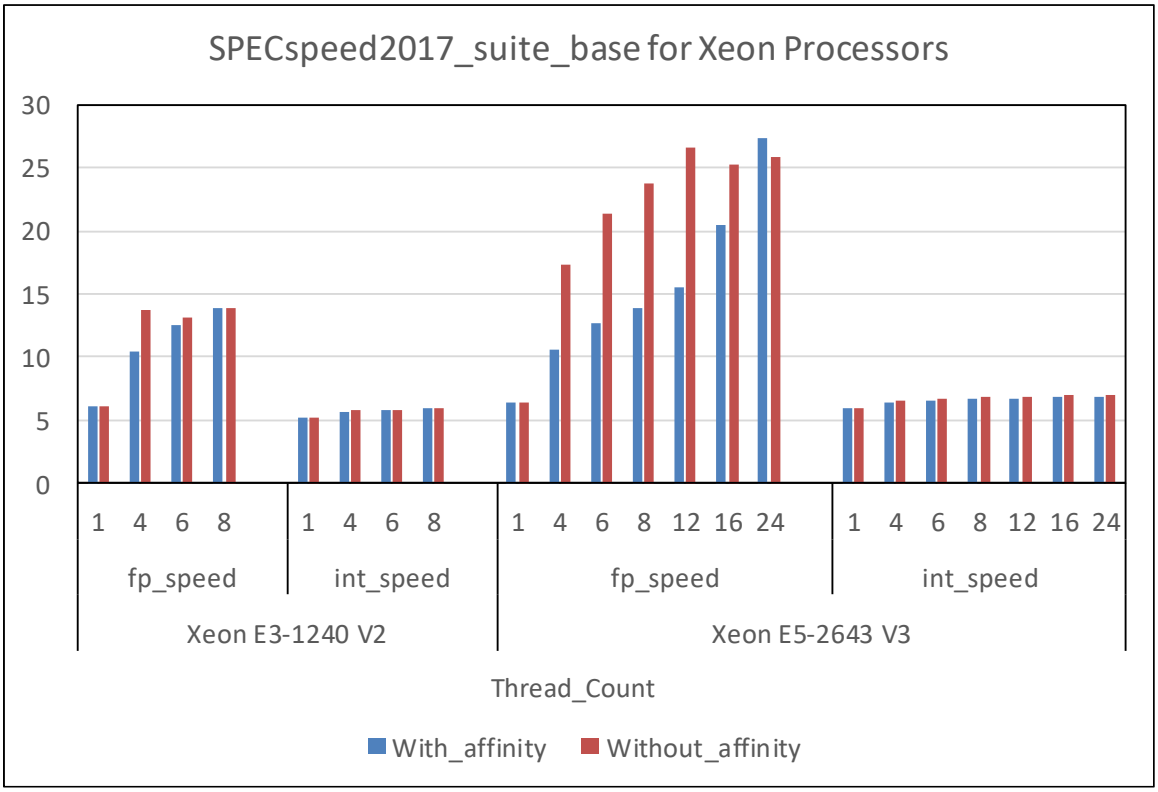

Figure 4.10 Baseline Evaluation of Speed Benchmarks on Xeon Processors

In the case where the parent thread requires data from a child thread that is pinned to another core, the latency increases, and performance decreases. The distance between the parent thread and the child threads for an OS-scheduled thread placement could vary dynamically with data requirements, causing performance improvements. It is clearly observed in *mtleconte* (Xeon E5-2643 V3) which houses two physical processors, where thread affinity pins the threads across processors and the communication overhead deteriorates performance for up to 70% when compared to dynamic OS-scheduling. This trend is noticeable when the number of created threads

in benchmarks is less than or equal to the physical cores. However, when the number of threads reaches the number of available logical cores, the degree of improvement become less noticeable as the communication gap becomes unavoidable. Hence, OS-scheduling shows performance degradation on *mtleconte* (Xeon E5-2643 V3) when the thread count matches the number of logical cores. The overhead of migration alongside the unavailability of resources takes a toll on performance.

4.3.2 SPEC CPU2017 Rate Benchmark Suites

Figure 4.11 shows the results for SPEC CPU2017 Rate benchmarks when running on machines with the Core i7 processors. The graph shows *fp_rate* and *int_rate* as a function of the number of copies. The rate benchmarks do show good scalability – the performance increases as the number of copies increase. Peak performance is observed when the number of copies matches the number of logical cores available, i.e., hyper-threading has minimal, if not no, effect on overall performance.

Using affinity to pin copies to logical cores shows no performance improvement for *fp_rate* benchmarks, in contrast, it degrades performance up to 9%. Hence OS-scheduling proves to be a better option. The *int_rate* benchmark suite shows a similar trend in performance. With the use of affinity, degradation of up to 10% is observed.

Comparing the performance of the two Core processors, the 8th generation *clingmansdome* (Core i7-8700K) outperforms the 4th generation *mtsano* (Core i7-4770). The performance improvements are significant when multiple copies are run for *fp_speed*. But with *int_speed* benchmarks, the difference is minimal as the clock frequency seem to be the deciding factor.

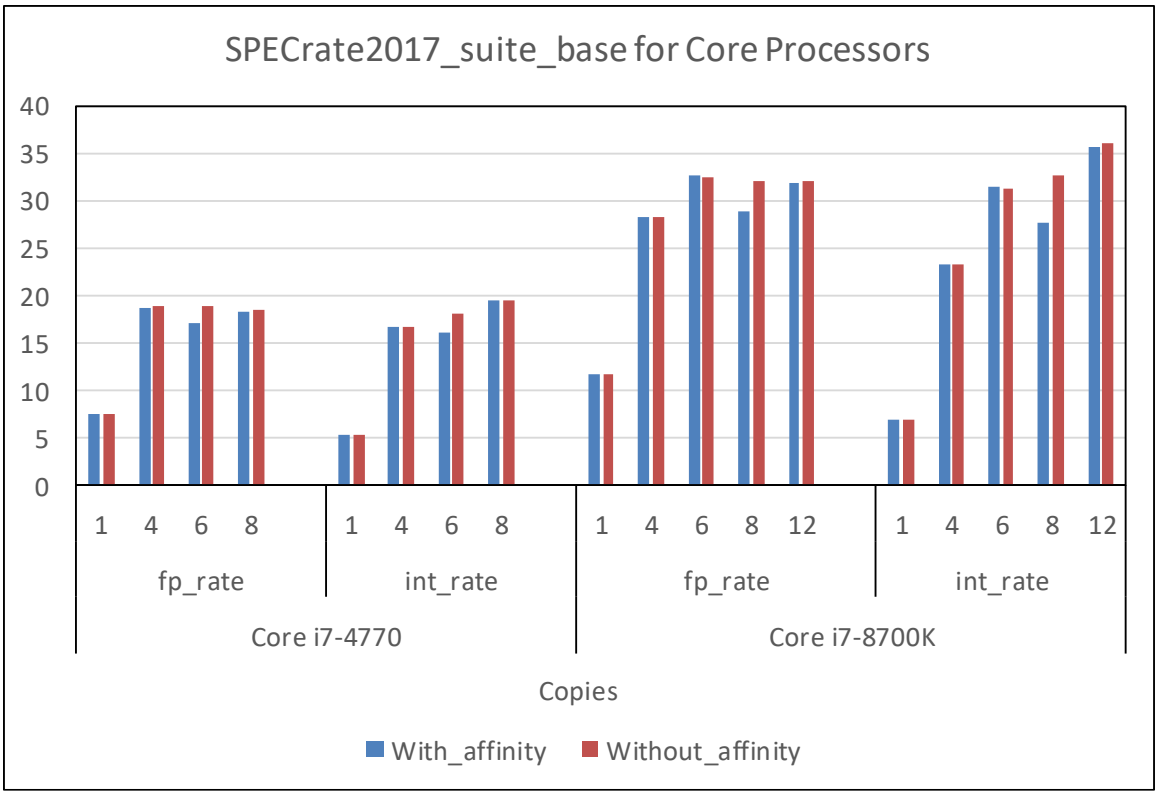

Figure 4.11 Baseline Evaluation of Rate benchmarks on Core Processors

Figure 4.12 shows the results for SPEC CPU2017 Rate benchmarks when run on machines with Xeon processors. Like the trends observed on Core processors, the performance scales well with the number of copies for both *fp_rate* and *int_rate*. Affinity plays a bigger role for multiple copies as the contention for hardware resources impact performance. In the case where two copies are pinned to the same physical core, the contention in shared resources degrades performance. Dynamic OS-scheduling can avoid this situation if the number of copies is less or equal to the number of physical cores. When the number of copies becomes more than the physical cores, then contention becomes unavoidable.

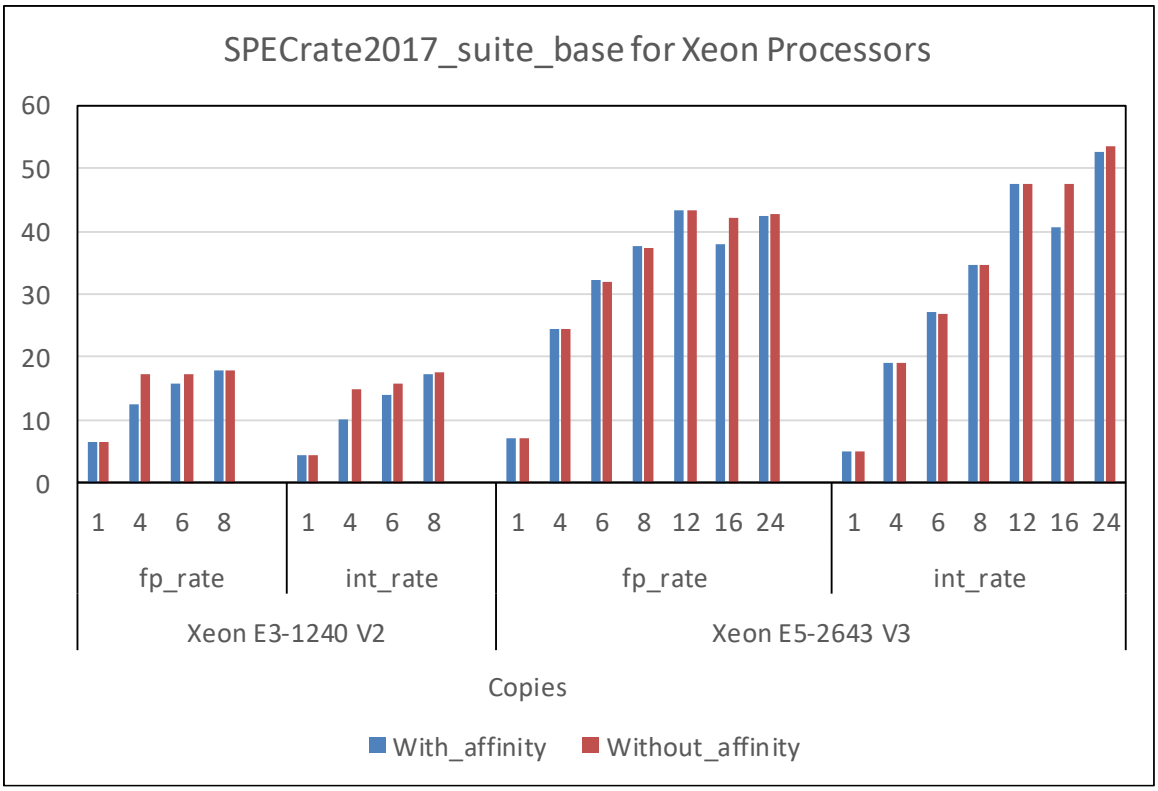

Figure 4.12 Baseline Evaluation of Rate benchmarks on Xeon Processors

CHAPTER 5

SPEC CPU2017 BENCHMARKS CHARACTERIZATION

This chapter gives a detailed view of characteristics of the SPEC CPU2017 benchmark suites collected while running them on *clingmansdome* – a machine that houses the latest 8th generation of Intel Core i7 processor (Core i7-8700K) with 6 physical cores and 12 logical cores. The SPEC CPU2017 Speed benchmarks are run with one, six, and twelve threads. The SPEC CPU2017 Rate benchmarks are run with one, six and twelve copies. To maintain uniformity and fairness, no other process apart from operating system processes are run on the machine during benchmark evaluation. The frequency governor used for the runs is on-demand power. Hyper-threading and hardware prefetching are enabled.

The benchmarks are represented by the assigned number followed by the name and suffix "_s" for speed or "_r" for rate benchmarks as shown in Table 2.2 and Table 2.3. If a SPEC benchmark execution involves multiple reference data inputs, statistics are collected for each run separately and a suffix starting with "_" followed by letters in alphabetical order indicates different benchmark runs. Tools described in section 3.4 are used to gain insights into the behavior of all the benchmarks. Section 5.1 describes the results for SPEC CPU2017 Speed benchmarks characterization. Section 5.2 describes the results for SPEC CPU2017 Rate benchmarks characterization.

5.1 SPEC CPU2017 Speed Benchmarks Characterization

The results for the SPEC CPU2017 characterization are divided into five subsections. Each subsection encompasses *fp_speed* and *int_speed* results in that order. Though the runs carried out were for 1,6 and 12 threads, results for single-thread and results that have significant differences from a single-thread execution are depicted here. Parameters that stay consistent with the change in the number of threads are not discussed. Execution times for multiple threads are discussed in CHAPTER 6.

5.1.1 General View of Benchmarks

The first set of experiments collects information about individual SPEC CPU2017 Speed benchmarks run under the Linux *perf* utility. Specifically, the following parameters are collected: total execution time, total number of clock cycles, total number of retired instructions, total number of retired branch instructions, total number of retired load instructions (instructions that read at least one operand from memory), and total number of retired store instructions (memory write instructions).

Table 5.1 shows the collected parameters for single-threaded *fp_speed* benchmarks. The first observation regarding the execution times (column titled Time) is that each benchmark run takes a considerable amount of time, ranging from 584 s (603.bwaves_s) to 4,260 s (638.imagick_s). Accordingly, the number of executed instructions ranges from 3.31 trillion (649.fotonik3d_s) to 69.14 trillion (638.imagick_s). The number of retired load/memory-read instructions range from 13.48 trillion (638.imagick_s) to 49.90 trillion (644.nab_s) and the number of retired store/memory-write instruction range from as low as 316.23 billion (603.bwaves_s_b) to 1.43 trillion (627.cam4_s). Please note that branches encompass all control-flow instructions, loads encompass all instructions that read at least one operand from memory, and stores

encompass instructions that write the result into the memory. If an instruction does both memory read, and memory writes, the counters for both loads and stores are incremented.

All *fp_speed* benchmarks have a relatively small number of branch instructions, ranging from as low as 0.86% (603.bwaves_s_a) to 14.6% (638.imagick_s). One of the most interesting parameters is the average instructions per cycles (IPC) calculated as the number of instructions divided by the total number of clock cycles. The IPC ranges from as low as 0.83 for 654.rom_s to 3.47 for 638.imagick_s.

Table 5.1 General Parameters for *fp_speed* Benchmarks

fp_speed	Time [s]	Cycles [Billion]	Instructions [Billion]	Branches [Billion]	Loads [Billion]	Stores [Billion]	IPC
603.bwaves_s_a	584.6	2,736.40	4,543.73	38.91	3,261.23	332.07	1.66
603.bwaves_s_b	599.0	2,800.94	4,272.56	37.23	3049.67	316.23	1.53
607.cactuBSSN_s	1,314.4	6,150.85	8,812.93	136.57	4,530.17	970.78	1.43
619.lbm_s	798.8	3,733.40	3,830.41	82.94	1,456.43	647.41	1.03
621.wrf_s	919.5	4,305.56	7,729.07	615.98	3,148.75	591.10	1.80
627.cam4_s	1,401.9	6,564.39	12,079.88	1,228.06	2,835.28	1,438.92	1.84
628.pop2_s	1,013.9	4,730.23	8,121.75	624.82	3,010.64	928.35	1.72
638.imagick_s	4,260.3	1,9937.4	69,141.70	10,083.48	1,3408.3	387.45	3.47
644.nab_s	1,724.7	8,070.99	13,489.82	1,460.08	4,990.24	1,102.54	1.67
649.fotonik3d_s	644.8	3,018.89	3,315.81	101.52	1,851.76	375.92	1.10
654.roms_s	1,510.2	7,068.97	5,867.98	235.25	3,368.39	466.18	0.83

Table 5.2 shows the collected parameters for single-threaded *int_speed* benchmarks. Several benchmarks, e.g., 600.perlbench_s and 602.gcc_s, have multiple reference data inputs marked with alphabetically ordered suffixes. These benchmarks, in general, take less time to execute, ranging from 16.7 s (625.x264_s_a) to 953.8 s (657.xz_s_b). The number of loads ranges from 38.06 billion (625.x264_s_a) to 1.14

trillion (657.xz_s_a) and the number of stores range from as low as 16.81 billion (625.x264_s_a) to 462.04 billion (648.exchange2_s). The benchmarks also vary widely in terms of instruction frequency distribution. Thus, 625.x264_s has only 7.3 to 8 % of branches, whereas 602.gcc_s has 23.2 to 27.9%, depending on the input sets. The IPC across benchmarks varies from as low as 0.67 for 657.xz_s_b to relatively as high as 2.59 for 600.perlbench_s_a.

Table 5.2 General Parameters for *int_speed* Benchmarks

int_speed	Time [s]	Cycles [Billion]	Instructions [Billion]	Branches [Billion]	Loads [Billion]	Stores [Billion]	IPC
600.perlbench_s_a	108.1	505.88	1,311.37	258.98	359.00	247.62	2.59
600.perlbench_s_b	66.0	308.21	716.17	153.58	232.07	123.60	2.32
600.perlbench_s_c	68.1	318.67	713.62	142.61	221.23	131.98	2.24
602.gcc_s_a	194.2	907.80	1,468.41	409.75	483.08	71.77	1.62
602.gcc_s_b	80.5	375.51	548.41	127.55	153.01	73.49	1.46
602.gcc_s_c	76.5	357.57	532.24	125.25	149.49	69.65	1.49
605.mcf_s	320.5	1,499.32	1,193.65	288.87	422.47	91.84	0.80
620.omnetpp_s	299.0	1,399.21	1,101.10	242.71	372.28	190.77	0.79
623.xalancbmk_s	211.6	989.48	964.65	230.12	273.35	57.18	0.97
625.x264_s_a	16.7	77.86	181.92	13.31	38.06	16.81	2.34
625.x264_s_b	49.6	231.76	570.47	45.59	113.93	44.20	2.46
625.x264_s_c	53.2	247.95	604.38	48.19	124.94	48.73	2.44
631.deepsjeng_s	218.5	1,022.12	1,777.00	230.38	428.05	186.21	1.74
641.leela_s	332.8	1,557.87	1,927.53	298.07	510.41	181.13	1.24
648.exchange2_s	195.3	913.54	2,062.57	233.34	750.13	462.04	2.26
657.xz_s_a	721.4	3,377.22	4,720.70	732.82	1,147.87	351.72	1.40
657.xz_s_b	953.8	4,464.23	3,002.84	445.58	723.61	260.95	0.67

5.1.2 Control-Flow Instructions and Branch Prediction Accuracy

Branches are often one of the key parameters determining overall performance. In this set of experiments, the following branch parameters are collected: (a) the number of retired branches per 1K (Kilo) instructions, (b) the number of branch misses per 1K instructions, and (c) the percentage of branch misses. Depending on the type, frequency, and distribution of branches, branch misses tend to be costly in terms of wasted clock cycles. As the processors under study use speculative execution, each misprediction requires clearing out the instructions in the pipeline for the mispredicted block and re-steering the front-end to a different block, wasting numerous CPU clock cycles. Minimizing branch misprediction directly contributes to improved performance.

Table 5.3 and Table 5.4 shows the branch-related parameters for *fp_speed* and *int_speed*, respectively. As described above, the frequency of branches is relatively low in *fp_speed* benchmarks, ranging from 8.5 (603.bwaves_s_a) to 145.8 (638.imagick_s) per 1K instructions. The branch predictors are very accurate, and the percentage of branch misses is below 1% for all benchmarks except two (644.nab_s and 619.lbm_s).

On the other hand, the frequency of branches in *int_speed* is significantly higher, ranging from 73 (625.x264_s_a) to 279 (602.gcc_s_a) per 1K instructions. The frequency of branch misses varies from as low as 0.33% (600.perlbench_s_c) to 10.8% (641.leela_s). The *int_speed* benchmarks suffer from a high misprediction rate when compared to *fp_speed*. The benchmarks with relatively high branch miss rate (> 2%) are good candidates for research efforts targeting improved branch prediction accuracy.

Table 5.3 Branch Characteristics for *fp_speed*

fp_speed	branches per 1K instructions	branch misses per 1K instructions	% branch misses
603.bwaves_s_a	8.56	0.05	0.63
603.bwaves_s_b	8.71	0.05	0.56
607.cactuBSSN_s	15.50	0.01	0.05
619.lbm_s	21.65	0.59	2.70
621.wrf_s	79.70	0.81	1.02
627.cam4_s	101.66	0.74	0.73
628.pop2_s	76.93	0.46	0.60
638.imagick_s	145.84	0.33	0.22
644.nab_s	108.24	3.17	2.93
649.fotonik3d_s	30.62	0.07	0.23
654.roms_s	40.09	0.19	0.49

Table 5.4 Branch Characteristics for *int_speed*

int_speed	branches per 1K instructions	branch misses per 1K instructions	% branch misses
600.perlbench_s_a	197.49	1.90	0.96
600.perlbench_s_b	214.45	1.36	0.63
600.perlbench_s_c	199.84	0.67	0.33
602.gcc_s_a	279.04	2.33	0.83
602.gcc_s_b	232.58	5.72	2.46
602.gcc_s_c	235.34	5.44	2.31
605.mcf_s	242.01	21.35	8.82
620.omnetpp_s	220.42	4.54	2.06
623.xalancbmk_s	238.55	0.93	0.39
625.x264_s_a	73.17	1.00	1.36
625.x264_s_b	79.91	1.25	1.56
625.x264_s_c	79.74	1.48	1.86
631.deepsjeng_s	129.65	6.01	4.63
641.leela_s	154.64	16.82	10.88
648.exchange2_s	113.13	3.37	2.98
657.xz_s_a	155.23	11.14	7.17
657.xz_s_b	148.39	11.05	7.45

5.1.3 Cache Hierarchy

Cache behavior is a crucial factor in determining system performance. To evaluate its impact, the following events are collected: (a) the number of L2 references per 1K instructions; (b) the number of L2 misses per 1K instructions; (c) the percentage of L2 misses; (d) the number of last-level cache (LLC) references per 1K instructions; (e) the number of LLC misses per 1K instructions, and (f) the percentage of LLC misses. This set of parameters should give us a good insight on how many references are observed at L2 and LLC caches and what percentage of these references end up being misses.

Table 5.5 and Table 5.6 show cache-related parameters for *fp_speed* and *int_speed*, respectively. For *fp_speed*, the number of L2 references (which is equivalent to the number of L1 misses) ranges from as low as 23.4 (644.nab_s) to as high as 467 (619.lbm_s) per 1K instructions. The number of L2 misses per 1K instructions ranges from 4.89 (644.nab_s) to 118.24 (619.lbm_s). The number of LLC references per 1K instructions ranges from 9.07 (644.nab_s) to 193.87 (654.roms_s). It should be noted that the number of LLC references exceed the number of L2 misses as it includes requests initiated by hardware and software data prefetchers. The number of LLC misses ranges from as low as 0.16 (638.imagick_s) per 1K instructions to 85.39 (654.roms_s). For *int_speed* benchmarks the number of L2 references range from 0.22 (648.exchange2_s) to 330.48 (605.mcf_s) per 1K instructions. The number of L2 misses per 1K instructions ranges from 0.01 (648.exchange2_s) to 108.55 (605.mcf_s). The number of LLC references per 1K instructions range from 0.2 (648.exchange2_s) to 156.32 (605.mcf_s). The number of LLC misses ranger from 0.0000348 (648.exchange2_s) to 33.75 (620.omnetpp_s) for 1K instructions.

Table 5.5 L2 and LLC Instruction Breakdown for *fp_speed*

fp_speed	L2 ref. per 1K Inst.	L2 misses per 1K Inst.	% L2 misses	LLC ref. per 1K Inst.	LLC misses per 1K Inst.	% LLC misses
603.bwaves_s_a	182.95	57.81	31.60	84.69	52.64	62.18
603.bwaves_s_b	194.59	58.02	29.82	82.82	50.80	61.85
607.cactuBSSN_s	109.47	18.47	16.87	26.78	11.94	44.58
619.lbm_s	467.04	118.24	25.32	158.36	64.94	41.01
621.wrf_s	130.47	35.21	26.98	50.53	11.92	23.60
627.cam4_s	66.07	14.55	22.15	22.62	7.20	31.86
628.pop2_s	236.78	51.25	21.65	79.96	13.67	17.11
638.imagick_s	31.65	9.88	31.03	18.37	0.16	0.85
644.nab_s	23.45	4.89	20.86	9.07	1.37	15.11
649.fotonik3d_s	348.55	89.70	25.75	118.45	68.67	57.93
654.roms_s	532.03	140.30	26.38	193.87	85.39	44.04

Table 5.6 L2 and LLC Instruction Breakdown for *int_speed*

int_speed	L2 ref. per 1K Inst.	L2 misses per 1K Inst.	% L2 misses	LLC ref. per 1K Inst.	LLC misses per 1K Inst.	% LLC misses
600.perlbench_s_a	16.72	4.27	25.53	6.95	0.35	5.04
600.perlbench_s_b	38.55	8.47	21.98	13.34	0.31	2.32
600.perlbench_s_c	23.36	7.24	30.99	12.15	4.39	36.37
602.gcc_s_a	119.41	33.05	27.68	50.24	17.41	34.81
602.gcc_s_b	80.09	22.77	28.43	37.17	2.50	6.71
602.gcc_s_c	72.91	21.32	29.24	34.92	3.03	8.67
605.mcf_s	330.48	108.55	32.85	156.32	28.55	18.27
620.omnetpp_s	170.54	52.77	30.93	79.66	33.75	42.37
623.xalancbmk_s	243.56	88.07	36.16	131.54	5.14	3.90
625.x264_s_a	21.99	4.13	18.80	7.10	2.19	30.66
625.x264_s_b	24.38	5.54	22.69	9.69	0.93	9.66
625.x264_s_c	21.52	4.87	22.64	8.43	0.86	10.13
631.deepsjeng_s	18.84	3.65	19.32	5.32	3.51	65.98
641.leela_s	10.10	1.50	14.87	2.79	0.04	1.52
648.exchange2_s	0.22	0.01	5.04	0.02	0.00	0.22
657.xz_s_a	40.20	11.41	28.42	18.09	2.02	11.19
657.xz_s_b	107.89	36.56	33.88	55.77	21.46	38.47

5.1.4 Top-down Microarchitectural Analysis Method Results

This section presents the Top-down Microarchitectural Analysis Method results for the SPEC CPU2017 Speed benchmarks. Figure 5.1 shows the results of TMAM for all the speed benchmarks executed with one thread. As described in Section 4.1, all available pipeline slots are classified into four groups: *Retiring*, *Bad Speculation*, *Front-End Bound*, and *Back-End Bound*. The figure also illustrates the IPC on the secondary axis. It should be noted that IPC expresses the number of ISA instructions retired per clock cycles, whereas TMAM analysis looks at micro-operations. Typically, an Intel64 instruction is translated into one or more micro-operations, but multiple micro-operations can be fused together (even from different instructions: macro-fusion).

Looking at *fp_speed*, the average IPC is 1.64, ranging from 0.83 (654.roms_s) to 3.47 (638.imagick_s). The *Retiring* slots averages to 45%, the *Front-End Bound,* and *Bad Speculation* have stalls averaging up to 3% and 2% respectively. However, the *Back-End Bound* accounts for 50% of the stalls emerging as a major bottleneck for the suite. There is a strong correlation between the *Retiring* slots and the IPC metric – the higher the percentage of *Retiring* slots, the higher IPC. Thus, 638.imagick_s has 92% of slots in *Retiring* which translates into IPC of 3.47. On the other side, 654.rom_s has only 23% of slots in *Retiring* which translates into IPC of 0.83. Majority of benchmarks in *fp_speed* is limited by the *Back-End Bound* (75% for 654.rom_s), whereas the impact of *Front-End Bound* and *Bad Speculation* stalls is relatively small.

With respect to *int_speed*, the average aggregate IPC is 1.70, ranging from 0.67 (657.xz_b_s) to 2.59 (600.perlbench_s_a). With 39% of *Retiring* slots, *int_speed* has a higher percentile of *Bad Speculation* and *Front-End Bound* stalls at 15% and 16%,

respectively. The *Back-End Bound* stalls account for 30%. Looking at individual benchmarks, the distribution of IPC varies less drastically when compared to *fp_speed*. 600.perlbench_s_a has the highest IPC of 2.59 with 60% *Retiring* slots, 12% *Bad Speculation*, 19% *Front-End Bound* and 8% *Back-End Bound* stalls. 657.xz_s_b has the smallest IPC of 0.67 as it has just 14% of *Retiring* slots, 17% of slots wasted for *Bad Speculation*, 4% for *Front-End Bound* and a 65% delay caused by *Back-End Bound* stalls.

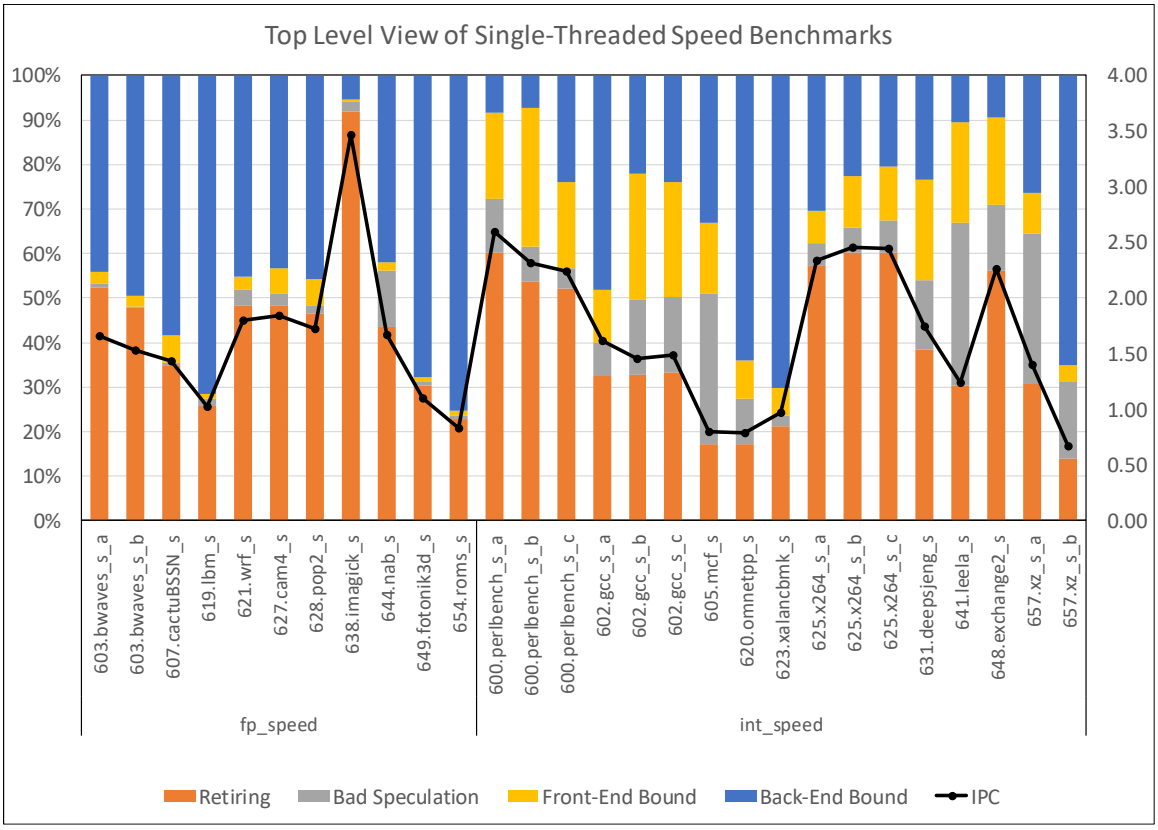

Figure 5.1 Top-Level View of Single-Threaded Speed Benchmarks

A view of the speed benchmarks executed with six threads is seen in Figure 5.2. Though the general behavior remains identical for single-threaded *fp_speed,* a huge increase in *Back-End Bound* stalls is observed. On the other hand, *int_speed* does not show any change because all the benchmarks but one, are single threaded.

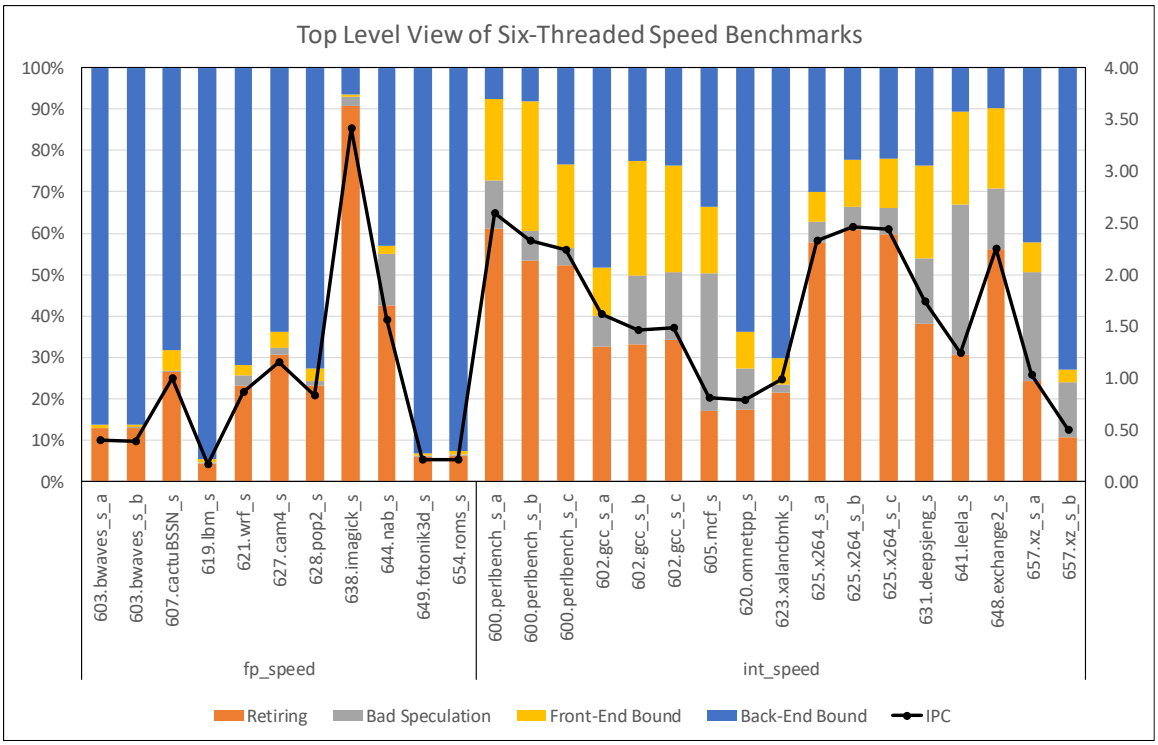

Figure 5.2 Top-Level View of Six-Threaded Speed Benchmarks

With most of the speed benchmarks having the *Back-End Bound* stalls as the bottleneck for both single-threaded execution and six thread execution, a deeper look at *Back-End Bound* is required. Figure 5.3 shows the *Back-End Bound* stalls breakdown for all the speed benchmarks executed single-threaded. The *Back-End Bound* stalls are further divided into *Core Bound* and *Memory Bound* stalls. Averaging across *fp_speed* benchmarks, the *Memory Bound* stalls account for 31% and *Core Bound* for 19% of the total slots. For *int_speed* benchmarks, the *Memory Bound* stalls account for 19% and *Core Bound* stalls account for 11%.

Figure 5.4 shows the *Back-End Bound* stall breakdown for speed benchmarks executed with six threads. *Memory Bound* stalls average increases to 60% for *fp_speed* and *Core Bound* stall average reduces to just 10% during six thread execution.

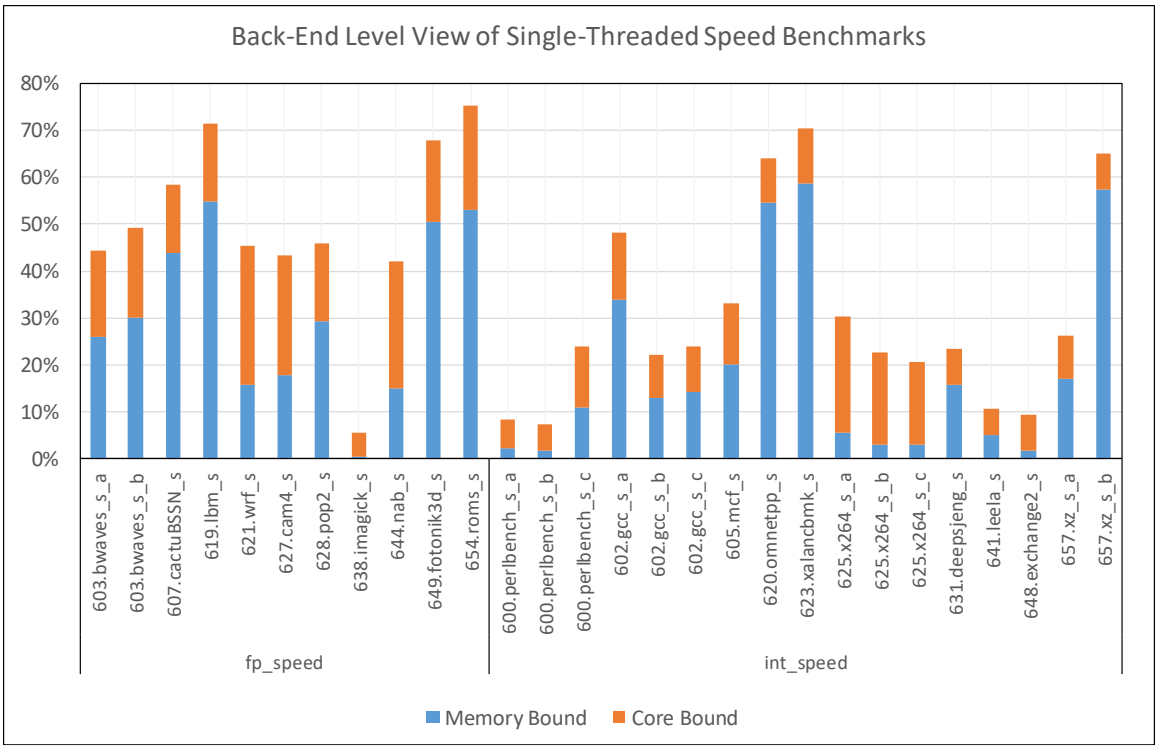

Figure 5.3 Back-End Level View of Single-Threaded Speed Benchmarks

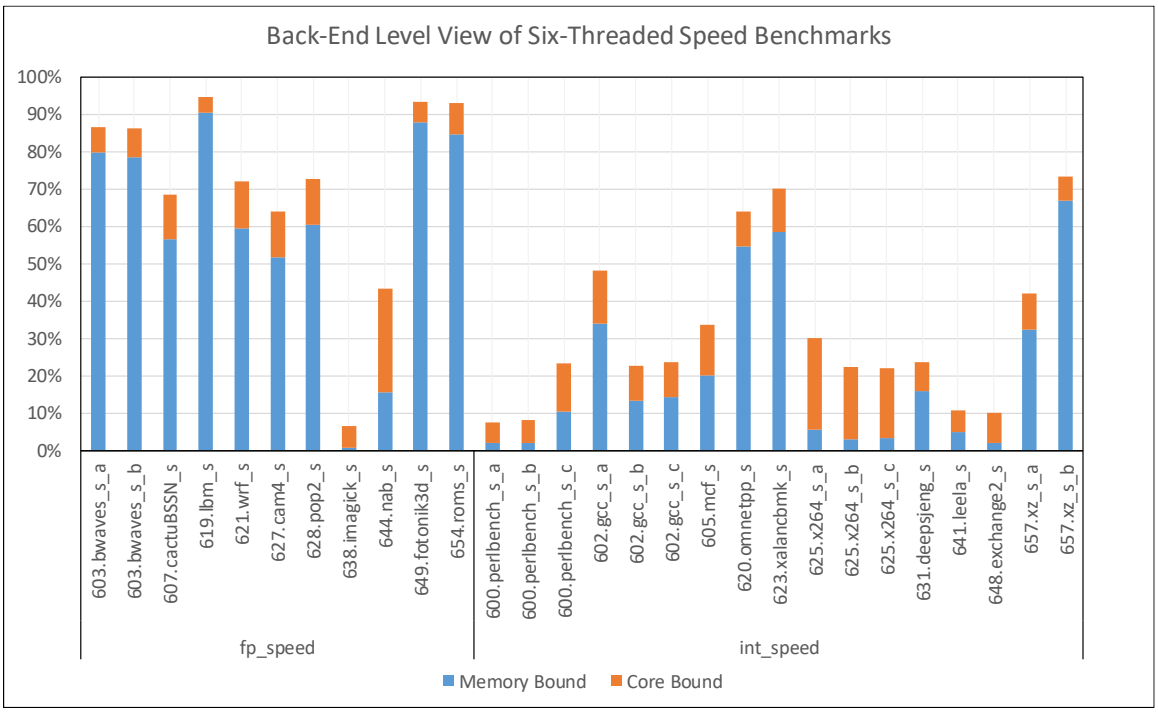

Figure 5.4 Back-End Level View of Six-Threaded Speed Benchmarks

For most of the speed benchmarks, memory bound stalls were the biggest bottleneck. Whereas the Top-level view describes pipeline slot utilization, it does not directly translate into clock cycles spent on memory reference operations. However, a Memory-Level view from *Intel VTune Amplifier* shown in Figure 5.5 quantifies the percentages of the total clock cycles wasted (stalled) in different levels of the memory hierarchy, including L1 cache, L2 cache, L3 cache, Memory, or store buffers for single threaded execution. Most of the benchmarks spend a high percentage of time in main memory (DRAM). Increasing memory speed and bandwidth could help mitigate this issue.

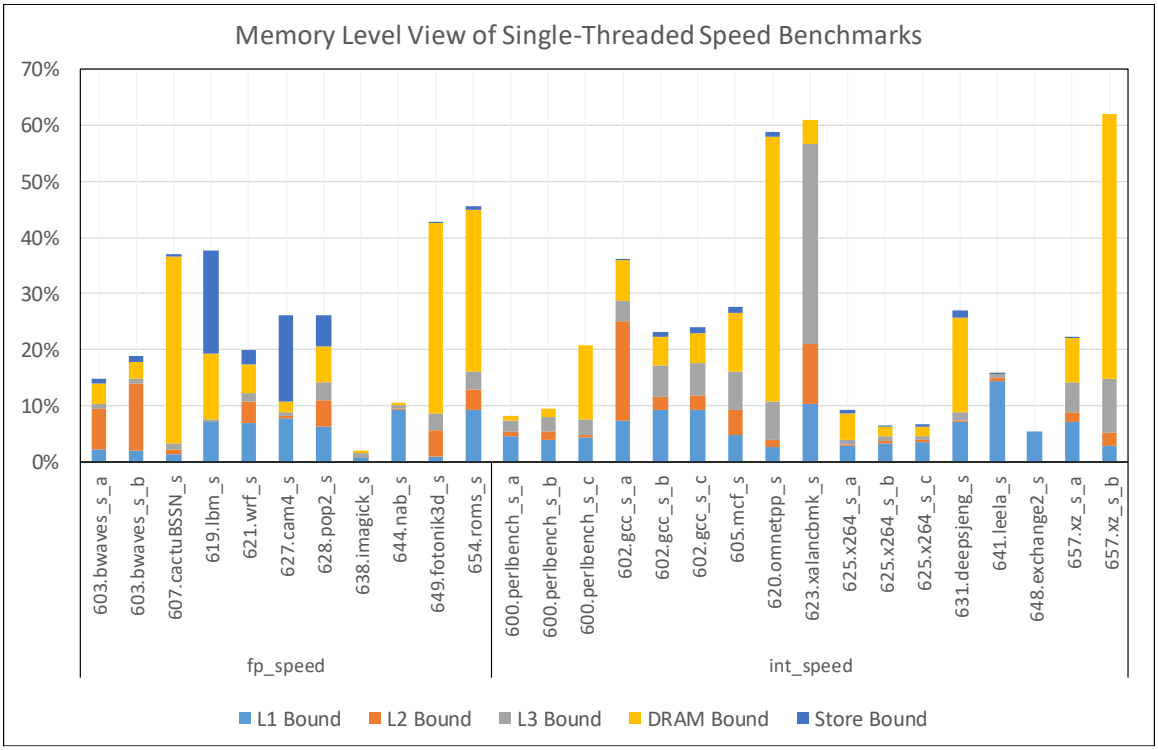

Figure 5.5 Memory Level View of Single-Threaded Speed Benchmarks

Figure 5.6 gives the Memory-Level view for the speed benchmarks for six thread execution. Similar to single threaded execution, most of the benchmarks spend

a high percentage of time in main memory (DRAM) which proves to be a significant bottleneck.

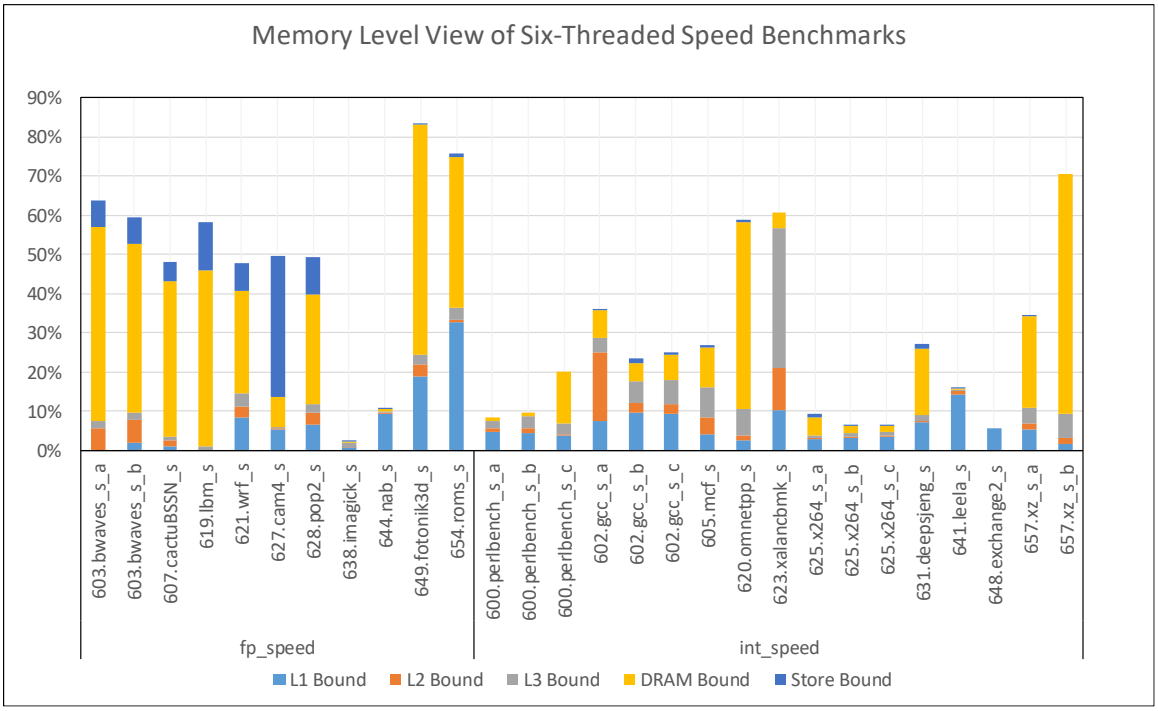

Figure 5.6 Memory Level View of Six-Threaded Speed Benchmarks

5.1.5 Clock Rates, Energy, and Power

The clock frequency and the number of threads vary for different executions. In the interest of defining the impact thread and frequency scaling on performance and energy consumption, a study was conducted to evaluate either impacts [25]. For single threaded execution, the processor makes use of all the turbo bins to clock out at 4.67 GHz for all the benchmarks, whereas for six thread execution, equal distribution of the turbo bins results in a clock frequency of 4.30 GHz. Therefore, a single-threaded application will execute at higher CPU clock frequency, and the energy consumption will also be higher [26]. Table 5.7 and Table 5.8 show the runtime (column

time), the energy consumed, the average power for each benchmark executed with one and six threads and the *PerfEE speedup* (column Speedup). A metric to determine the overview of performance and energy efficiency is defined here, *PerfEE* (Performance and Energy Efficiency) represents the metric. The *PerfEE* metric for an individual SPEC speed benchmark, *SBi*, running with *N* threads, $PerfEE.Speed\ (SBi, N)$, is defined as follows:

$$PerfEE.Speed\ (SBi, N) = \frac{1}{ExeTime\ (SBi, N) \times Energy\ consumed(SBi, N)} \qquad Eq.5.1$$

where *ExeTime (SBi, N)* is the execution time for the N-threaded benchmark.

To evaluate the performance and energy efficiency across *N*-thread combinations with respect to 1-thread execution, a speedup ratio is used as shown in Eq.5.2

$$PerfEE.Speedup(SBi, N) = \frac{ExeTime(SBi, 1) \times Energy\ consumed(SBi, 1)}{ExeTime(SBi, N) \times Energy\ consumed(SBi, N)} \qquad Eq.5.2$$

A *PerfEE Spee*dup number more than one means that running N threads produces better performance and energy savings. In the case of *fp_speed* benchmarks, most of the benchmarks have a good *PerfEE speedup* and would suggest 6-thread execution over 1-thread execution. On the other hand, there are benchmarks like 619.lbm_s and 649.fotonik3d_s that have a loss of performance and energy consumption for 6-thread runs. As these benchmarks do not have speedup for parallel execution, and yet utilize the cores and consume energy they degrade in performance.

In the case of the *int_speed* benchmarks, as all but one (657.xz_s) are single threaded, irrespective of the number of threads allocated, the *PerfEE speedup* remains 1 for all but one.

Table 5.7 Energy and Power Analysis for *fp_speed*

	1 Thread			6 Threads			PerfEE
fp_speed	Time [s]	Energy [J]	Power [W]	Time [s]	Energy [J]	Power [W]	Speed up
603.bwaves_s_a	584.58	14099.45	24.58	451.47	17260.90	39.55	1.06
603.bwaves_s_b	599.04	14089.08	23.79	429.04	16495.40	40.03	1.19
607.cactuBSSN_s	1314.43	30321.58	21.28	347.04	17160.10	48.12	6.69
619.lbm_s	798.77	18869.83	23.07	962.27	37913.80	39.25	0.41
621.wrf_s	919.51	23273.49	25.32	372.74	18857.40	51.27	3.04
627.cam4_s	1401.87	32991.86	23.04	420.00	21556.70	50.88	5.11
628.pop2_s	1013.92	28493.09	26.19	399.78	21605.50	54.74	3.34
638.imagick_s	4260.29	103927.35	24.06	804.03	51310.40	64.17	10.73
644.nab_s	1724.69	35973.77	21.01	337.91	18162.60	54.42	10.11
49.fotonik3d_s	644.81	16581.59	25.39	644.64	26302.40	41.25	0.63
654.roms_s	1510.21	39697.60	25.47	1127.15	53309.00	46.28	1.00

Table 5.8 Energy and Power Analysis for *int_speed*

	1 Thread			6 Threads			PerfEE
int_speed	Time [s]	Energy [J]	Power [W]	Time [s]	Energy [J]	Power [W]	Speed up
600.perlbench_s_a	108.13	2679.95	24.68	108.49	2615.17	24.00	1.02
600.perlbench_s_b	65.96	1676.44	25.10	65.63	1643.97	24.80	1.03
600.perlbench_s_c	68.08	1561.42	22.75	68.24	1552.49	22.69	1.00
602.gcc_s_a	194.19	4119.30	21.22	194.48	4102.64	21.09	1.00
602.gcc_s_b	80.46	1870.62	23.18	80.28	1848.62	23.09	1.01
602.gcc_s_c	76.50	1775.93	23.07	76.41	1749.47	22.75	1.02
605.mcf_s	320.51	7397.63	22.26	313.61	7174.20	21.92	1.05
620.omnetpp_s	298.95	7145.10	21.36	298.98	7030.45	21.05	1.02
623.xalancbmk_s	211.57	4406.91	20.70	210.91	4336.91	20.73	1.02
625.x264_s_a	16.74	381.96	23.02	16.73	366.34	15.54	1.04
625.x264_s_b	49.56	1179.00	23.74	49.66	1158.20	23.41	1.02
625.x264_s_c	53.20	1249.48	23.56	53.51	1227.96	22.01	1.01
631.deepsjeng_s	218.48	5476.70	22.82	218.52	5426.85	22.54	1.01
641.leela_s	332.79	7519.14	22.58	332.63	7513.54	22.48	1.00
648.exchange2_s	195.30	4676.96	23.92	196.20	4567.88	23.41	1.02
657.xz_s_a	721.40	16861.07	22.14	255.65	10259.60	41.53	4.64
657.xz_s_b	953.83	19767.11	18.33	307.72	11682.50	33.74	5.24

5.2 SPEC CPU2017 Rate Benchmarks Characterization

Similar to the discussions for Speed benchmarks, the results for the SPEC CPU2017 Rate benchmarks characterization are divided into five subsections. Each subsection encompasses *fp_rate* and *int_rate* results in that order. Though the runs carried out were for 1,6 and 12 copies, results for single-copy execution are shown in this section for the fact that for multiple copies, most parameters are just summed up arithmetically. Execution times for multiple copies are discussed in CHAPTER 6.

5.2.1 General View of Benchmarks

The primary set of experiments collects information about individual SPEC CPU2017 Rate benchmarks run under the Linux *perf* utility. Parameters collected are as follows: the total execution time, total number of clock cycles, total number of retired instructions, total number of retired branch instructions, total number of retired load instructions, and total number of retired store instructions.

Table 5.9 shows the collected parameters for single-copy *fp_rate* benchmarks. A primary observation is that the execution time of the rate benchmarks are relatively small with respect to the Speed counter parts, ranging from 33.3 s (503.bwaves_r_a) to 285.1 s (549.fotonik3d_r). The number of retired instructions starts from as low as 234 billion (503.bwaves_r_a) to as high as 2.6 trillion (511.povray_r). Branches range from 1.07% (503.bwaves_r_c) to 16.38% (526.blender_r) of the total retired instructions. Loads/memory reads range from 152.54 billion (503.bwaves_r_a) to 999.35 billion (510parest_r). Store/memory writes vary from 20.78 billion (503.bwaves_r_a) to 337.52 billion (510parest_r). The rate benchmarks are light weight benchmarks designed to be less intensive so that multiple copies are run to test throughput. The

average instruction per cycle (IPC) for *fp_rate* ranges from 1.04 for 554.roms_r to 2.68 for 508.namd_r.

Table 5.9 General Parameters for *fp_rate* Benchmarks

fp_rate	Time [s]	Cycles [Billion]	Instructions [Billion]	Branches [Billion]	Loads [Billion]	Stores [Billion]	IPC
503.bwaves_r_a	33.3	156.23	234.00	2.52	152.54	20.78	1.50
503.bwaves_r_b	53.1	247.46	369.82	3.99	240.24	32.94	1.49
503.bwaves_r_c	41.5	194.57	288.44	3.11	187.49	25.70	1.48
503.bwaves_r_d	50.3	235.58	349.62	3.73	228.45	30.86	1.48
507.cactuBSSN_r	141.6	662.91	1,065.26	18.17	546.44	119.26	1.61
508.namd_r	156.1	730.72	1,959.10	40.53	732.51	218.57	2.68
510.parest_r	240.1	1,122.39	2,368.70	246.23	999.35	86.54	2.11
511.povray_r	239.6	1,121.53	2,609.36	416.26	998.25	337.52	2.33
519.lbm_r	75.9	355.16	567.27	11.59	231.50	74.61	1.60
521.wrf_r	160.7	750.51	1,343.14	104.92	552.50	101.79	1.79
526.blender_r	195.1	913.73	1,688.78	276.70	568.06	83.38	1.85
527.cam4_r	153.4	718.66	1,499.99	172.51	428.55	148.93	2.09
538.imagick_r	208.3	977.49	2,508.37	320.30	493.65	199.48	2.57
544.nab_r	181.5	849.94	1,382.99	149.97	482.13	131.82	1.63
549.fotonik3d_r	285.1	1,335.32	1,401.94	36.10	814.60	162.77	1.05
554.roms_r	149.9	701.38	730.01	29.04	431.52	60.16	1.04

Table 5.10 shows the collected parameters for single-copy *int_rate* benchmarks. Collectively *int_rate* is the smallest benchmark suite in SPEC CPU2017 with execution times ranging from 15.3 s (525.x264_r_a) to 310.3 s (520.omnetpp_r). The number of retired instructions ranges from 204.91 billion (502.gcc_r_d) to 2062.58 billion (548.exchange2_r). The frequency of branch instructions varies from 7.48% (525.x264_r_a) to 24.95% (502.gcc_r_c). The number of loads/memory-reads varies from 37.13 billion (525.x264_r_a) to 750.13 billion (548.exchange2_r). Stores/memory-

writes range from 16.21 billion (525.x264_r_a) to 462.04 billion (548.exchange2_r). The instruction per cycle (IPC) ranges from as low as 0.76 for 520.omnetpp_r to 2.59 for 500.perlbench_r_a.

Table 5.10 General Parameters for *int_rate* Benchmarks

int_rate	Time [s]	Cycles [Billion]	Instructions [Billion]	Branches [Billion]	Loads [Billion]	Stores [Billion]	IPC
500.perlbench_r_a	108.0	504.52	1,308.81	259.05	358.94	247.49	2.59
500.perlbench_r_b	66.2	309.34	718.88	154.12	232.53	123.93	2.32
500.perlbench_r_c	67.8	317.03	713.76	142.75	220.71	131.39	2.25
502.gcc_r_a	29.6	138.08	207.27	49.07	58.04	27.16	1.50
502.gcc_r_b	35.0	163.62	243.22	58.09	68.65	32.51	1.49
502.gcc_r_c	34.4	160.93	259.48	64.75	76.96	26.00	1.61
502.gcc_r_d	33.5	156.84	204.91	49.62	59.84	26.12	1.31
502.gcc_r_e	47.8	223.53	257.11	58.66	72.37	40.56	1.15
505.mcf_r	174.6	817.75	677.42	153.51	227.96	80.16	0.83
520.omnetpp_r	310.3	1,452.72	1,101.09	242.71	372.28	190.77	0.76
523.xalancbmk_r	208.0	972.63	964.21	229.93	273.28	57.19	0.99
525.x264_r_a	15.3	71.71	169.37	12.67	37.13	16.21	2.37
525.x264_r_b	46.0	214.84	536.91	44.15	114.08	45.26	2.50
525.x264_r_c	49.3	230.45	569.25	46.58	124.56	49.20	2.47
531.deepsjeng_r	186.2	870.96	1,525.87	197.95	366.10	158.92	1.75
541.leela_r	333.0	1,559.32	1,927.88	298.13	510.42	181.14	1.23
548.exchange2_r	195.7	915.55	2,062.58	233.34	750.13	462.04	2.25
557.xz_r_a	87.6	408.10	361.93	52.58	83.28	31.64	0.89
557.xz_r_b	96.0	449.73	923.11	191.04	238.07	30.01	2.05
557.xz_r_c	74.4	347.24	513.61	82.97	126.48	31.00	1.48

5.2.2 Control-Flow Instructions and Branch Prediction Accuracy

Table 5.11 and Table 5.12 show the branch related parameters collected during runtime for *fp_rate* and *int_rate* benchmarks, respectively. As discussed in the previous section, the frequency of branches is relatively low, ranging from 10.66 (503.bwaves_r_d) to 163.85 (526.blender_r) per 1K instructions. The misprediction rate ranges from as low as 0.02 (507.cactuBSSN_r and 503.bwaves_r_c) to 5.57 (526.blender_r) per 1K instructions.

Table 5.11 Branch Characteristics for *fp_rate*

fp_rate	branches per 1K instructions	branch misses per 1K instructions	% branch misses
503.bwaves_r_a	10.75	0.04	0.33
503.bwaves_r_b	10.79	0.03	0.32
503.bwaves_r_c	10.76	0.02	0.18
503.bwaves_r_d	10.66	0.06	0.53
507.cactuBSSN_r	17.06	0.02	0.10
508.namd_r	20.69	0.97	4.70
510.parest_r	103.95	4.68	4.50
511.povray_r	159.53	1.09	0.68
519.lbm_r	20.43	0.03	0.14
521.wrf_r	78.11	0.89	1.14
526.blender_r	163.85	5.57	3.40
527.cam4_r	115.01	1.14	0.99
538.imagick_r	127.69	0.96	0.75
544.nab_r	108.44	4.40	4.06
549.fotonik3d_r	25.75	0.07	0.26
554.roms_r	39.78	0.10	0.25

On the other hand, the frequency of branches in *int_rate* is significantly higher than that of its Speed counterpart. The number of retired branch instructions ranges

from 74.83 (525.x264_r_a) to 249.53 (502.gcc_r_c) per 1K instructions. The branch misses are relatively high, ranging from 0.69 (500.perlbench_r_c) to 22.74 (505.mcf_r) per 1K instructions. Benchmarks with relatively high branch miss rates (>4% in this suite) are good candidates for research for improving branch prediction units.

Table 5.12 Branch Characteristics for *int_rate*

int_rate	branches per 1K instructions	branch misses per 1K instructions	% branch misses
500.perlbench_r_a	197.93	1.90	0.96
500.perlbench_r_b	214.39	1.35	0.63
500.perlbench_r_c	200.00	0.67	0.34
502.gcc_r_a	236.74	5.44	2.30
502.gcc_r_b	238.82	4.90	2.05
502.gcc_r_c	249.53	3.87	1.55
502.gcc_r_d	242.14	1.78	0.73
502.gcc_r_e	228.15	1.96	0.86
505.mcf_r	226.61	22.74	10.03
520.omnetpp_r	220.42	4.52	2.05
523.xalancbmk_r	238.47	0.95	0.40
525.x264_r_a	74.83	1.07	1.43
525.x264_r_b	82.23	1.32	1.61
525.x264_r_c	81.82	1.57	1.92
531.deepsjeng_r	129.73	6.25	4.81
541.leela_r	154.64	16.82	10.88
548.exchange2_r	113.13	3.37	2.98
557.xz_r_a	145.29	10.29	7.08
557.xz_r_b	206.96	3.52	1.70
557.xz_r_c	161.54	8.61	5.33

5.2.3 Cache Hierarchy

Table 5.13 and Table 5.14 show cache-related parameters for *fp_rate* and *int_rate* respectively. For *fp_rate*, the number of L2 references per 1K instructions (which is equivalent to the number of L1 misses) ranges from as low as 28.36 (526.blender_r) to as high as 512.91 (554.roms_r) per 1K instructions. The number of L2 misses per 1K instructions ranges from 1.65 (508.namd_r) to 153.58 (554.roms_r). The number of LLC references per 1K instructions ranges from 2.40 (508.namd_r) to 206.12 (554.roms_r). Hardware and software prefetchers account for the increased number of LLC reference with respect to L2 misses. The number of LLC misses ranges from as low as 0.03 (538.imagick_r) to 62.50 (549.fotonik3d_r) per 1K instructions.

Table 5.13 L2 and LLC Instruction Breakdown for *fp_rate*

fp_rate	L2 ref. per 1K Inst.	L2 misses per 1K Inst.	% L2 misses	LLC ref. per 1K Inst.	LLC misses per 1K Inst.	% LLC misses
503.bwaves_r_a	236.23	79.00	33.44	108.30	61.87	57.20
503.bwaves_r_b	235.22	78.57	33.40	107.78	61.38	56.85
503.bwaves_r_c	235.27	77.90	33.09	107.30	61.64	57.45
503.bwaves_r_d	238.14	79.65	33.44	108.41	61.57	56.81
507.cactuBSSN_r	132.64	18.19	13.73	26.90	8.51	31.64
508.namd_r	31.62	1.65	5.22	2.40	0.63	26.43
510.parest_r	157.42	48.76	30.97	74.52	1.34	1.79
511.povray_r	58.26	4.40	7.60	8.92	0.00	0.00
519.lbm_r	221.37	45.37	20.47	68.48	52.73	77.12
521.wrf_r	135.16	36.09	26.64	51.90	11.58	22.30
526.blender_r	28.36	9.51	33.56	14.38	1.43	9.81
527.cam4_r	111.31	25.74	23.07	38.39	3.28	8.53
538.imagick_r	17.94	2.11	11.74	3.40	0.03	0.73
544.nab_r	30.11	5.23	17.37	9.38	1.21	12.87
549.fotonik3d_r	330.99	118.37	35.76	160.18	62.50	39.32
554.roms_r	512.91	153.58	29.94	206.12	49.58	24.05

For *int_rate* benchmarks, the number of L2 references range from 0.23 (548.exchange2_r) to 241.22 (505.mcf_r) per 1K instructions. The number of L2 misses per 1K instructions ranges from 0.02 (548.exchange2_r) to 88.73 (523.xalancbmk_r). The number of LLC references per 1K instructions range from 0.01 (548.exchange2_r) to 130.85 (523.xalancbmk_r). The number of LLC misses ranges from 0.0000670 (548.exchange2_r) to 34.19 (520.omnetpp_r) per 1K instructions.

Table 5.14 L2 and LLC Instruction Breakdown for *int_rate*

int_rate	L2 ref. per 1K Inst.	L2 misses per 1K Inst.	% L2 misses	LLC ref. per 1K Inst.	LLC misses per 1K Inst.	% LLC misses
500.perlbench_r_a	17.06	4.32	25.25	7.00	0.34	5.01
500.perlbench_r_b	38.44	8.46	22.05	13.82	0.31	2.21
500.perlbench_r_c	23.77	7.28	30.75	11.95	4.50	37.78
502.gcc_r_a	73.99	21.51	29.06	35.21	3.80	10.79
502.gcc_r_b	77.03	23.26	30.18	37.00	4.93	13.30
502.gcc_r_c	82.38	25.14	30.50	40.45	5.58	13.83
502.gcc_r_d	101.03	29.96	29.66	46.28	18.66	40.31
502.gcc_r_e	195.85	60.05	30.66	81.70	15.29	18.58
505.mcf_r	241.22	70.63	29.28	105.48	24.32	23.05
520.omnetpp_r	171.83	53.07	30.88	79.74	34.19	43.01
523.xalancbmk_r	246.69	88.73	35.97	130.85	5.19	3.96
525.x264_r_a	23.99	4.42	18.43	7.65	2.36	30.77
525.x264_r_b	27.58	6.56	23.77	10.20	1.00	9.80
525.x264_r_c	23.78	5.47	22.99	9.39	0.92	9.76
531.deepsjeng_r	15.50	2.80	18.09	4.36	2.33	53.49
541.leela_r	10.72	1.64	15.31	2.60	0.04	1.60
548.exchange2_r	0.23	0.02	7.83	0.01	0.00	0.78
557.xz_r_a	77.71	25.95	33.36	38.48	11.10	28.73
557.xz_r_b	37.72	8.07	21.40	12.11	0.76	6.30
557.xz_r_c	37.84	11.15	29.48	16.80	3.27	19.58

5.2.4 Top-down Microarchitectural Analysis Method Results

This section presents the Top-down Microarchitecture Analysis Method results for the SPEC CPU2017 Rate benchmarks. Figure 5.7 shows the results of TMAM for all the rate benchmarks executed with one copy. The figure also illustrates the IPC on the secondary axis.

Looking at *fp_rate*, the average IPC is 1.77, ranging from 1.04 (554.roms_r) to 2.68 (508.namd_r). The *Retiring* slots averages to 49%, the *Front-End Bound,* and *Bad Speculation* have minimal stalls averaging up to 5% and 6% respectively. However, the *Back-End Bound* accounts for 40% of the stalls. The correlation between the *Retiring* slots and the IPC metric seen earlier for speed benchmarks hold true for rate benchmarks too. Thus, 508.namd_r has 61% of slots in *Retiring* which translates into IPC of 2.68. On the other side, 549.fotonik3d_r and 554.roms_r have only 28% of slots in *Retiring* which translates into IPC of 1.05 and 1.04 respectively. Expectedly, the majority of benchmarks in the *fp_rate* suite is *Back-End Bond* 70% (549.fortonik3d_r and 554.roms_r), whereas the impact of *Front-End Bound* and *Bad Speculation* stalls is relatively small.

With respect to *int_rate*, the average IPC is 1.69, ranging from 0.76 (520.omnetpp_r) to 2.59 (500.perlbench_s_a). With 39% of *Retiring* slots, *int_rate* has a higher percentage of *Bad Speculation* and *Front-End Bound* stalls at 14% and 17%, respectively. The *Back-End Bound* stalls account for 30%. Looking at individual benchmarks, 500.perlbench_s_a has the highest IPC of 2.59 with 61% *Retiring* slots, 12% *Bad Speculation*, 19% *Front-End Bound* and 8% *Back-End Bound* stalls. 520.omnetpp_r has the smallest IPC of 0.76 as it has just 17% of *Retiring* slots, 10% of slots

wasted for *Bad Speculation*, 9% wasted on the *Front-End Bound* and a 64% delay caused by *Back-End Bound*.

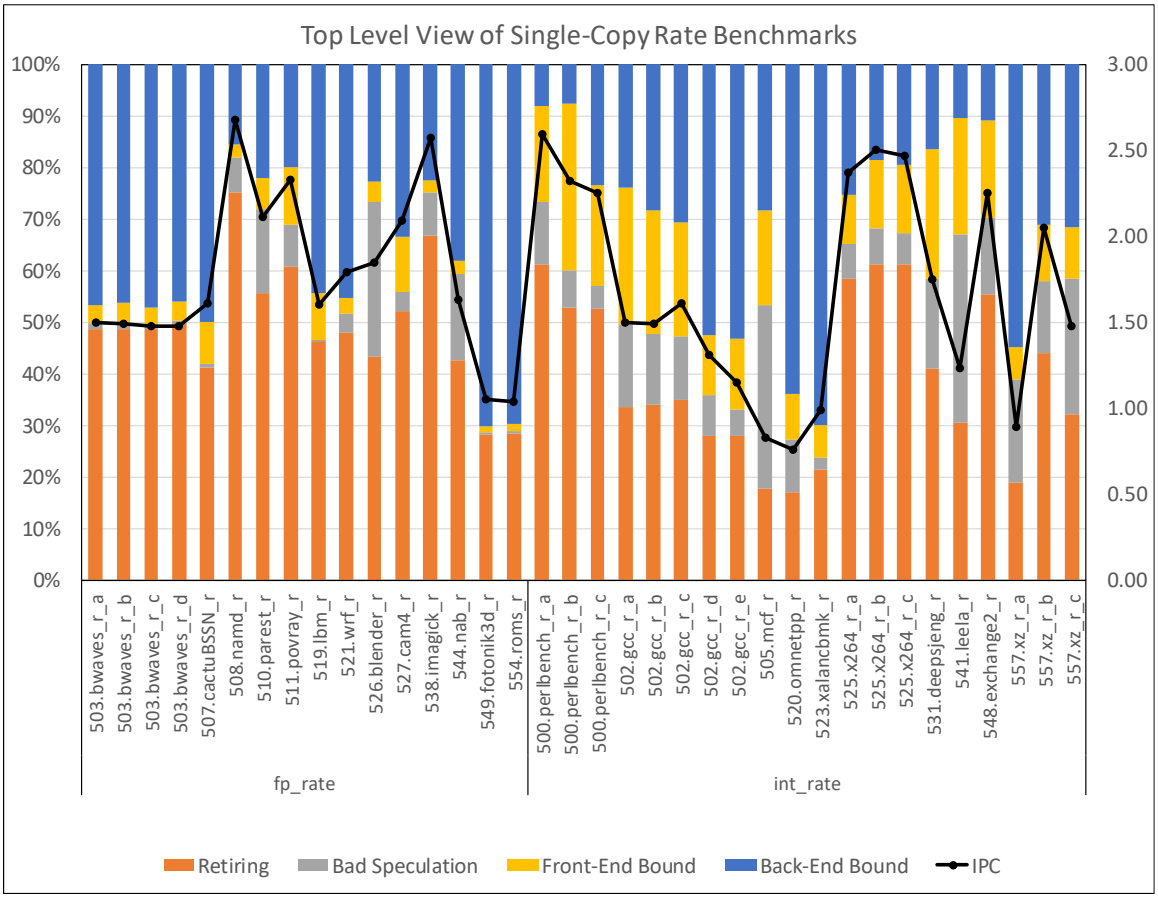

Figure 5.7 Top-Level View of Single-Copy Rate Benchmarks

A view of the Rate benchmarks executed with six copies is seen in Figure 5.8. The general view remains identical for *fp_rate* and *int_rate* for 1-copy execution. However, a huge increase in *Back-End Bound* stalls is observed, especially for the *fp_rate* benchmarks. The *Memory-Bound* and *Core-Bound* delays are caused by contention and multiple copies using the same shared resources on and off the core.

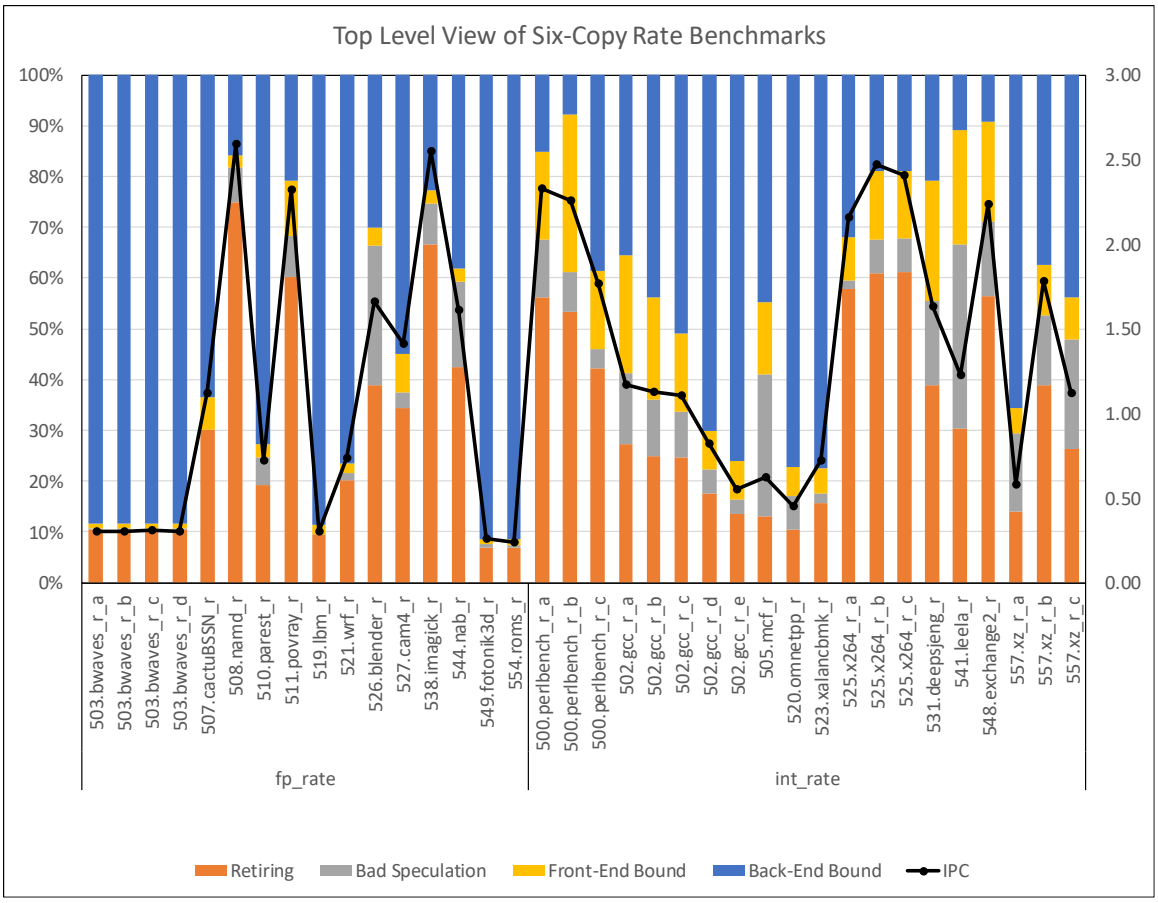

Figure 5.8 Top-Level View of Six-Copy Rate Benchmarks

With most of the Rate benchmarks having *Back-End Bound* as the bottleneck, a deeper look is required. Figure 5.9 shows the *Back-End Bond* breakdown for all the rate benchmarks for a single copy execution. The *Back-End Bound* stalls are further divided into *Core Bound* and *Memory Bound* stalls. Averaging across *fp_rate*, the *Memory Bound* stalls account for 22% and *Core Bound* for 18% of the total slots. For *int_rate* the *Memory Bound* stalls account for 19% and *Core Bound* stalls account for 11%.

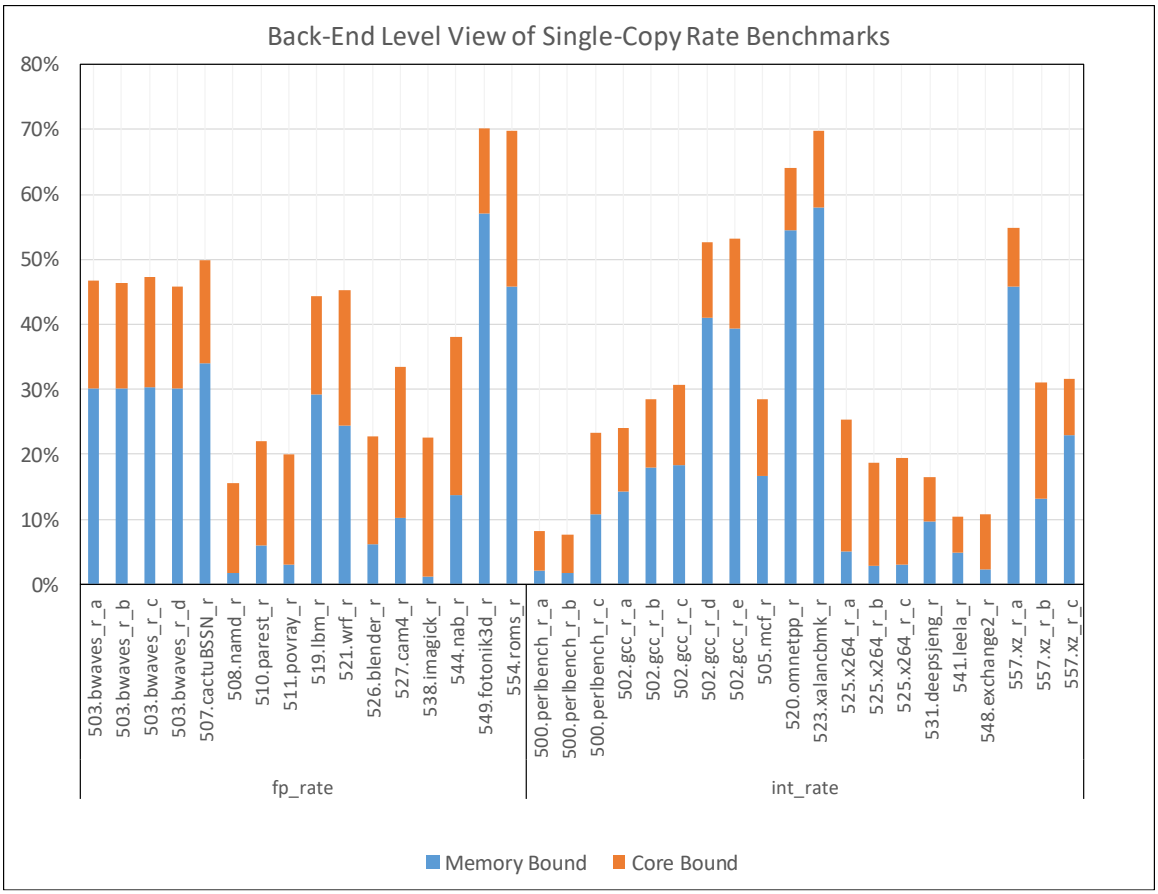

Figure 5.9 Back-End Level View of Single-Copy Rate Benchmarks

Figure 5.10 shows the *Back-End Bond* breakdown for all the rate benchmarks for 6-copy execution. The *Memory Bound* delay is significantly higher for six-copy runs. The *fp_rate* benchmarks seem to be affected more by the *Memory Bound* delay than *int_rate* benchmarks which suggest that they are more memory intensive. The *Core Bound*. Averaging across *fp_rate*, the *Memory Bound* stalls account for 53% and *Core Bound* for 11% of the total slots. For *int_rate* the *Memory Bound* stalls account for 30% and *Core-Bound* stalls account for 10%.

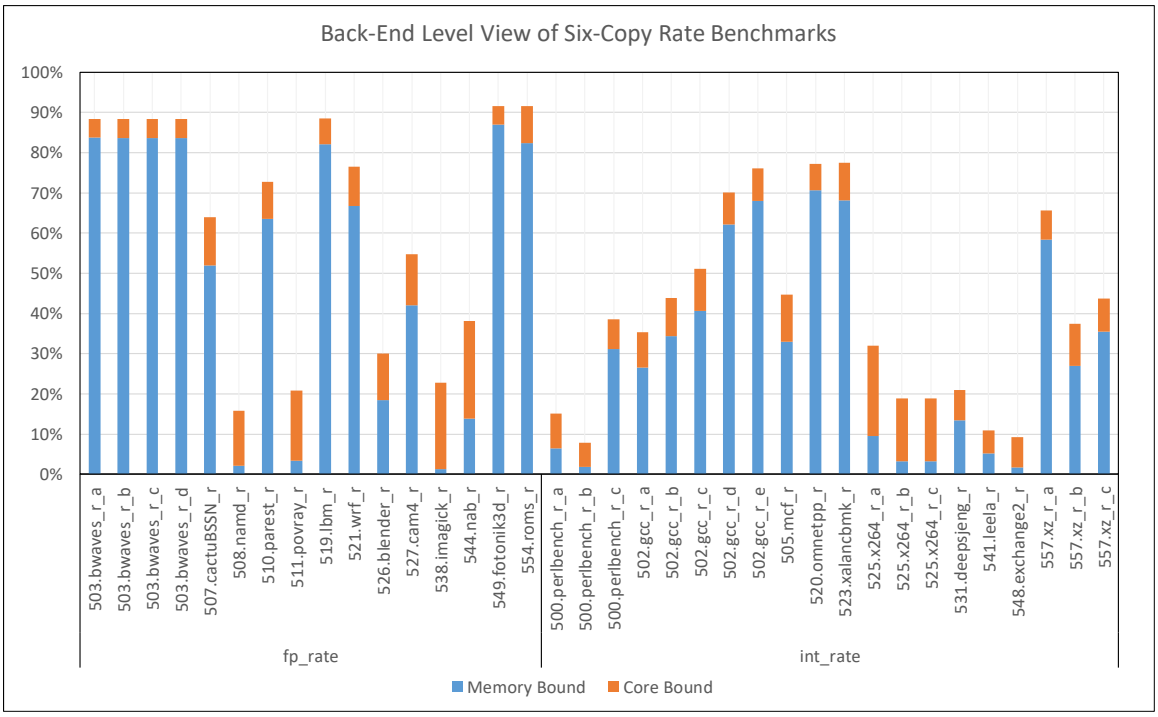

Figure 5.10 Back-End Level View of Six-Copy Rate Benchmarks

For most of the rate benchmarks, memory bound stalls were the biggest bottleneck. Memory-Level from *Intel VTune Amplifier* quantifies the percentages of the total clock cycles wasted (stalled) in different levels of memory hierarchy, including L1 cache, L2 cache, L3 cache, Memory, or store buffers, as shown in Figure 5.11. Most of the benchmarks spend a high percentage of time in main memory (DRAM). Increasing memory speed and bandwidth could help mitigate this issue.

Figure 5.12 represents the percentages of the total clock cycles wasted (stalled) in memory hierarchy for 6-copy execution. A much higher percentage of stalls is observed when compared to 1-copy execution. Memory is seen to be the biggest bottleneck for throughput application.

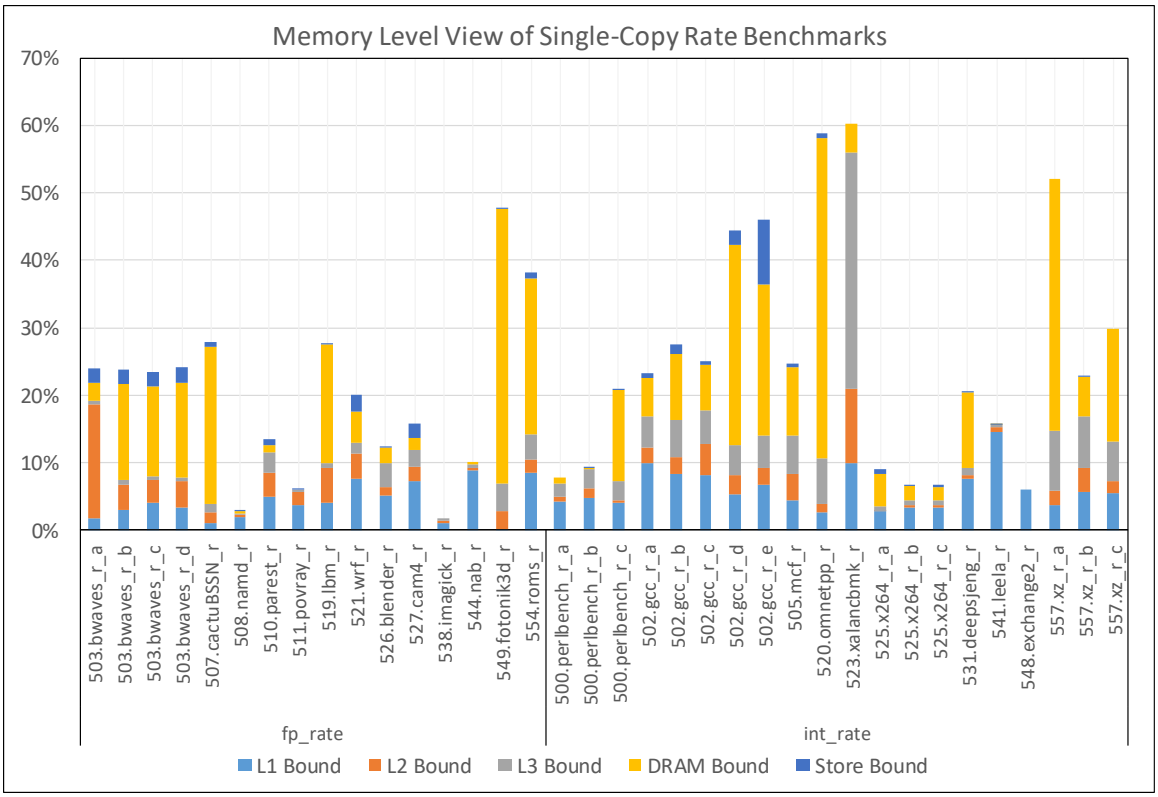

Figure 5.11 Memory Level View of Single-Copy Rate Benchmarks

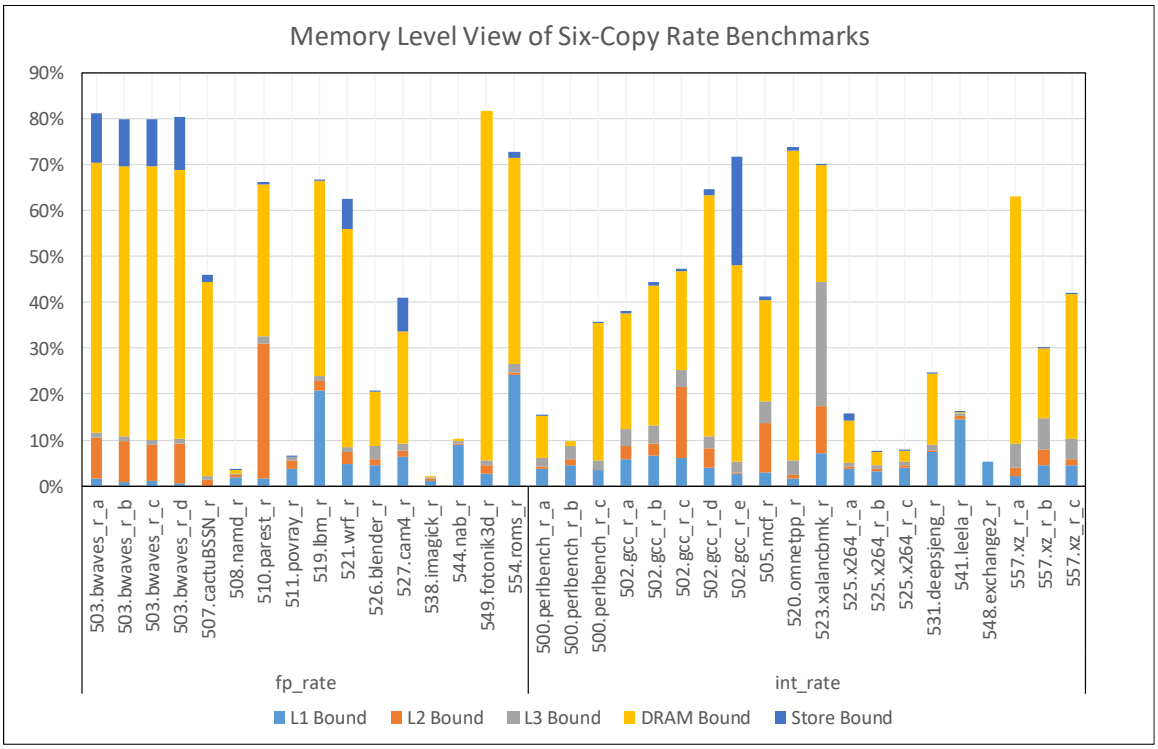

Figure 5.12 Memory Level View of Six-Copy Rate Benchmarks

5.2.5 Clock Rates, Energy, and Power

Table 5.15 and Table 5.16 show the runtime (column time), the energy consumed, the average power for each benchmark executed with one and six copies and the *PerfEE speedup* (column Speedup). A metric to determine the overview of performance and energy efficiency is defined here, *PerfEE* (Performance and Energy Efficiency) represents the metric. The *PerfEE* metric for an individual SPEC rate benchmark, *RBi*, running with *N* copies, $PerfEE.Rate(RBi, N)$, is defined as follows:

$$PerfEE.Rate\ (RBi, N) = \frac{1}{ExeTime\ (RBi, N) \times Energy\ consumed(RBi, N)} \quad Eq.5.1$$

where *ExeTime (RB$_i$, N)* is the execution time for the N-copy benchmark.

To evaluate the performance and energy efficiency across *N*-copy execution with respect to 1-copy execution a speedup ratio is used as shown in Eq.5.2.

$$PerfEE.Speedup(RBi, N) = \frac{ExeTime(RBi, 1) * N * Energy\ consumed(SBi, 1) * N}{ExeTime(SBi, N) \times Energy\ consumed(SBi, N)} \quad Eq.5.2$$

For the comparison to be fair, the *ExeTime* and *Energy consumed* is equated to the number of copies run. The logic is to have a ratio that compares the execution parameters of single copy execution and an *N* copy execution. To do so, the single copy execution parameters are converted to match the N copy execution parameters. A *PerfEE.Spee*dup number more than one would mean that running N copies provides better performance and energy savings.

A few of the *fp_rate* benchmarks have *PerfEE Speedup* of less than 1 (503.bwaves_r, 519.lbm_r, and 554.roms_r). Resource and memory contention caseses degradation in performance. It is better to run single copy execution in series for these benchmarks to have better performance and to save energy. For all other benchmarks, a significant speedup as high as12.06 (538.imagick_r) is observed.

In the case of *int_rate,* as the benchmarks are not memory intensive, all the benchmarks show good *PerfEE speedup* ranging from 3.44 (502.gcc_r_e) to 12.08 (541.leela_r).

Table 5.15 Machine Parameters for *fp_rate*

	1 Copy			6 Copies			
fp_rate	Time [s]	Energy [J]	Power [W]	Time [s]	Energy [J]	Power [W]	PerfEE Speedup
503.bwaves_r_a	33.33	897.53	27.16	179.75	6965.39	40.35	0.86
503.bwaves_r_b	53.07	1416.07	27.22	284.83	11236.90	40.61	0.85
503.bwaves_r_c	41.55	1121.67	27.18	220.21	8705.32	40.95	0.88
503.bwaves_r_d	50.30	1345.20	27.22	267.11	10648.70	40.53	0.86
507.cactuBSSN_r	141.61	3483.87	22.94	223.09	12015.30	53.40	6.63
508.namd_r	156.07	3827.73	24.11	175.27	11074.90	65.81	11.08
510.parest_r	240.09	6394.82	25.63	765.72	36661.40	47.48	1.97
511.povray_r	239.59	6587.88	26.55	263.03	19386.60	71.25	11.14
519.lbm_r	75.88	2423.70	29.88	442.31	21221.60	48.88	0.71
521.wrf_r	160.65	4213.29	25.64	421.41	21201.30	50.11	2.73
526.blender_r	195.10	4836.83	23.57	237.39	14059.30	58.01	10.18
527.cam4_r	153.39	4057.21	25.54	245.65	14650.60	59.65	6.23
538.imagick_r	208.26	5075.38	24.02	227.52	14212.70	62.46	11.77
544.nab_r	181.53	3965.02	21.64	199.21	10783.70	55.18	12.06
549.fotonik3d_r	285.05	7287.68	24.59	1259.17	50875.20	40.61	1.17
554.roms_r	149.85	4220.30	26.45	704.06	32754.90	46.80	0.99

Table 5.16 Machine Parameters for *int_rate*

	1 Copy			6 Copies			
int_rate	Time [s]	Energy [J]	Power [W]	Time [s]	Energy [J]	Power [W]	PerfEE Speedup
500.perlbench_r_a	108.01	2764.75	24.45	128.90	8094.22	61.94	10.30
500.perlbench_r_b	66.16	1737.51	25.12	72.93	4943.74	66.49	11.48
500.perlbench_r_c	67.77	1696.25	22.98	93.97	5211.05	55.73	8.45
502.gcc_r_a	29.59	709.15	23.14	41.34	2274.05	54.30	8.04
502.gcc_r_b	35.03	841.06	22.90	50.72	2722.34	53.68	7.68
502.gcc_r_c	34.45	818.10	23.00	54.45	2773.36	51.16	6.72
502.gcc_r_d	33.50	747.46	21.61	58.74	2700.81	46.23	5.68
502.gcc_r_e	47.82	1092.04	22.21	107.77	5073.20	46.11	3.44
505.mcf_r	174.61	4013.49	21.99	253.74	13198.90	50.36	7.53
520.omnetpp_r	310.32	7479.93	21.64	589.94	27500.10	46.09	5.15
523.xalancbmk_r	207.96	4630.70	21.16	314.09	14565.50	46.34	7.58
525.x264_r_a	15.31	373.28	23.95	19.61	976.26	32.17	10.75
525.x264_r_b	45.96	1164.81	24.63	50.44	3256.41	64.67	11.73
525.x264_r_c	49.27	1236.82	24.55	55.30	3449.63	58.79	11.50
531.deepsjeng_r	186.15	4606.82	23.43	218.11	13081.40	59.61	10.82
541.leela_r	332.98	7772.05	22.73	363.97	21184.30	58.19	12.08
548.exchange2_r	195.71	4784.83	23.90	213.46	13384.60	62.19	11.80
557.xz_r_a	87.61	1990.88	19.46	144.38	6477.58	43.67	6.71
557.xz_r_b	96.04	2340.74	21.63	120.79	6690.70	53.18	10.01
557.xz_r_b	74.39	1785.66	21.32	106.47	5427.22	50.57	8.28

CHAPTER 6

ARCHITECTURAL EVALUATION

This chapter gives the compilation of the obtained results for the SPEC CPU2017 benchmark suites for multiple threads/copies across all the test systems discussed in section 3.3. Section 6.1 gives execution time across test machines and evaluates speedup when multiple threads are run for the speed benchmarks and multiples copies for rate benchmarks are run. Section 6.2 evaluates the advantages and disadvantages of hardware prefetching and for different benchmarks in the SPEC CPU2017 benchmark suites.

6.1 SPEC CPU2017 Execution Evaluation on Test System

This section shows the results of performance evaluation of the speed and rate benchmarks on all the test machines while varying the number of threads/copies. The tables in this section give two parameters, the execution time (or runtime) and the speedup. The execution time is obtained directly from the SPEC CPU2017 *runcpu* utility when benchmarks are run with the reference data inputs. The primary goal of this analysis is to quantify scalability of individual benchmarks as a function of the number of threads for the speed benchmarks, and the number of copies for the rate benchmarks. The speedup metric for an individual SPEC speed benchmark, *SBi*, running with N threads, $Speedup.Speed\ (SBi, N)$, is calculated as shown in Eq.6.1, where $ExeTime\ (SB_i, 1)$ is the execution time for the single-threaded benchmark and $ExeTime\ (SB_i, N)$ is the execution time for the N-threaded benchmark. The speedup metric for

an individual SPEC rate benchmark, *RBi*, when run with N copies, $SpeedUp.Rate\ (RBi, N)$, is shown in Eq.6.2, where *ExeTime (RBi,1)* is the execution time for one copy of the benchmark, and *ExeTime (RBi,N)* is the execution time for *N* copies.

$$Speedup.Speed\ (SBi\ , N) = \frac{ExeTime\ (SBi, 1)}{ExeTime\ (SBi, N)} \qquad Eq.6.1$$

$$Speedup.Rate\ (RBi\ , N) = \frac{N * ExeTime\ (RBi, 1)}{ExeTime\ (RBi, N)} \qquad Eq.6.2$$

The secondary aspect of this analysis is to compare the performance of processors from the same class, as well the performance of all processors considered (Core i7 vs. Xeon). It should be noted that all the test systems have processors that operate at different clock frequencies. Moreover, the processor clock frequency at any point in time depends on the number of active cores. Table 6.1 shows the measured runtime CPU clock frequencies for all the test systems. In evaluating different test platforms, rather than analyzing the performance of individual benchmarks, the SPEC composite metric *SPEC2017_ratio_base* is used for individual benchmark suites. In comparing the performance of two different machines, say A and B, the speedup is determined as shown in Eq.6.3.

$$SpeedupAtoB\ (N) = \frac{SPEC2017\_ratio\_base\ (A, N)}{SPEC2017\_ratio\_base\ (B, N)} \qquad Eq.6.3$$

Secondly, to eliminate the impact of clock frequencies on performance, a scaled speedup metric is derived as shown in Eq. 4, where FreqA and FreqB are active clock

frequencies of test machines A and B, respectively. As all the test machines use the same operating systems and run benchmarks compiled using the same compiler, the scaled speedup captures differences in performance due to processor microarchitecture and memory latency.

$$SpeedupScaledAtoB\ (N) = \frac{SPEC2017\_ratio\_base(A,N) \cdot FreqB}{SPEC2017\_ratio\_base(B,N) \cdot FreqA} \qquad Eq.\ 6.4$$

Table 6.1 Runtime CPU Clock Frequency for all Test Systems.

Test systems		Frequency [GHz]						
	Threads/Copies	1	4	6	8	12	16	24
mtsano Core i7-4770	*fp_speed*	3.69	3.30	3.27	3.20			
	int_speed	3.88	3.84	3.85	3.84			
	fp_rate	3.64	3.35	3.24	3.17			
	int_rate	3.88	3.44	3.39	3.27			
clingmansdome Core i7-8700k	*fp_speed*	4.68	4.39	4.30	4.31	4.29		
	int_speed	4.67	4.64	4.63	4.63	4.63		
	fp_rate	4.68	4.39	4.30	4.31	4.30		
	int_rate	4.67	4.39	4.30	4.31	4.30		
vidrak Xeon E3-1240 V2	*fp_speed*	3.78	3.78	3.59	3.59			
	int_speed	3.78	3.78	3.76	3.76			
	fp_rate	3.70	3.70	3.59	3.58			
	int_rate	3.70	3.70	3.59	3.59			
mtleconte Xeon E5-2643 V3	*fp_speed*	3.64	3.56	3.50	3.50	3.52	3.53	3.54
	int_speed	3.66	3.65	3.65	3.64	3.64	3.65	3.65
	fp_rate	3.61	3.59	3.51	3.51	3.51	3.51	3.51
	int_rate	3.66	3.64	3.57	3.57	3.57	3.57	3.58

6.1.1 SPEC CPU2017 *fp_speed*

Table 6.2 and Figure 6.1 show the execution times and speedups for *fp_speed* on *mtsano* (Core i7-4770) when the number of threads is 1, 4, 6, and 8. The performance of 4-thread benchmarks is found to be the best, with the individual speedups

ranging from 1.11 (619.lbm_s) to 3.13 (607.cactuBSSN_s). Table 6.3 and Figure 6.2 show the execution times and speedups for *fp_speed* on *clingmansdome* (Core i7-8700k) when the number of threads is 1, 4, 6, 8, and 12. The performance of six-thread benchmarks is found to be the best, with the individual speedups ranging from 0.83 (619.lbm_s) to 5.32 (638.imagick_s, 644.nab_s).

Table 6.4 and Figure 6.3 show execution times and speedups for *fp_speed* on *vidrak* (Xeon E3-1240 V2) when the number of threads is 1, 4, 6 and 8. The performance of 4-thread benchmarks is found to be the best, with the individual speedups ranging from 1.16 (619.lbm_s) to 3.69 (638.imagick_s and 644.nab_s). Table 6.5 and Figure 6.4 shows the execution times and speedups of *fp_speed* on *mtleconte* (Xeon E5-2643 V3) when the number of threads is 1, 4, 6, 8, 12, 16, and 24. The performance of the twelve-thread benchmarks is found to be the best, with the individual speedups ranging from 1.53 (619.lbm_s) to 10.73 (644.nab_s).

A broad view of the *fp_speed* benchmarks is seen here in terms of scalability. The benchmark 619.lbm_s show little to none parallelism in *mtsano* (Core i7-4770) and the Xeons, however, in *clingmansdome* (Core i7-8700k) any increase in the number of threads from 1, in effect degrades performance irrespective of the number of threads.

Most of the benchmarks perform the best with good speedup when the number of threads is equal to the number of physical cores. However, for some of those benchmarks, any further increase in the number of threads essentially degrades performance because logical cores do not provide the same speedup as physical cores (603.bwaves_s, 621.wrf_s, 628.pop2_s, 649.fotonik3d_s, and 654.roms_s). A few bench-

marks have excellent scalability showing performance improvements when the number of threads matches the number of logical cores (607.cactuBSSN_s, 627.cam4_s, and 644.nab_s).

An interesting observation to note is that benchmarks like 607.cactuBSSN_s have speedup for cases when the number of software threads created equal the number of physical core and logical cores, however, it shows show significant degradation in performance when the number of software threads is between the number of physical cores and the number of logical cores. As these applications are OS scheduled, an unusually high number of context switches and CPU migrations are observed for this condition causing the performance to degrade.

An important parameter that impacts scalability is the percentage of parallelizable code in the benchmark. Even in the case of a perfectly parallelizable application that is memory intensive, increase in the number of threads might end up hurting performance. This could explain the performance degradation seen for benchmarks when the number of created software threads exceed the number of physical cores. The fact that architecture is not a big factor for scalability can clearly be observed by these runs as the behavior for an individual benchmark remains consistent across architectures and computing platforms.

Finally, the *fp_speed* benchmark suite shows excellent variability with a wide range of execution times and speedups. This sort of behavior is extremely useful for stressing various aspects of a machine using a single suite.

Table 6.2 Runtime and Speedup of *fp_speed* in Core i7-4770

Mtsano	Runtime (sec) & Speedup			
Core i7-4770	*fp_speed*			
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads
603.bwaves_s	1,521.77	1,119.37	1,169.62	1,173.50
	1.00	1.36	1.30	1.30
607.cactuBSSN_s	2,068.59	661.66	710.92	622.76
	1.00	3.13	2.91	3.32
619.lbm_s	1,286.82	1,163.55	1,192.10	1,209.18
	1.00	1.11	1.08	1.06
621.wrf_s	1,421.59	644.88	668.29	667.32
	1.00	2.20	2.13	2.13
627.cam4_s	2,296.64	798.14	724.98	665.13
	1.00	2.88	3.17	3.45
628.pop2_s	1,795.71	638.52	762.42	718.40
	1.00	2.81	2.36	2.50
638.imagick_s	5,743.99	1,728.26	1,749.33	1,688.03
	1.00	3.32	3.28	3.40
644.nab_s	2,538.04	712.33	671.01	649.47
	1.00	3.56	3.78	3.91
649.fotonik3d_s	995.73	816.07	833.83	846.45
	1.00	1.22	1.19	1.18
654.roms_s	2,435.36	1,566.64	1,591.33	1,768.64
	1.00	1.55	1.53	1.38

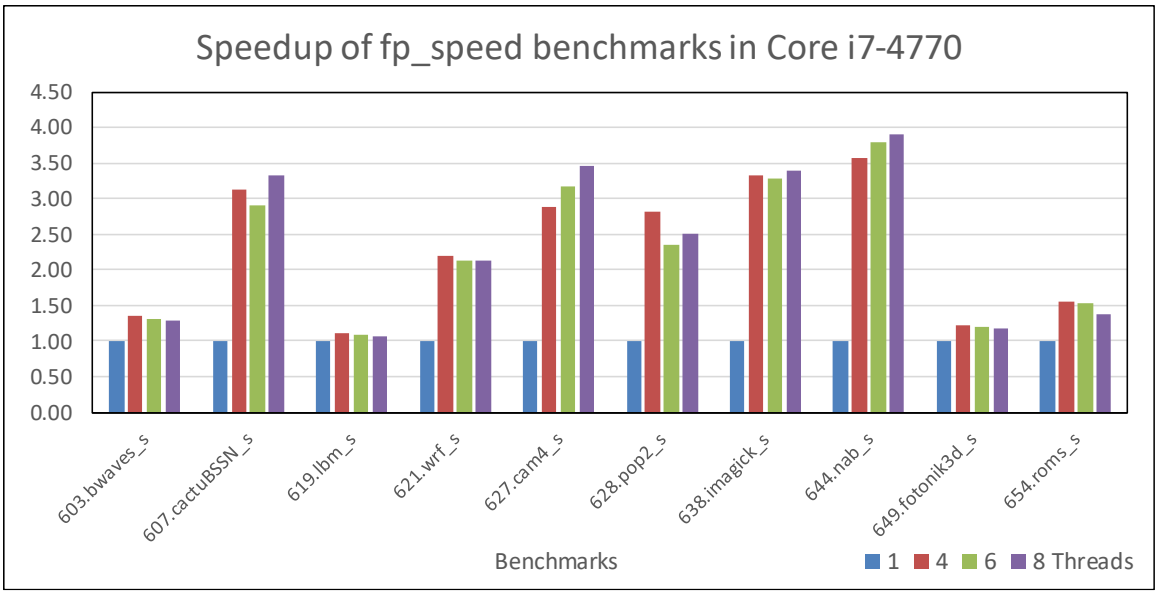

Figure 6.1 Speedup of *fp_speed* Benchmarks in Core i7-4770

Table 6.3 Runtime and Speedup of *fp_speed* in Core i7-8700K

clingmansdome	Runtime (sec) & Speedup				
Core i7-8700K	*fp_speed*				
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads	12 Threads
603.bwaves_s	1,184.63	855.14	883.91	900.51	947.98
	1.00	1.39	1.34	1.32	1.25
607.cactuBSSN_s	1,328.01	441.28	345.33	391.84	316.48
	1.00	3.01	3.85	3.39	4.20
619.lbm_s	800.08	927.47	962.06	977.49	1,028.64
	1.00	0.86	0.83	0.82	0.78
621.wrf_s	913.50	416.46	373.41	390.26	350.58
	1.00	2.19	2.45	2.34	2.61
627.cam4_s	1,405.44	512.62	420.80	427.42	421.70
	1.00	2.74	3.34	3.29	3.33
628.pop2_s	1,014.45	410.15	390.80	439.23	458.29
	1.00	2.47	2.60	2.31	2.21
638.imagick_s	4,263.84	1,155.69	800.85	830.20	774.63
	1.00	3.69	5.32	5.14	5.50
644.nab_s	1,724.49	467.38	324.33	312.90	258.56
	1.00	3.69	5.32	5.51	6.67
649.fotonik3d_s	644.87	621.22	647.39	648.68	678.18
	1.00	1.04	1.00	0.99	0.95
654.roms_s	1,518.88	1,090.72	1,118.48	1,125.80	1,312.23
	1.00	1.39	1.36	1.35	1.16

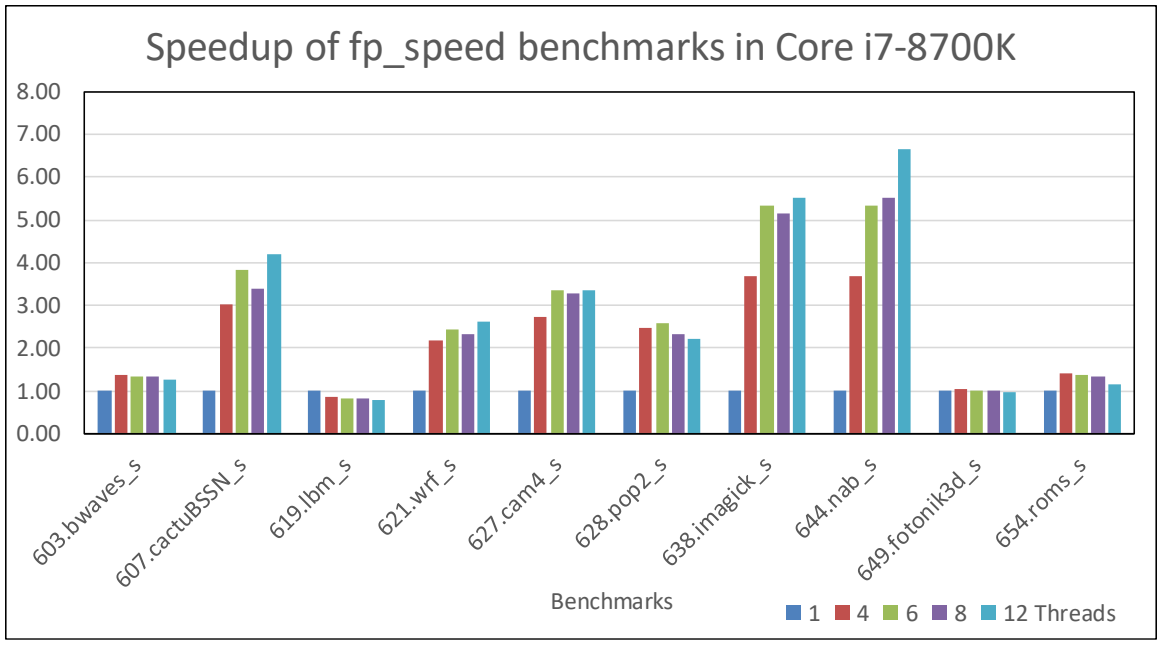

Figure 6.2 Speedup of *fp_speed* Benchmarks in Core i7-8700K

Table 6.4 Runtime and Speedup of *fp_speed* in Xeon E3-1240 V2

vidrak	Runtime (sec) & Speedup			
Xeon E3-1240 V2	*fp_speed*			
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads
603.bwaves_s	1,632.91	1,130.93	1,181.82	1,174.38
	1.00	1.44	1.38	1.39
607.cactuBSSN_s	2,883.80	878.94	1,110.46	898.42
	1.00	3.28	2.60	3.21
619.lbm_s	1,391.80	1,196.80	1,224.25	1,239.48
	1.00	1.16	1.14	1.12
621.wrf_s	1,685.65	695.28	738.19	686.13
	1.00	2.42	2.28	2.46
627.cam4_s	2,320.91	787.88	737.74	645.07
	1.00	2.95	3.15	3.60
628.pop2_s	2,024.61	678.29	772.91	714.37
	1.00	2.98	2.62	2.83
638.imagick_s	7,330.61	1,986.60	2,093.62	2,040.22
	1.00	3.69	3.50	3.59
644.nab_s	3,122.68	846.98	770.63	694.42
	1.00	3.69	4.05	4.50
649.fotonik3d_s	1,059.71	837.83	852.61	864.52
	1.00	1.26	1.24	1.23
654.roms_s	2,985.47	1,716.92	1,759.35	1,848.40
	1.00	1.74	1.70	1.62

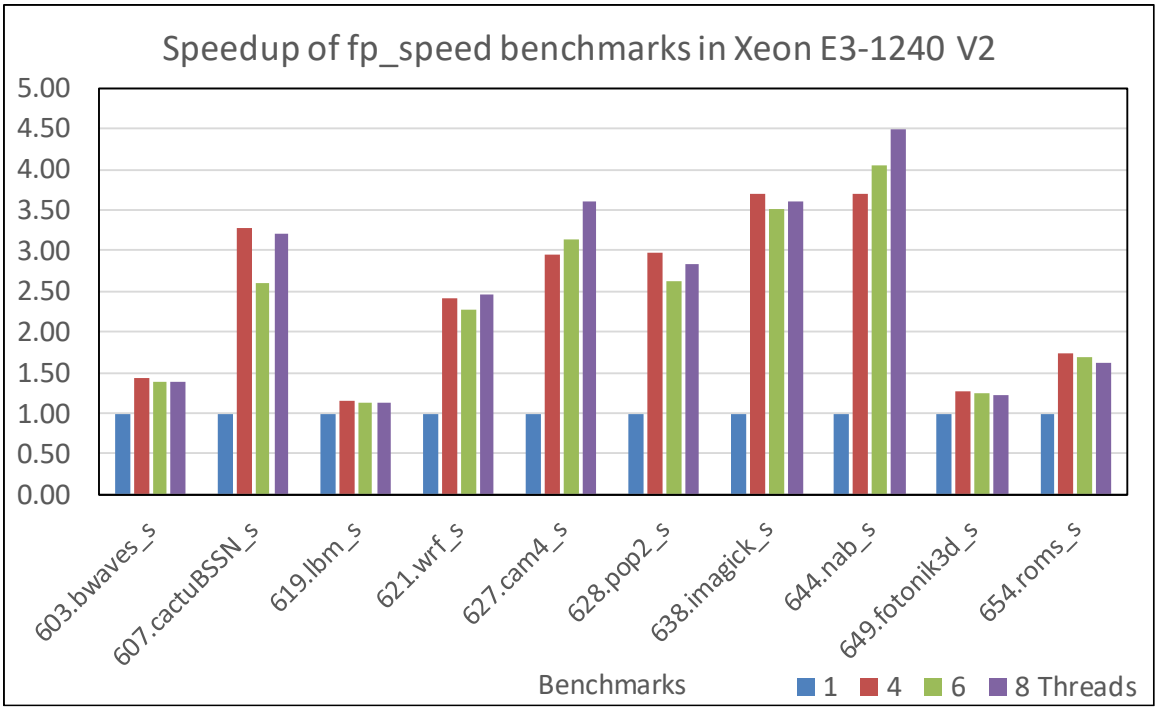

Figure 6.3 Speedup of *fp_speed* Benchmarks in Xeon E3-1240 V2

Table 6.5 Runtime and Speedup of *fp_speed* in Xeon E5-2643 V3

mtleconte	Runtime (sec) & Speedup						
Xeon E5-2643 V3	*fp_speed*						
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads	12 Threads	16 Threads	24 Threads
603.bwaves_s	1,828.46	886.79	839.81	845.87	866.43	920.33	849.07
	1.00	2.06	2.18	2.16	2.11	1.99	2.15
607.cactuBSSN_s	2,436.50	785.27	565.66	483.16	380.54	409.52	344.57
	1.00	3.10	4.31	5.04	6.40	5.95	7.07
619.lbm_s	1,628.31	874.78	865.50	899.95	1,061.79	1,115.64	898.32
	1.00	1.86	1.88	1.81	1.53	1.46	1.81
621.wrf_s	1,392.00	525.30	438.72	397.58	328.70	350.52	404.54
	1.00	2.65	3.17	3.50	4.23	3.97	3.44
627.cam4_s	2,167.90	754.33	577.92	493.35	434.77	478.08	430.53
	1.00	2.87	3.75	4.39	4.99	4.53	5.04
628.pop2_s	1,842.77	602.11	478.38	426.14	409.60	450.80	483.81
	1.00	3.06	3.85	4.32	4.50	4.09	3.81
638.imagick_s	6,070.80	1,609.45	1,094.47	832.54	571.84	603.60	538.80
	1.00	3.77	5.55	7.29	10.62	10.06	11.27
644.nab_s	2,677.77	702.61	477.30	360.92	249.53	241.30	222.69
	1.00	3.81	5.61	7.42	10.73	11.10	12.02
649.fotonik3d_s	1,255.07	654.24	640.33	642.09	675.53	709.80	701.89
	1.00	1.92	1.96	1.95	1.86	1.77	1.79
654.roms_s	2,529.91	960.34	812.16	754.57	759.95	756.55	1,053.77
	1.00	2.63	3.12	3.35	3.33	3.34	2.40

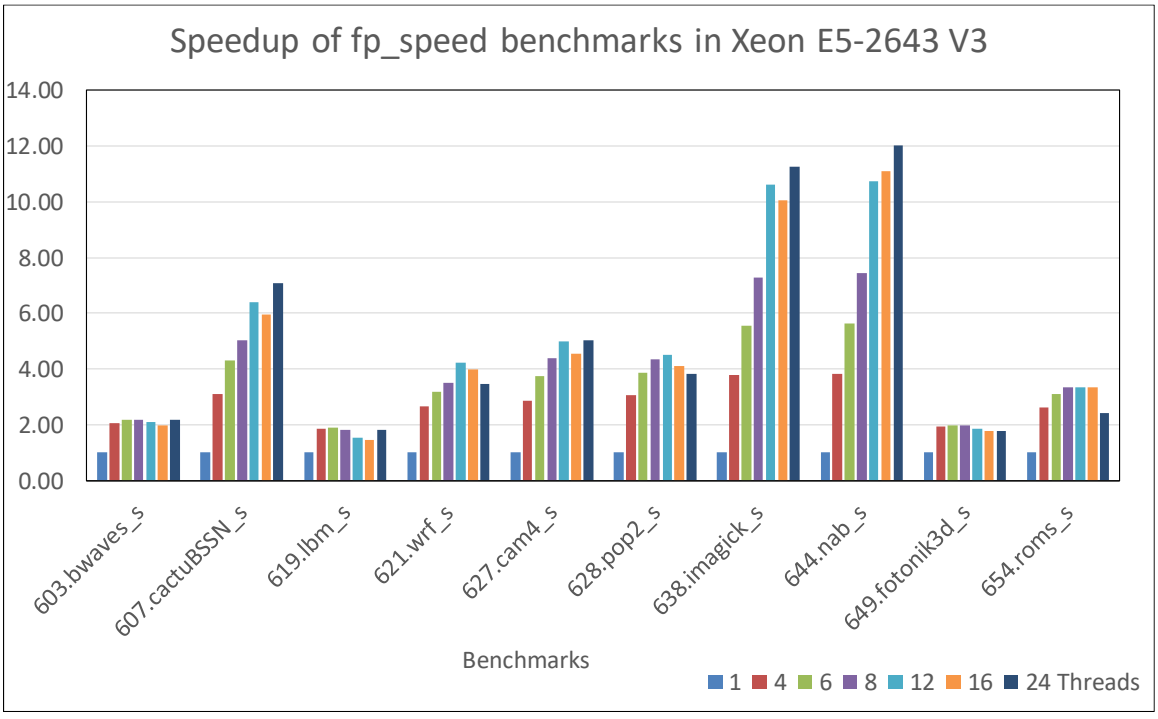

Figure 6.4 Speedup of *fp_speed* Benchmarks in Xeon E5-2643 V3

Table 6.6 shows the *SPECspeed2017_fp_base* number across all the test systems when the number of threads is varied. Conforming to the previous individual benchmark speedups, all the test systems show the best performance when the number of threads is equal to the number of physical cores. The Core processors outperform the Xeon processors on the single threaded benchmarks, partly because the Core processors operate at a higher frequency. Figure 6.5 gives a visual representation of the *SPECspeed2017_fp_base* numbers for different thread counts on all the test systems.

Table 6.6 *SPECspeed2017_fp_base* on all Test Systems

fp_speed				
	mtsano	*clingmansdome*	*vidrak*	*mtleconte*
Thread Count	Core i7-4770	Core i7-8700K	Xeon E3-1240 V2	Xeon E5-2643 V3
1	7.06	10.81	6.079	6.44
4	15.01	21.69	13.75	17.37
6	14.64	24.45	13.18	21.29
8	14.95	23.62	13.93	23.78
12		24.22		26.53
16				25.22
24				25.82

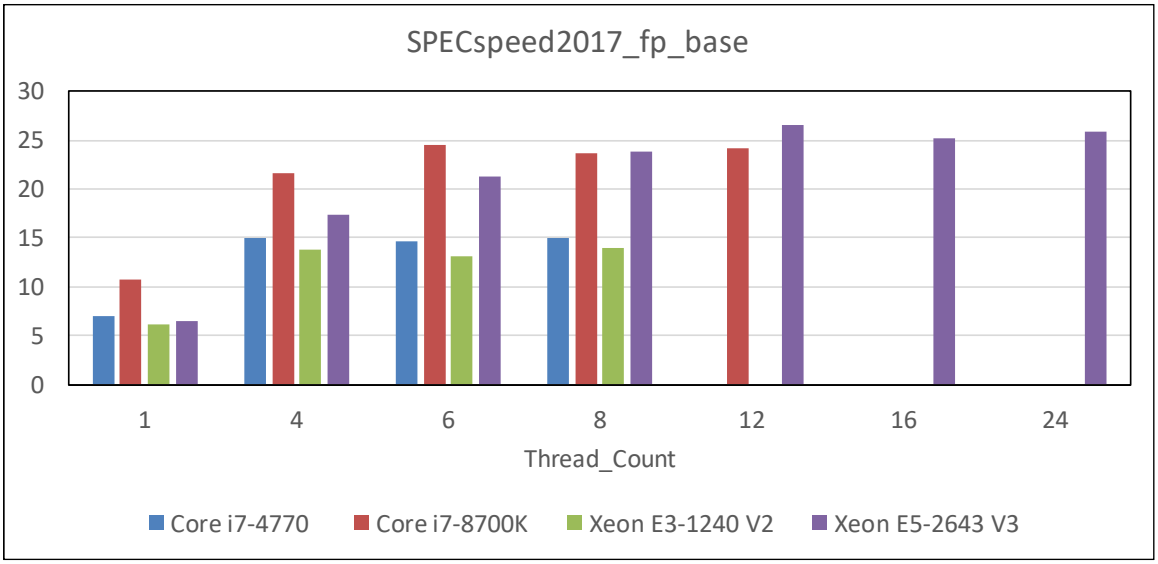

Figure 6.5 *SPECspeed2017_fp_base* Results on all Test Systems

A performance comparison across the test machines is made in Table 6.7. The speedups here are defined in Eq.6.3 and Eq.6.4. While comparing the two core i7's (*mtsano and clingmansdome),* the speedup of *clingmansdome* over *mtsano* is substantial, signifying that combination of architectural improvements and faster memory. All of which contribute to ~50% gains in performance, however, the scaled speedup shows that a major portion of this gains come for a higher clock frequency. While comparing the Xeons, it should be noted that the difference in the operating clock frequency is relatively small and *vidrak* has a faster clock than *mtleconte.* However, *mtleconte* outperforms *vidrak* and shows a speedup of ~10% for single-threaded applications and ~75% for eight threaded applications. Cross-comparison of machines in the case of *vidrak* and *mtleconte,* the newer architecture in *mtsano* contributed to ~20% gains. In the case of the two hexa-core machines, *clingmansdome and mtleconte,* the newer i7 fails to keep up when thread-count increases because the speed benchmarks are largely memory oriented and *mtleconte* has a bigger last level cache. Figure 6.6 and Figure 6.7 show excerpts from auto-generated reports from SPEC CPU2017 runs of the *fp_speed* benchmarks on *clingmansdome* for one and six threads, respectively.

Table 6.7 Relative Speedups for *fp_speed* on Different Test Machines.

fp_speed								
	Clingmansdome to mtsano		*mtsano to vidrak*		*mtleconte to vidrak*		*clingmansdome to mtleconte*	
Thread Count	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup
1	1.53	1.21	1.16	1.19	1.06	1.10	1.68	1.30
4	1.45	1.09	1.09	1.19	1.26	1.27	1.25	1.02
6	1.67	1.27	1.11	1.22	1.62	1.66	1.15	0.93
8	1.58	1.17	1.07	1.20	1.71	1.75	0.99	0.81
12							0.91	0.75

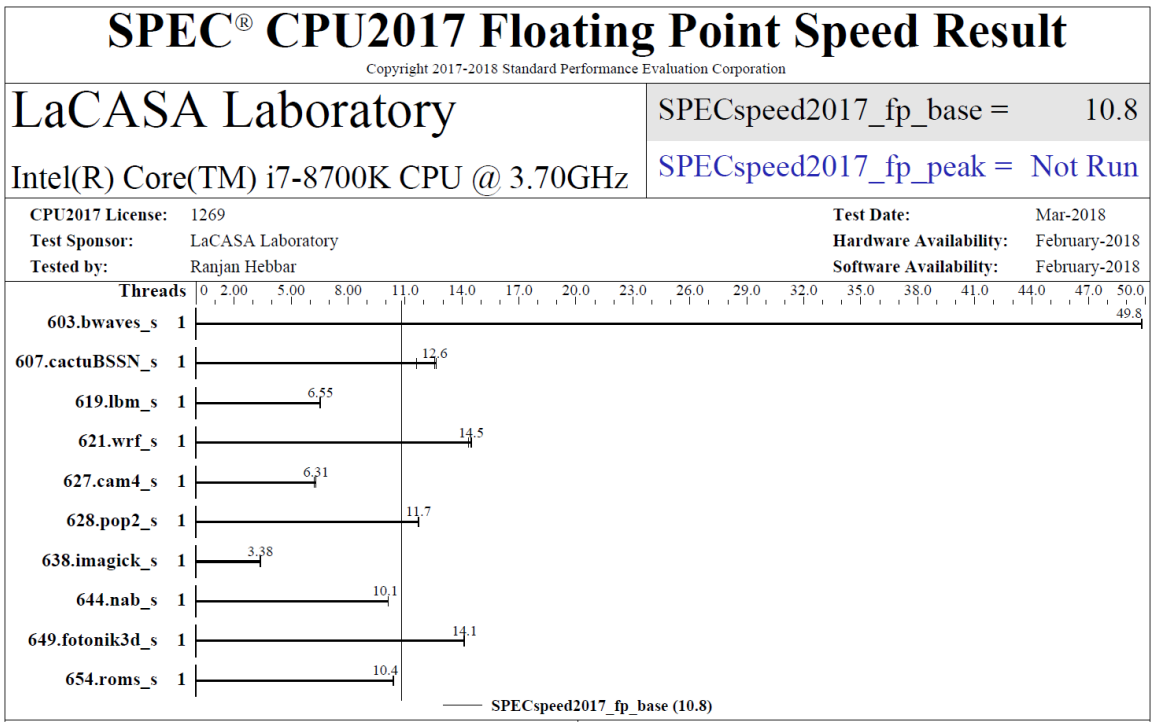

Figure 6.6 SPEC CPU2017 Report Excerpt for Single-Threaded *fp_speed*

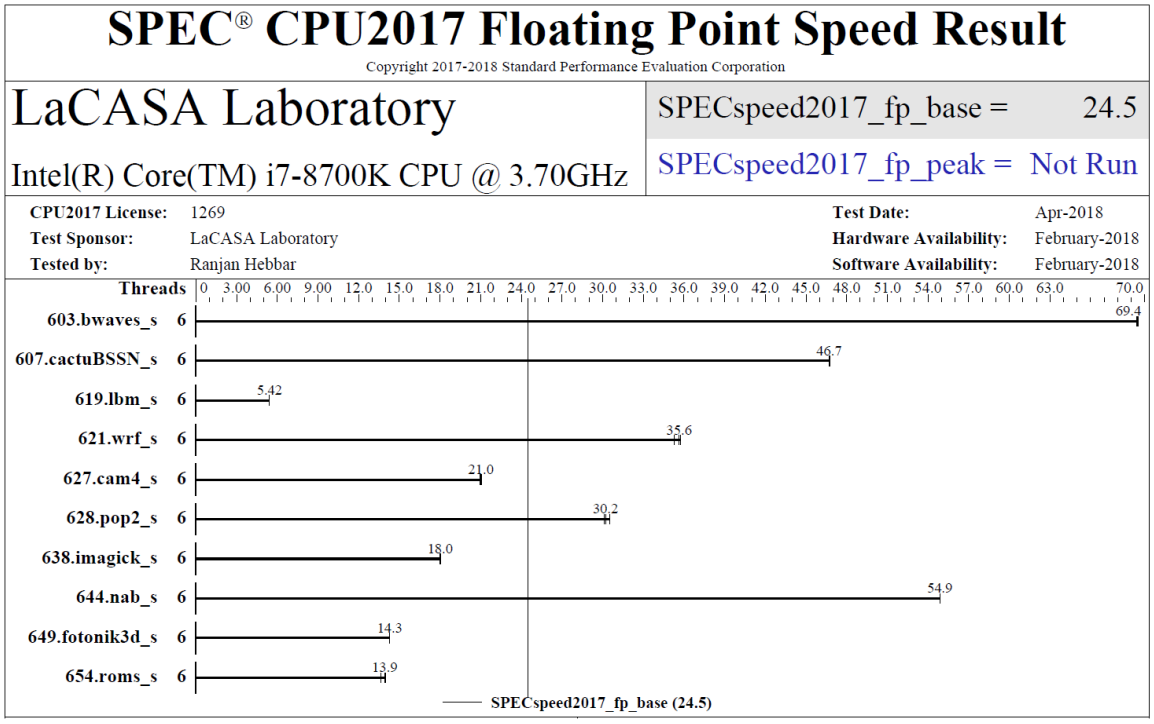

Figure 6.7 SPEC CPU2017 Report Excerpt for Six-Threaded *fp_speed*

6.1.2 SPEC CPU2017 *int_speed*

Table 6.8 and Figure 6.8 show the execution time and speedups for *int_speed* on *mtsano* (Core i7-4770) when the number of threads is 1, 4, 6, and 8. The performance of 8-thread benchmarks is found to be the best, with speedup ranging from 1 (all except 657.xz_s) to 3.53 (657.xz_s). Table 6.9 and Figure 6.9 show the execution time and speedups for *int_speed* on *clingmansdome* (Core i7-8700k) when the number of threads is 1, 4, 6, 8, and 12. The performance of 12-thread benchmarks is found to be the best, with speedup ranging from 1 (All except 657.xz_s) to 3.56 (657.xz_s).

Table 6.10 and Figure 6.10 show the execution time and speedups for *int_speed* on *vidrak* (Xeon E3-1240) when the number of threads is 1, 4, 6, and 8. The performance of 8-thread benchmarks is found to be the best, with speedup ranging from 1 (all benchmarks except 657.xz_s) to 3.39 (657.xz_s). Table 6.11 and Figure 6.11 give the execution time and speedups of *int_speed* on *mtleconte* (Xeon E5-2643 V3) when the number of threads is 1, 2, 4, 6, 12, 16, and 24. The performance of 24-thread benchmarks is found to be the best, ranging from 1 (all benchmarks except 657.xz_s) to 5.08 (657.xzs).

It is observed that all but one benchmark (657.xz_s) is not parallelizable in the entire *int_speed* benchmark suite. All other benchmark behavior is comparable across all the test machines with only one software thread created irrespective of the number of threads allocated. The benchmark that parallelizes scales well for increasing thread count to show speedup. The best speedup is obtained when the software threads occupy all the logical cores.

Table 6.8 Runtime and Speedup of *int_speed* in Core i7-4770

mtsano	Runtime (sec) & Speedup			
Core i7-4770	*int_speed*			
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads
600.perlbench_s	334.27	336.24	334.94	334.90
	1.00	0.99	1.00	1.00
602.gcc_s	473.69	473.64	473.79	474.24
	1.00	1.00	1.00	1.00
605.mcf_s	395.85	396.04	397.40	396.38
	1.00	1.00	1.00	1.00
620.omnetpp_s	400.26	399.41	400.09	400.16
	1.00	1.00	1.00	1.00
623.xalancbmk_s	281.93	280.93	282.35	283.29
	1.00	1.00	1.00	1.00
625.x264_s	169.35	168.89	169.23	169.18
	1.00	1.00	1.00	1.00
631.deepsjeng_s	276.59	275.10	276.21	276.24
	1.00	1.01	1.00	1.00
641.leela_s	396.00	395.28	396.08	395.79
	1.00	1.00	1.00	1.00
648.exchange2_s	245.80	247.24	246.71	246.68
	1.00	0.99	1.00	1.00
657.xz_s	2,115.93	795.29	705.45	598.69
	1.00	2.66	3.00	3.53

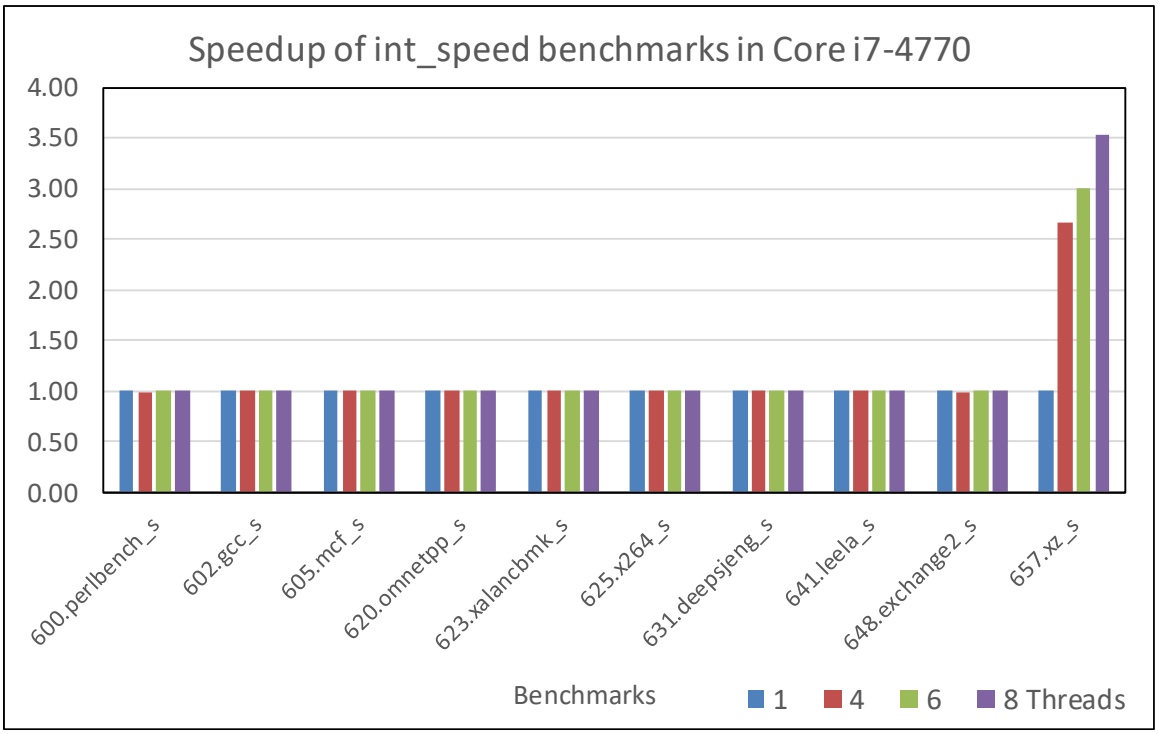

Figure 6.8 Speedup of *int_speed* Benchmarks in Core i7-4770

Table 6.9 Runtime and Speedup of *int_speed* in Core i7-8700K

clingmansdome	Runtime (sec) & Speedup				
Core i7-8700K	*int_speed*				
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads	12 Threads
600.perlbench_s	241.60	240.31	244.20	240.97	242.38
	1.00	1.01	0.99	1.00	1.00
602.gcc_s	350.72	351.70	350.74	351.43	351.62
	1.00	1.00	1.00	1.00	1.00
605.mcf_s	315.59	312.98	314.21	314.34	313.62
	1.00	1.01	1.00	1.00	1.01
620.omnetpp_s	306.55	306.56	307.89	309.26	308.81
	1.00	1.00	1.00	0.99	0.99
623.xalancbmk_s	210.50	208.89	212.15	210.64	211.38
	1.00	1.01	0.99	1.00	1.00
625.x264_s	119.05	119.24	119.07	119.29	119.27
	1.00	1.00	1.00	1.00	1.00
631.deepsjeng_s	218.27	218.29	218.31	218.45	218.34
	1.00	1.00	1.00	1.00	1.00
641.leela_s	332.77	332.89	332.76	332.79	333.06
	1.00	1.00	1.00	1.00	1.00
648.exchange2_s	195.10	195.30	196.61	196.45	195.64
	1.00	1.00	0.99	0.99	1.00
657.xz_s	1,757.88	682.49	567.39	511.06	493.56
	1.00	2.58	3.10	3.44	3.56

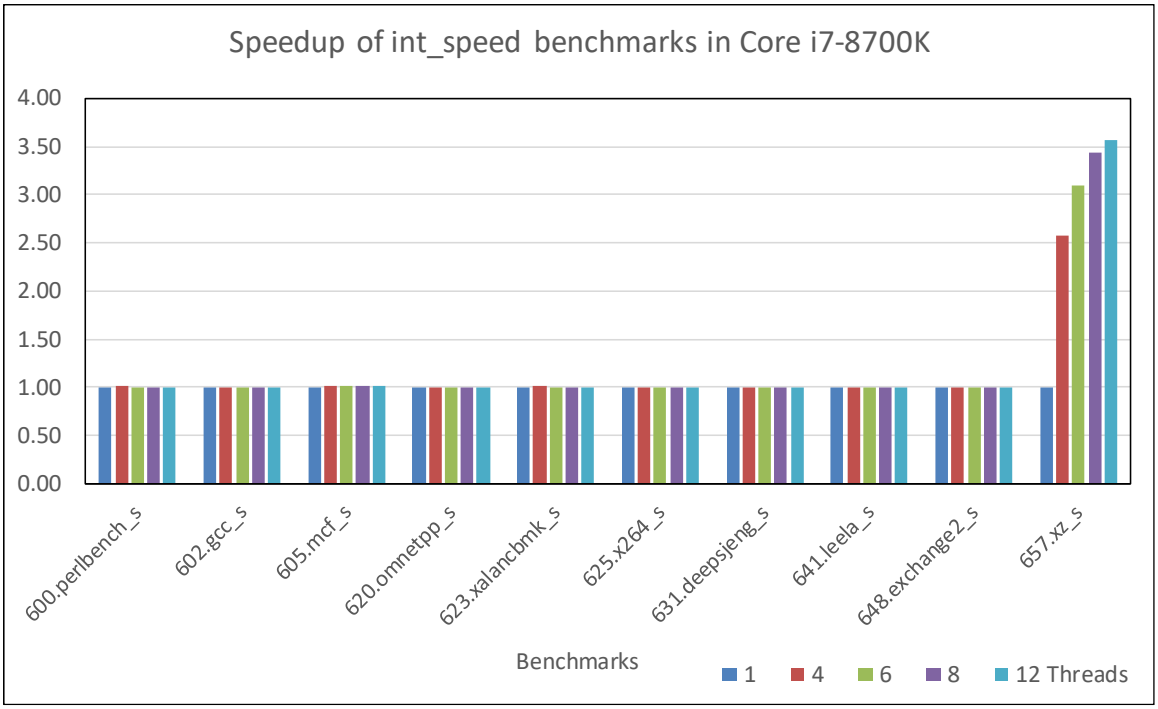

Figure 6.9 Speedup of *int_speed* Benchmarks in Core i7-8770K

Table 6.10 Runtime and Speedup of *int_speed* in Xeon E3-1240 V2

vidrak	Runtime (sec) & Speedup			
Xeon E3-1240 V2	*int_speed*			
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads
600.perlbench_s	441.33	438.57	440.29	441.54
	1.00	1.01	1.00	1.00
602.gcc_s	522.28	521.72	521.45	521.67
	1.00	1.00	1.00	1.00
605.mcf_s	472.81	473.17	472.25	473.61
	1.00	1.00	1.00	1.00
620.omnetpp_s	437.90	436.11	434.84	436.91
	1.00	1.00	1.01	1.00
623.xalancbmk_s	325.90	326.03	325.71	326.89
	1.00	1.00	1.00	1.00
625.x264_s	249.28	250.11	247.74	249.53
	1.00	1.00	1.01	1.00
631.deepsjeng_s	308.77	308.36	308.86	308.54
	1.00	1.00	1.00	1.00
641.leela_s	414.05	413.73	415.11	413.88
	1.00	1.00	1.00	1.00
648.exchange2_s	365.59	365.60	365.59	365.70
	1.00	1.00	1.00	1.00
657.xz_s	2,054.42	792.70	728.94	606.30
	1.00	2.59	2.82	3.39

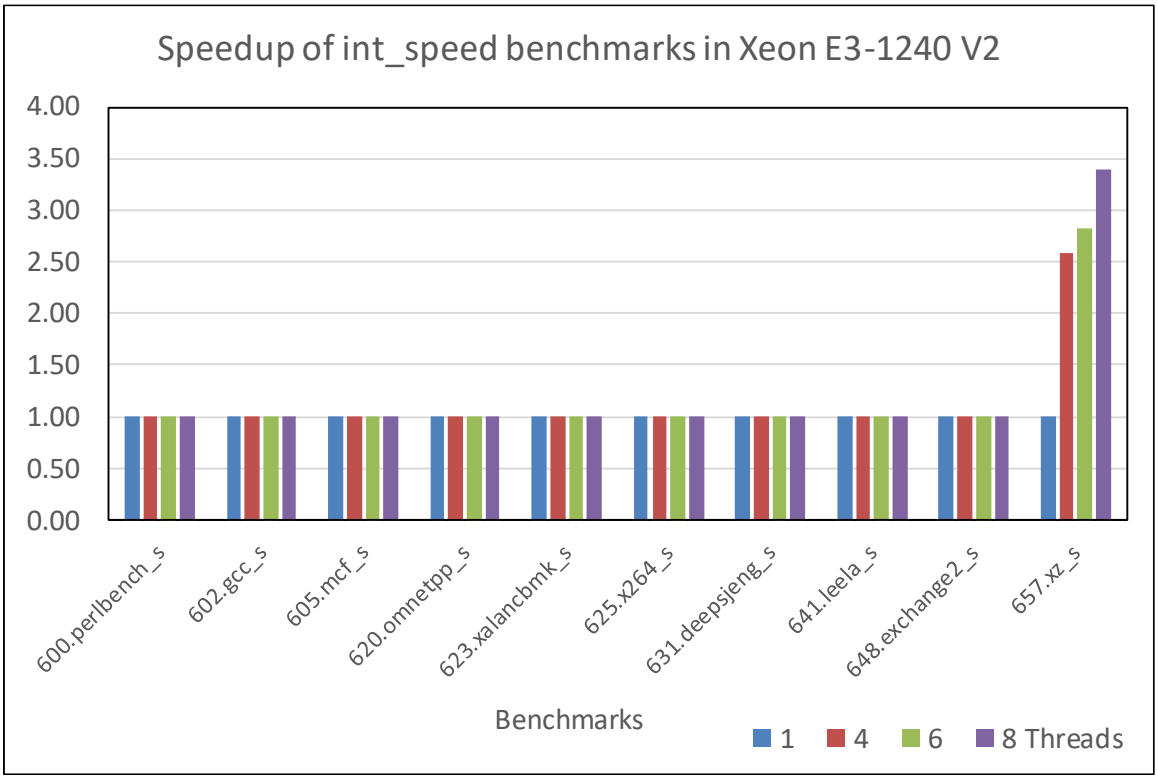

Figure 6.10 Speedup of *int_speed* Benchmarks in Xeon E3-1240 V2

Table 6.11 Runtime and Speedup of *int_speed* in Xeon E5-2643 V3

mtleconte	Runtime (sec) & Speedup						
Xeon E5-2643 V3	*int_speed*						
Benchmarks	1 Thread	4 Threads	6 Threads	8 Threads	12 Threads	16 Threads	24 Threads
600.perlbench_s	330.79	332.26	331.42	329.12	338.65	336.83	330.46
	1.00	1.00	1.00	1.01	0.98	0.98	1.00
602.gcc_s	481.86	482.77	486.51	483.50	489.00	485.84	483.20
	1.00	1.00	0.99	1.00	0.99	0.99	1.00
605.mcf_s	473.31	473.18	474.00	473.83	474.08	474.28	474.13
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
620.omnetpp_s	432.17	408.25	430.89	431.00	385.48	396.27	382.88
	1.00	1.06	1.00	1.00	1.12	1.09	1.13
623.xalancbmk_s	316.87	316.26	315.35	315.65	315.40	313.53	318.42
	1.00	1.00	1.00	1.00	1.00	1.01	1.00
625.x264_s	174.57	173.62	174.83	173.58	173.48	174.03	174.16
	1.00	1.01	1.00	1.01	1.01	1.00	1.00
631.deepsjeng_s	295.55	298.93	294.38	294.69	294.37	308.03	310.78
	1.00	0.99	1.00	1.00	1.00	0.96	0.95
641.leela_s	416.04	416.08	415.86	416.82	416.15	416.12	417.11
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
648.exchange2_s	259.88	259.98	258.94	258.81	259.36	259.04	259.34
	1.00	1.00	1.00	1.00	1.00	1.00	1.00
657.xz_s	1,972.62	732.93	614.01	496.76	469.65	403.15	388.38
	1.00	2.69	3.21	3.97	4.20	4.89	5.08

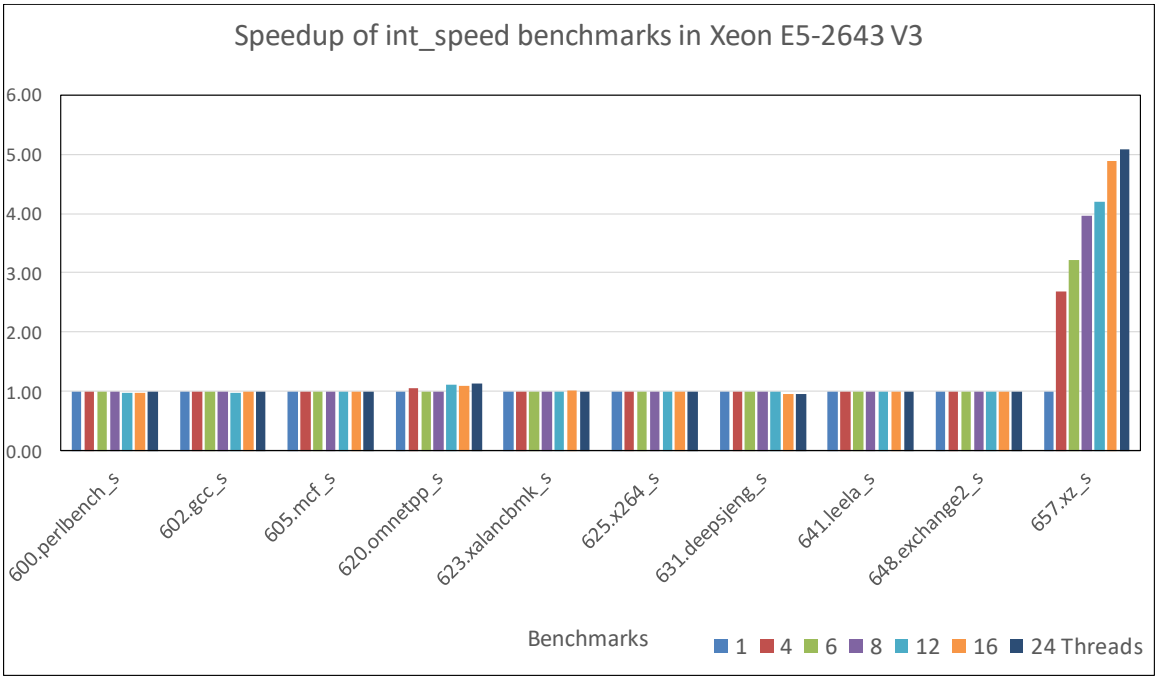

Figure 6.11 Speedup of *int_speed* Benchmarks in Xeon E5-2643 V3

Table 6.12 shows the *SPECspeed2017_int_ba*se number across all the test systems when the number of threads is varied. The best performance was obtained for the execution when the number of software threads equaled the number of logical cores. Because only one benchmark shows improvements by increasing the number of threads, multi-thread gains were minimal across machines. Figure 6.5 gives a visual representation of the *SPECspeed2017_int_base* numbers for different thread counts on all the test systems.

Table 6.12 *SPECspeed2017_int_base* on all Test Systems

int_speed				
	mtsano	*clingmansdome*	*vidrak*	*mtleconte*
Thread_Count	Core i7-4770	Core i7-8700K	Xeon E3-1240 V2	Xeon E5-2643 V3
1	6.23	8.07	5.26	5.92
4	6.87	8.89	5.79	6.57
6	6.95	9.01	5.84	6.65
8	7.06	9.12	5.94	6.81
12		9.15		6.90
16				6.96
24				7.01

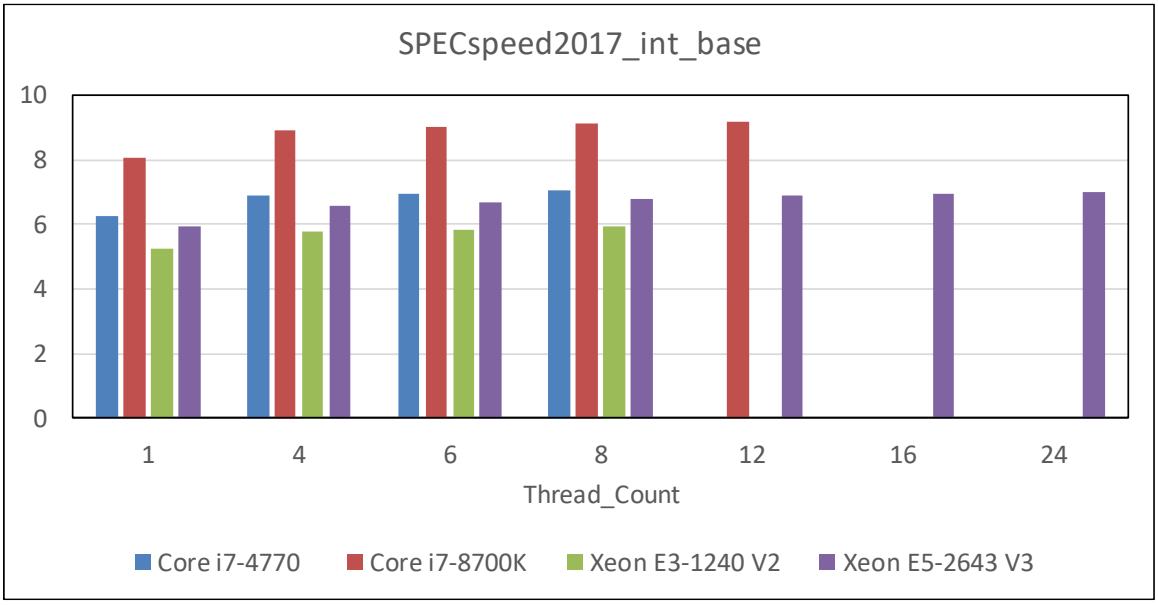

Figure 6.12 *SPECspeed2017_int_base* Results on all Test Systems

A performance comparison across the test machines is made in Table 6.13. The speedups here are defined in Eq.6.3 and Eq.6.4. While comparing the two i7's, *clingmansdome* has a speedup of ~30% over *mtsano*. However, the scaled speedup signifies architectural and memory improvements contributing to ~8% of the total speedup. The clock frequency is the biggest factor influencing performance. In comparing the Xeons, though *vidrak* has a faster clock, it lags in performance by ~18% because it has an older architecture. Cross-comparison between the Xeons and the Core processors show that the Core processors outperform the Xeons with and without the higher clock frequencies. It should be noted that the Xeons in either comparison are from an older generation than the Core processors. Between *mtsano and vidrak* with the newer architecture, *mtsano* has a speedup of ~16%. A closer equivalence is drawn between *clingmansdome* and *mtleconte* showing minimal changes to the integer functional unit. This also reinforces the fact that the design concentration for a Xeon is energy and reliability whereas for a core processor it is performance. Figure 6.13 and Figure 6.14 show excerpts from auto-generated reports from SPEC CPU2017 runs of the *int_speed* benchmarks on *clingmansdome* for one and six threads, respectively.

Table 6.13 Relative Speedups for *int_speed* on Different Test Machines

int_speed								
	clingmansdome to mtsano		*mtsano to vidrak*		*mtleconte to vidrak*		*clingmansdome to mtleconte*	
Thread Count	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup
1	1.30	1.07	1.18	1.15	1.13	1.17	1.36	1.06
4	1.29	1.07	1.19	1.16	1.13	1.18	1.35	1.05
6	1.30	1.08	1.19	1.16	1.14	1.18	1.35	1.05
8	1.29	1.07	1.19	1.16	1.15	1.19	1.34	1.04
12							1.33	1.03

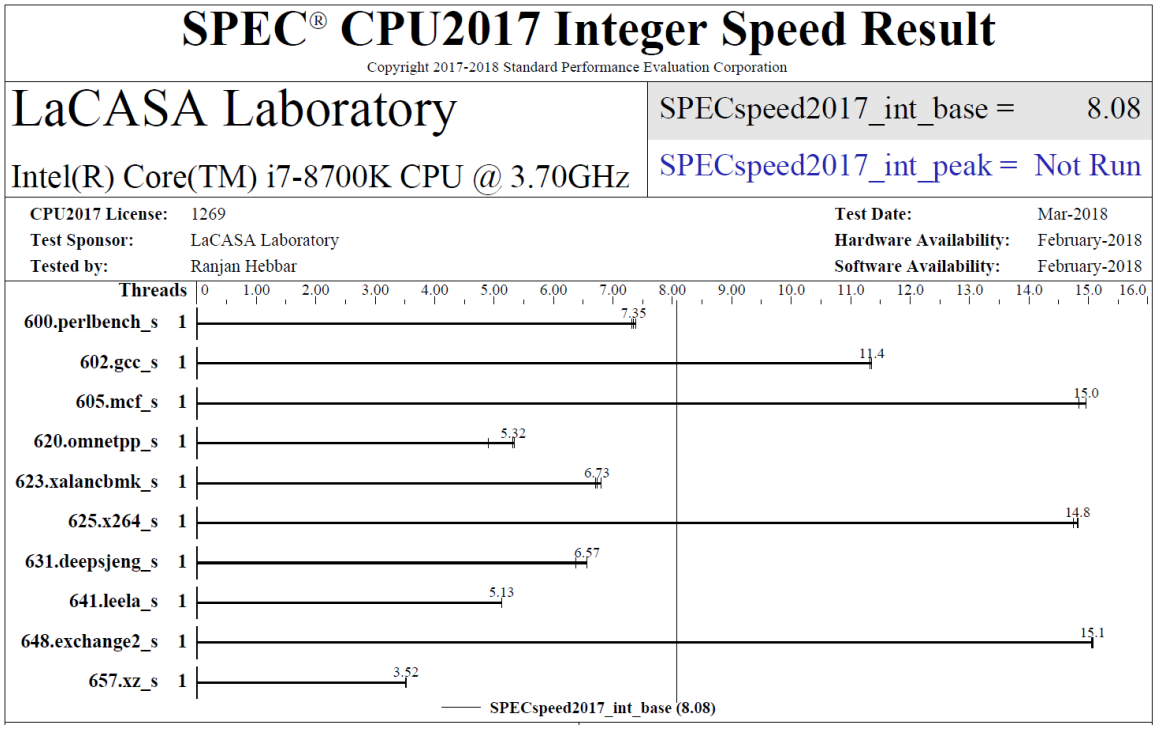

Figure 6.13 SPEC CPU2017 Report Excerpt for Single-Threaded *int_speed*

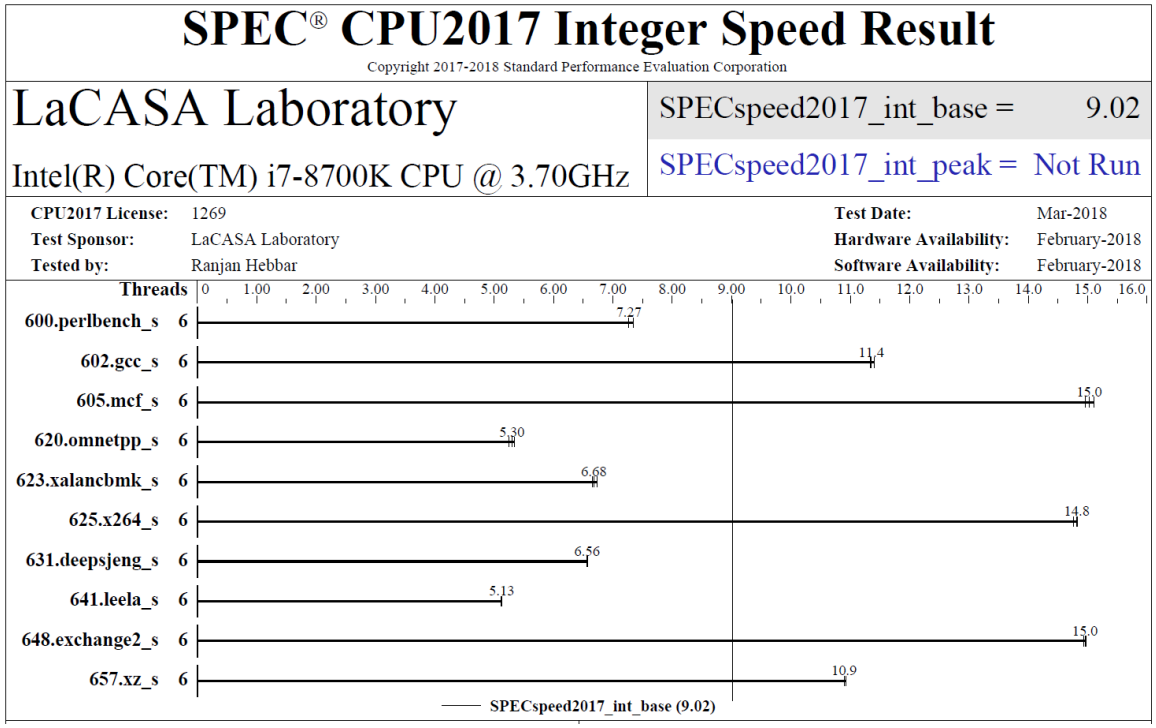

Figure 6.14 SPEC CPU2017 Report Excerpt for Six-Threaded *int_speed*

6.1.3 SPEC CPU2017 *fp_rate*

Table 6.14 and Figure 6.15 show the execution time and speedups for *fp_rate* on *mtsano* (Core i7-4770) when the number of copies is 1, 4, 6, 8. The performance of 8-copy benchmarks is found to be the best, with speedups ranging from 1.19 (503.bwaves_r) to 4.04 (544.roms_r and 511.povray_r). Table 6.15 and Figure 6.16 gives the execution time and speedups for *fp_rate* on *clingmansdome* (Core i7-8700k). The performance of 12-copy benchmarks is found to be the best, with speedups ranging from 0.95 (554.roms_r) to 7.00 (544.nab_r).

Table 6.16 and Figure 6.17 show execution time and speedups for *fp_rate* on *vidrak* (Xeon E3-1240) when the number of copies is 1, 4, 6, and 8. The performance of 8-copy benchmarks is found to be the best, with speedups ranging from 1.19 (503.bwaves_r) to 4.04 (544.roms_r and 511.povray_r). Table 6.17 and Figure 6.18 shows the execution time and speedups of *fp_rate* on *mtleconte* (Xeon E5-2643 V3). The performance of 24-copy benchmarks is found to be the best, with speedups ranging from 2.19 (503.bwaves_r) to 13.84 (544.nab_r).

The overview of the *fp_rate* benchmarks is given here in terms of scalability. The behavior of the rate benchmarks is similar across all test machines. Some benchmarks (507.cactuBSSN_r, 508.namd_r, 511.povray_r, 526.blender_r, 538.imagick_r and 544.nab_r) scale up until the number of copies match the number of logical cores. These benchmarks are not memory intensive and contention seems to not affect performance. The other set of benchmarks (503.bwaves_r, 510.parest_r, 519.lbm_r, 521.wrf_r, 527.cam4_r, 549.fotonik3d_r, and 554.roms_r), have initial speedup to multiple copies match the number of physical cores, however, a reduce in speedup is seen for an increase in copies thereafter.

Table 6.14 Runtime and Speedup of *fp_rate* in Core i7-4770

mtsano	Runtime (sec) & Speedup			
Core i7-4770	*fp_rate*			
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies
503.bwaves_r	250.01	806.82	1,235.16	1,685.09
	1.00	1.24	1.21	1.19
507.cactuBSSN_r	230.20	305.73	439.47	545.46
	1.00	3.01	3.14	3.38
508.namd_r	215.37	260.99	379.63	523.05
	1.00	3.30	3.40	3.29
510.parest_r	334.22	754.46	1,158.33	1,812.94
	1.00	1.77	1.73	1.47
511.povray_r	426.71	451.59	659.93	845.55
	1.00	3.78	3.88	4.04
519.lbm_r	185.24	399.95	607.34	835.03
	1.00	1.85	1.83	1.77
521.wrf_r	249.97	395.15	633.82	883.37
	1.00	2.53	2.37	2.26
526.blender_r	279.82	327.08	447.47	585.05
	1.00	3.42	3.75	3.83
527.cam4_r	257.22	333.11	494.29	680.14
	1.00	3.09	3.12	3.03
538.imagick_r	286.30	323.81	477.99	631.08
	1.00	3.54	3.59	3.63
544.nab_r	258.25	290.05	410.12	511.21
	1.00	3.56	3.78	4.04
549.fotonik3d_r	444.20	1,057.06	1,595.20	2,143.63
	1.00	1.68	1.67	1.66
554.roms_r	248.80	580.50	933.86	1,447.78
	1.00	1.71	1.60	1.37

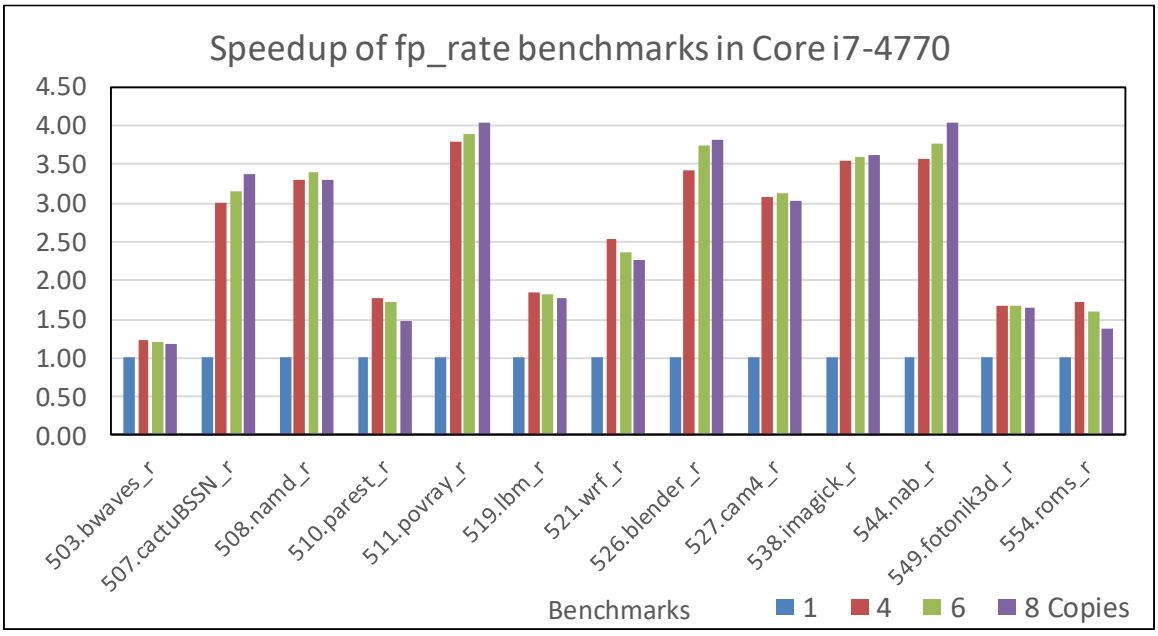

Figure 6.15 Speedup of *fp_rate* Benchmarks in Core i7-4770

Table 6.15 Runtime and Speedup of *fp_rate* in Core i7-8700K

clingmansdome	Runtime (sec) & Speedup				
Core i7-8700k	*fp_rate*				
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies	12 Copies
503.bwaves_r	178.28	615.51	948.56	1,294.44	2,033.67
	1.00	1.16	1.13	1.10	1.05
507.cactuBSSN_r	142.16	189.49	225.03	286.99	400.54
	1.00	3.00	3.79	3.96	4.26
508.namd_r	157.18	167.70	177.18	232.32	302.44
	1.00	3.75	5.32	5.41	6.24
510.parest_r	241.72	449.48	770.61	1,055.61	1,920.28
	1.00	2.15	1.88	1.83	1.51
511.povray_r	243.10	261.25	271.11	354.75	473.08
	1.00	3.72	5.38	5.48	6.17
519.lbm_r	76.08	284.51	444.09	603.56	932.68
	1.00	1.07	1.03	1.01	0.98
521.wrf_r	160.08	270.96	420.18	597.30	995.36
	1.00	2.36	2.29	2.14	1.93
526.blender_r	195.35	222.58	239.44	300.83	393.27
	1.00	3.51	4.90	5.19	5.96
527.cam4_r	153.28	192.75	247.51	352.10	621.46
	1.00	3.18	3.72	3.48	2.96
538.imagick_r	207.55	222.31	227.12	301.80	408.22
	1.00	3.73	5.48	5.50	6.10
544.nab_r	181.48	192.90	198.20	249.67	311.08
	1.00	3.76	5.49	5.82	7.00
549.fotonik3d_r	284.89	834.30	1,262.28	1,755.39	2,852.76
	1.00	1.37	1.35	1.30	1.20
554.roms_r	148.47	414.92	702.58	1,034.42	1,874.99
	1.00	1.43	1.27	1.15	0.95

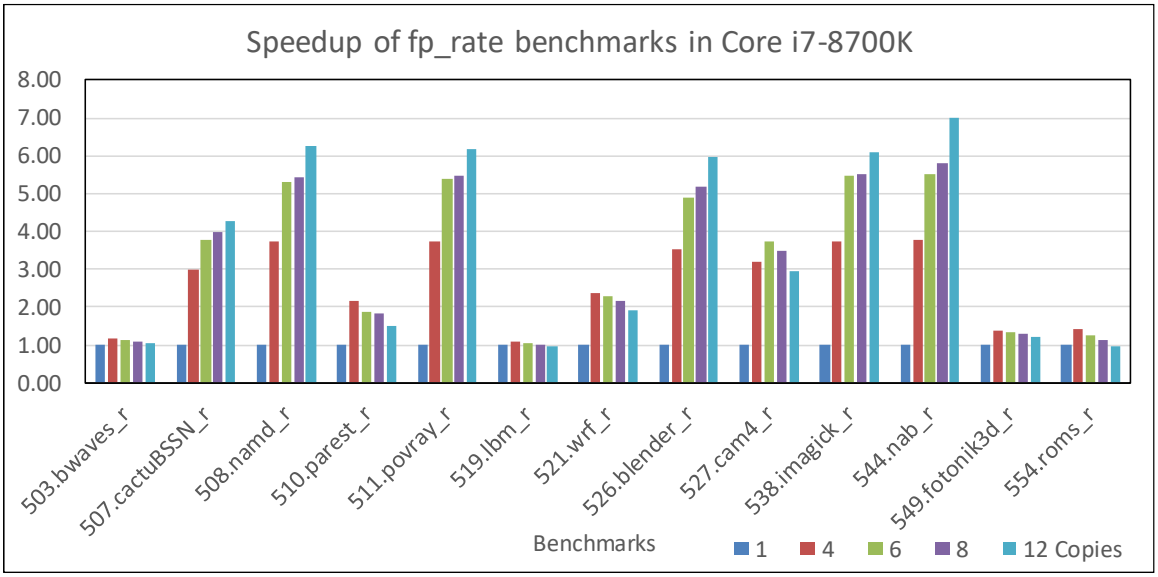

Figure 6.16 Speedup of *fp_rate* Benchmarks in Core i7-8700K

Table 6.16 Runtime and Speedup of *fp_rate* in Xeon E3-1240 V2

vidrak	Runtime (sec) & Speedup			
Xeon E3-1240 V2	*fp_rate*			
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies
503.bwaves_r	285.59	818.03	1,254.09	1,701.32
	1.00	1.40	1.37	1.34
507.cactuBSSN_r	272.36	345.51	534.03	657.41
	1.00	3.15	3.06	3.31
508.namd_r	290.80	313.98	444.93	540.10
	1.00	3.70	3.92	4.31
510.parest_r	362.63	773.01	1,257.86	1,855.45
	1.00	1.88	1.73	1.56
511.povray_r	397.98	427.60	618.67	748.35
	1.00	3.72	3.86	4.25
519.lbm_r	187.89	397.85	610.47	815.73
	1.00	1.89	1.85	1.84
521.wrf_r	295.22	403.93	637.17	879.49
	1.00	2.92	2.78	2.69
526.blender_r	284.77	327.44	455.42	522.98
	1.00	3.48	3.75	4.36
527.cam4_r	291.76	361.61	516.37	667.56
	1.00	3.23	3.39	3.50
538.imagick_r	445.54	475.41	740.15	938.71
	1.00	3.75	3.61	3.80
544.nab_r	323.29	343.46	487.43	553.26
	1.00	3.77	3.98	4.67
549.fotonik3d_r	476.99	1,081.50	1,632.74	2,175.05
	1.00	1.76	1.75	1.75
554.roms_r	281.40	597.33	1,005.40	1,512.13
	1.00	1.88	1.68	1.49

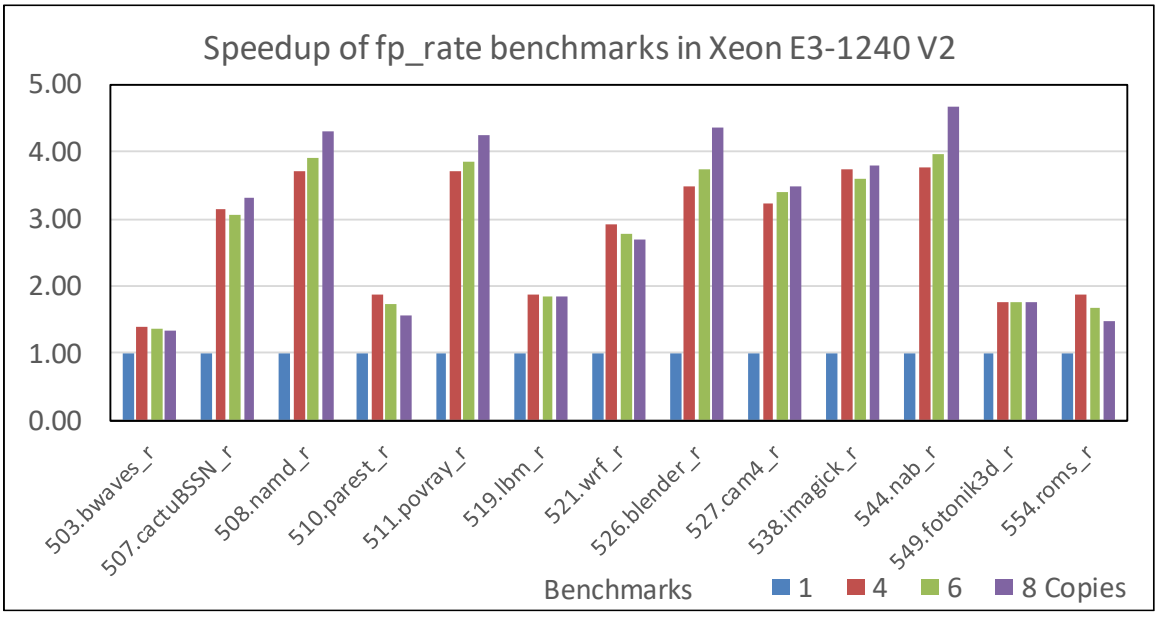

Figure 6.17 Speedup of *fp_rate* Benchmarks in Xeon E3-1240 V2

Table 6.17 Runtime and speedup of fp_rate in Xeon E5-2643 V3

mtleconte	Runtime (sec) & Speedup						
Xeon E5-2643 V3	*fp_rate*						
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies	12 Copies	16 Copies	24 Copies
503.bwaves_r	310.25	566.36	834.27	1,141.48	1,708.72	2,282.14	3,403.14
	1.00	2.19	2.23	2.17	2.18	2.18	2.19
507.cactuBSSN_r	275.16	301.63	333.71	370.02	431.03	580.16	746.15
	1.00	3.65	4.95	5.95	7.66	7.59	8.85
508.namd_r	228.86	236.12	240.05	245.78	244.55	322.62	424.55
	1.00	3.88	5.72	7.45	11.23	11.35	12.94
510.parest_r	332.61	341.51	370.35	449.50	954.85	1,547.69	3,249.53
	1.00	3.90	5.39	5.92	4.18	3.44	2.46
511.povray_r	340.04	345.71	351.25	352.61	361.10	487.94	666.46
	1.00	3.93	5.81	7.71	11.30	11.15	12.25
519.lbm_r	199.49	301.23	399.93	515.55	788.64	1,051.96	1,573.26
	1.00	2.65	2.99	3.10	3.04	3.03	3.04
521.wrf_r	246.30	292.84	347.66	421.51	690.33	982.91	1,638.33
	1.00	3.36	4.25	4.67	4.28	4.01	3.61
526.blender_r	274.16	284.25	295.51	301.75	320.82	403.68	523.74
	1.00	3.86	5.57	7.27	10.25	10.87	12.56
527.cam4_r	262.99	273.23	282.64	298.59	342.80	504.66	858.89
	1.00	3.85	5.58	7.05	9.21	8.34	7.35
538.imagick_r	291.05	293.97	303.06	304.32	306.16	417.37	548.26
	1.00	3.96	5.76	7.65	11.41	11.16	12.74
544.nab_r	273.03	276.96	283.32	291.07	290.76	384.50	473.32
	1.00	3.94	5.78	7.50	11.27	11.36	13.84
549.fotonik3d_r	527.19	777.76	1,075.80	1,399.02	2,072.87	2,752.13	4,146.49
	1.00	2.71	2.94	3.01	3.05	3.06	3.05
554.roms_r	252.19	320.71	454.31	616.20	985.20	1,374.74	2,387.06
	1.00	3.15	3.33	3.27	3.07	2.94	2.54

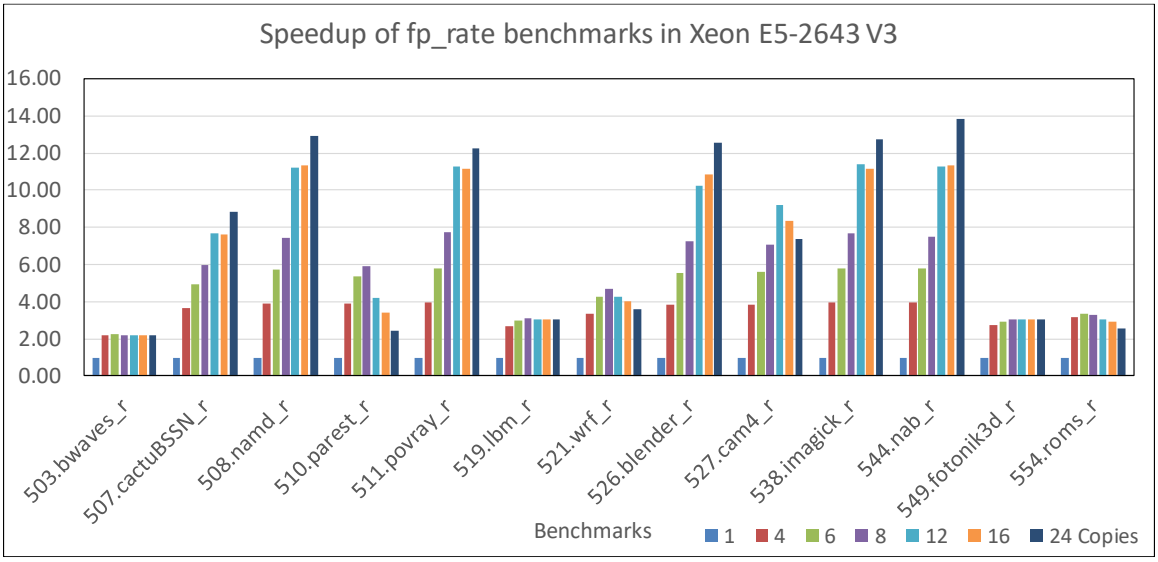

Figure 6.18 Speedup of *fp_rate* Benchmarks in Xeon E5-2643 V3

Table 6.18 shows the *SPECrate2017_fp_base* number across all the test systems for a different number of copies. The Core processors outperform the Xeon processors on single copy benchmarks partly due to the fact that the Core processors operate at a higher frequency. The best overall performance is observed when the number of copies matches the number of the physical core for each machine. Figure 6.5 is the visual representation of the *SPECspeed2017_fp_base* numbers for different thread counts on all the test systems.

Table 6.18 *SPECrate2017_fp_base* on all Test Systems

fp_rate				
	mtsano	*clingmansdome*	*vidrak*	*mtleconte*
Thread_Count	Core i7-4770	Core i7-8700K	Xeon E3-1240 V2	Xeon E5-2643 V3
1	7.52	11.82	6.55	7.22
4	18.81	28.20	17.42	24.61
6	18.92	32.49	17.30	32.00
8	18.53	32.14	17.82	37.27
12		32.02		43.33
16				42.11
24				42.58

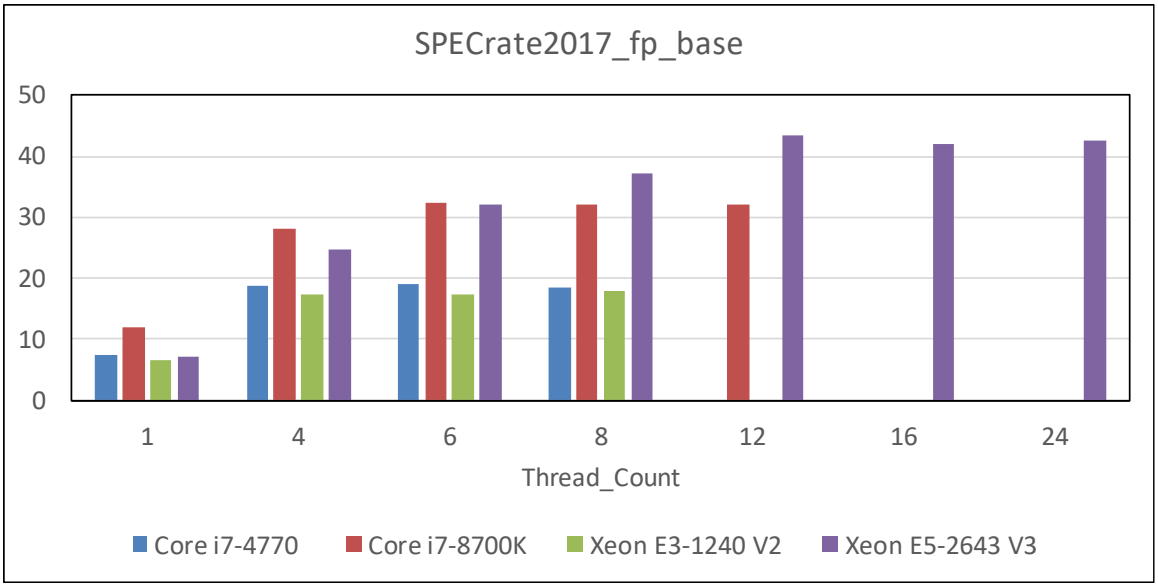

Figure 6.19 *SPECrate2017_fp_base* Results on all Test Systems

A performance comparison across the test machines is made in Table 6.19. The speedups here are defined in Eq.6.3 and Eq.6.4. While comparing the two core i7's (*mtsano and clingmansdome),* the speedup of *clingmansdome* over *mtsano* is substantial, ~50-70% gains in performance are observed. However, the scaled speedup shows that a major portion of this gains come for a higher clock frequency where architectural and memory advancements contributed to a speedup of ~20%. For the comparison of Xeons, *mtleconte* outperforms *vidrak* and shows a speedup of ~14% for single-copy application and a ~214% for eight-copy application because *mtleconte* has two physical hexa-cores whereas *vidrak* has a quad-core. Cross-comparison of machines in the case of *vidrak* and *mtleconte,* the newer architecture in *mtsano* contributed to ~20% gains. In the case of the two hexa-core machines, *clingmansdome and mtleconte,* the newer i7 fails to keep up when the number of copies increases because the *mtleconte* has two physical cores each of which has larger caches. Figure 6.20 and Figure 6.21 show excerpts from auto-generated reports from SPEC CPU2017 runs of the *fp_rate* benchmarks on *clingmansdome* for one and six threads, respectively.

Table 6.19 Relative Speedups for *fp_rate* on Different Test Machines.

fp_rate								
	clingmansdome to mtsano		*mtsano to vidrak*		*mtleconte to vidrak*		*clingmansdome to mtleconte*	
Copies	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup
1	1.57	1.24	1.15	1.18	1.10	1.14	1.64	1.27
4	1.50	1.13	1.08	1.17	1.41	1.42	1.15	0.93
6	1.72	1.31	1.09	1.20	1.85	1.90	1.02	0.83
8	1.73	1.29	1.04	1.16	2.09	2.14	0.86	0.70
12							0.74	0.61

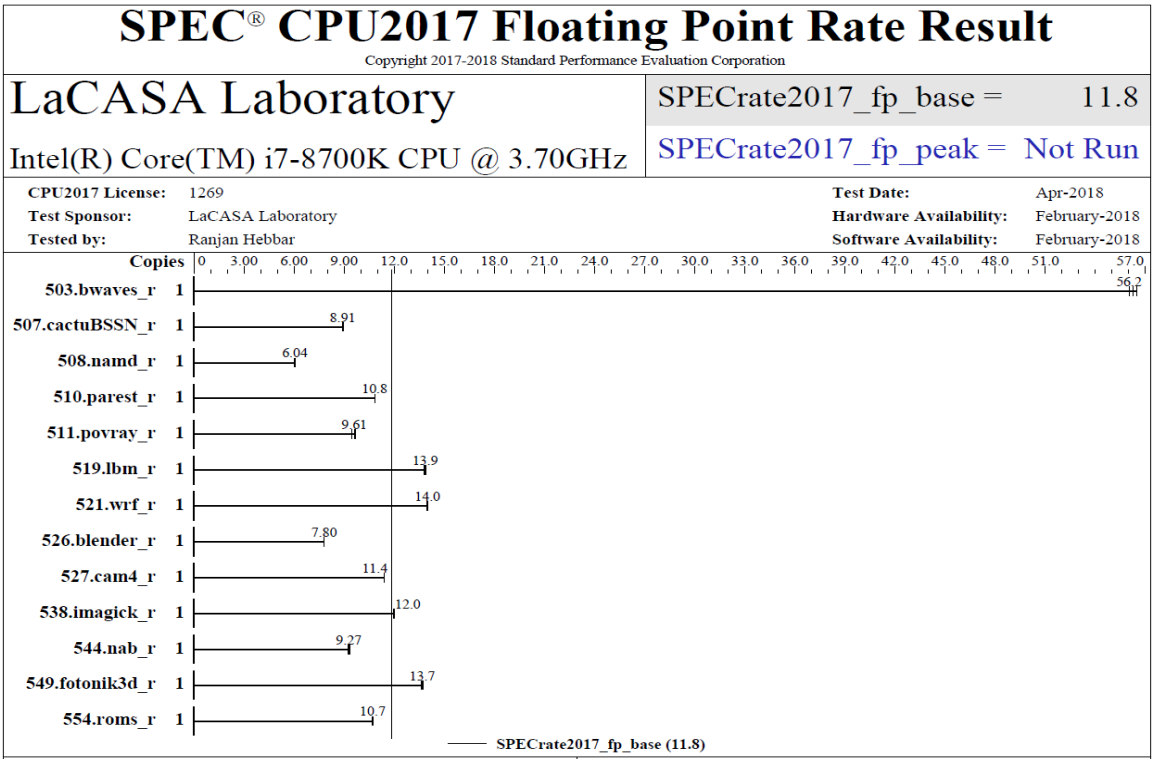

Figure 6.20 SPEC CPU2017 Report Excerpt for Single-Copy *fp_rate*

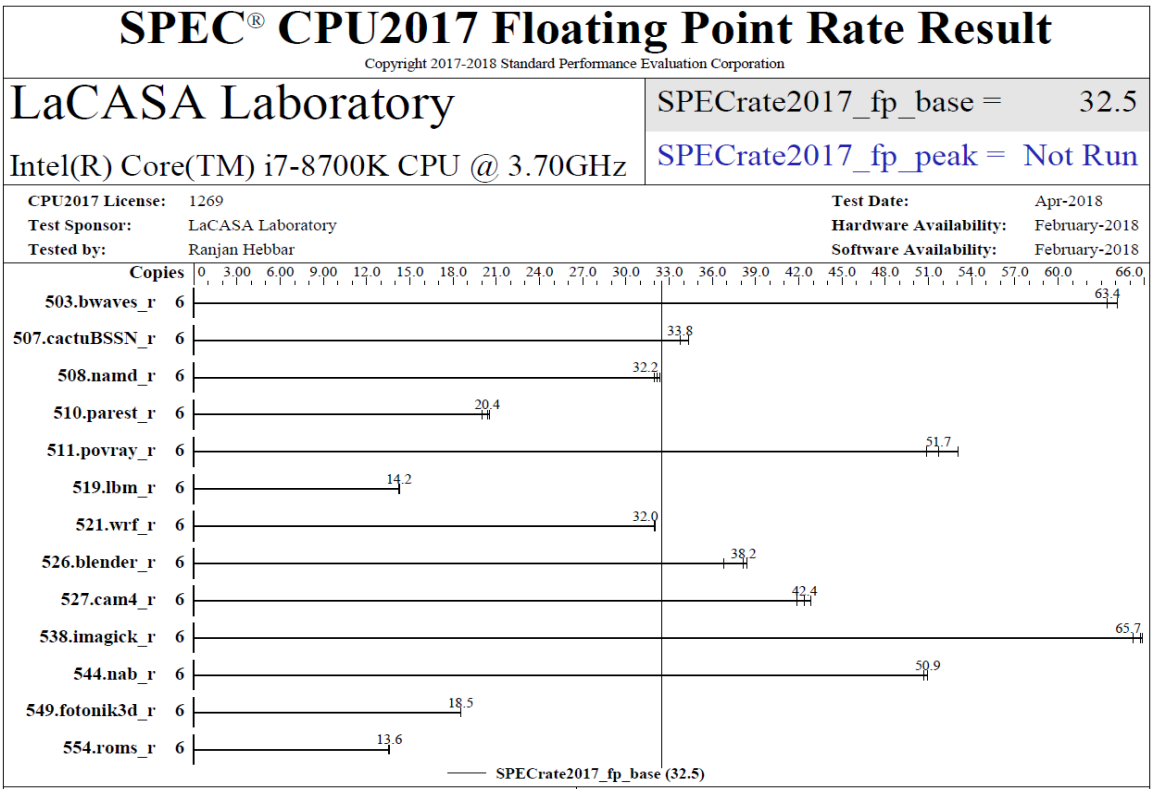

Figure 6.21 SPEC CPU2017 Report Excerpt Six-Copy *fp_rate*

6.1.4 SPEC CPU2017 *int_rate*

Table 6.20 and Figure 6.22 show execution time and speedup for *int_rate* on *mtsano* (Core i7-4770) for 1, 4, 6, and 8 copies. The performance of 8-copy benchmarks is found to be the best, with speedups ranging from 3.23 (502.gcc_r) to 4.35 (541.leela_r). Table 6.21 and Figure 6.23 show the execution time and speedup of *int_rate* on *clingmansdome* (Core i7-8700k). The performance of 12-copy benchmarks is found to be the best, with speedups ranging from 3.59 (520.omnetpp_r) to 7.35 (541.leela_r). Table 6.22 and Figure 6.24 shows execution time and speedup for *int_rate* on *vidrak* (Xeon E3-1240). The average speedup is found to be 3.26 for 4 copies, 4.31 for 6 copies and 6.59 for 8 copies. The performance of 8-copy benchmarks is found to be the best, with speedup ranging from 4.28 (505.mcf_r) to 7.44 (541.leela_r).

Table 6.23 and Figure 6.25 show the execution time and speedup of *int_rate* on *mtleconte* (Xeon E5-2643 V3). The performance of 8-copy benchmarks is found to be the best, with speedups ranging from 7.55 (505.mcf_r) to 14.92 (541.leela_r).

The overview of the *int_rate* benchmarks is given here in terms of scalability. The behavior of the rate benchmarks is similar across all test machines. All benchmarks show speedup till the number of copies matches the number of logical cores suggesting that the *int_rate* benchmarks are light and are not memory intensive. Though some benchmarks (505.gcc_r, 505.mcf_r, and 523.xalancbmk_r) have deterioration in performance when the system is fully loaded, i.e., the number of copies is the same as the number of logical cores.

Table 6.20 Runtime and Speedup of *int_rate* in Core i7-4770

mtsano	Runtime (sec) & Speedup			
Core i7-4770	*int_rate*			
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies
500.perlbench_r	334.28	428.50	631.28	789.01
	1.00	3.12	3.18	3.39
502.gcc_r	265.48	353.85	497.48	658.35
	1.00	3.00	3.20	3.23
505.mcf_r	210.16	284.07	398.15	502.69
	1.00	2.96	3.17	3.34
520.omnetpp_r	414.22	573.63	784.98	937.51
	1.00	2.89	3.17	3.53
523.xalancbmk_r	281.88	365.13	516.96	682.44
	1.00	3.09	3.27	3.30
525.x264_r	146.38	174.21	250.72	319.49
	1.00	3.36	3.50	3.67
531.deepsjeng_r	230.51	291.94	405.93	475.65
	1.00	3.16	3.41	3.88
541.leela_r	395.15	469.81	620.50	726.20
	1.00	3.36	3.82	4.35
548.exchange2_r	246.46	295.67	428.94	542.57
	1.00	3.33	3.45	3.63
557.xz_r	326.21	433.49	575.24	657.86
	1.00	3.01	3.40	3.97

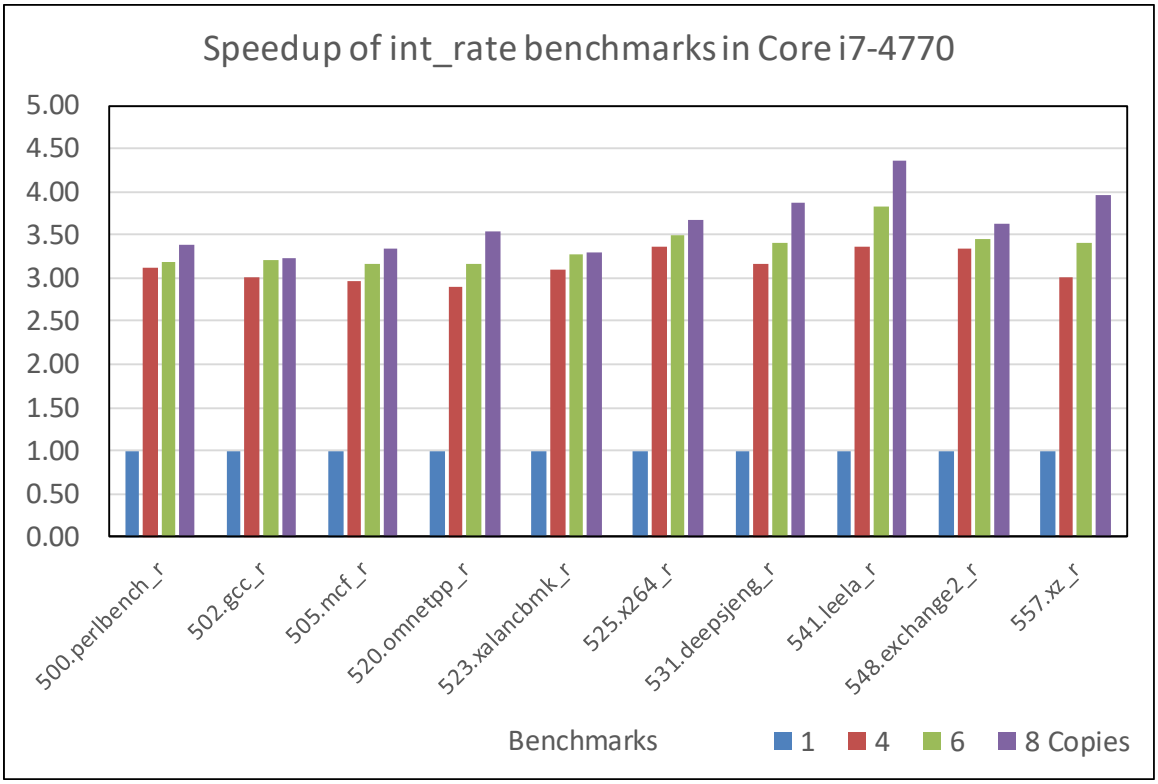

Figure 6.22 Speedup of *int_rate* Benchmarks in Core i7-4770

Table 6.21 Runtime and Speedup of *int_rate* in Core i7-8700K

clingmansdome	Runtime (sec) & Speedup				
Core i7-8700K	*int_rate*				
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies	12 Copies
500.perlbench_r	240.12	282.53	298.88	386.44	519.20
	1.00	3.40	4.82	4.97	5.55
502.gcc_r	179.75	233.40	290.42	362.55	572.71
	1.00	3.08	3.71	3.97	3.77
505.mcf_r	174.02	216.28	257.92	333.43	507.23
	1.00	3.22	4.05	4.18	4.12
520.omnetpp_r	333.66	474.15	596.07	770.67	1,114.49
	1.00	2.81	3.36	3.46	3.59
523.xalancbmk_r	207.29	271.09	316.22	394.43	570.77
	1.00	3.06	3.93	4.20	4.36
525.x264_r	110.25	118.64	122.79	158.48	205.54
	1.00	3.72	5.39	5.57	6.44
531.deepsjeng_r	186.20	203.20	212.90	276.44	336.71
	1.00	3.67	5.25	5.39	6.64
541.leela_r	333.02	355.64	366.80	446.36	543.96
	1.00	3.75	5.45	5.97	7.35
548.exchange2_r	195.88	209.96	216.26	285.32	387.30
	1.00	3.73	5.43	5.49	6.07
557.xz_r	257.74	320.13	361.68	454.92	562.34
	1.00	3.22	4.28	4.53	5.50

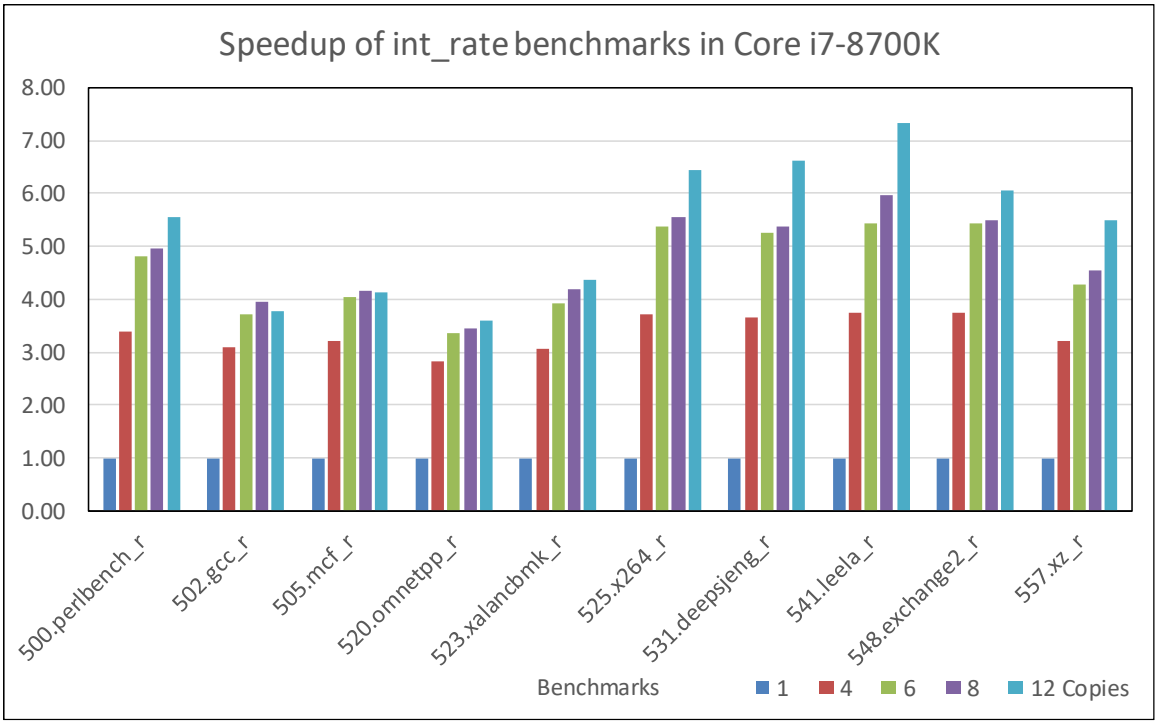

Figure 6.23 Speedup of *int_rate* Benchmarks in Core i7-8700K

Table 6.22 Runtime and Speedup of *int_rate* in Xeon E3-1240 V2

vidrak	Runtime (sec) & Speedup			
Xeon E3-1240 V2	*int_rate*			
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies
500.perlbench_r	444.23	507.01	734.96	863.22
	1.00	3.50	3.63	6.18
502.gcc_r	293.18	391.82	553.97	770.50
	1.00	2.99	3.18	4.57
505.mcf_r	252.08	369.22	535.49	706.08
	1.00	2.73	2.82	4.28
520.omnetpp_r	437.39	638.53	836.25	1,007.09
	1.00	2.74	3.14	5.21
523.xalancbmk_r	322.49	406.12	569.41	732.25
	1.00	3.18	3.40	5.29
525.x264_r	227.79	245.56	366.01	447.14
	1.00	3.71	3.73	6.11
531.deepsjeng_r	262.04	295.66	414.88	453.02
	1.00	3.55	3.79	6.94
541.leela_r	414.76	441.85	607.84	669.22
	1.00	3.75	4.09	7.44
548.exchange2_r	365.64	387.05	582.57	713.43
	1.00	3.78	3.77	6.15
557.xz_r	334.42	429.16	517.87	581.64
	1.00	3.12	3.87	6.90

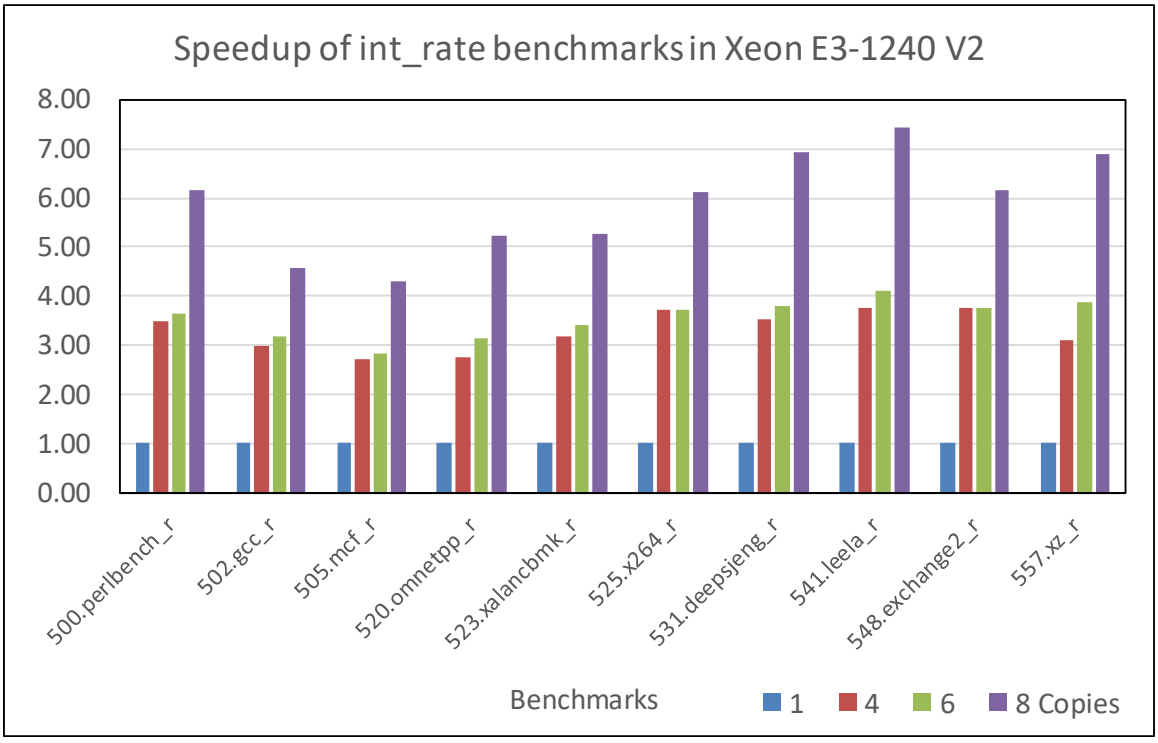

Figure 6.24 Speedup of *int_rate* Benchmarks in Xeon E3-1240 V2

Table 6.23 Runtime and Speedup of *int_rate* in Xeon E5-2643 V3

mtleconte	Runtime (sec) & Speedup						
Xeon E5-2643 V3	*int_rate*						
Benchmarks	1 Copy	4 Copies	6 Copies	8 Copies	12 Copies	16 Copies	24 Copies
500.perlbench_r	328.97	367.84	386.80	395.13	410.64	543.10	743.01
	1.00	3.58	5.10	6.66	9.61	9.69	10.63
502.gcc_r	267.30	295.04	315.76	339.64	400.25	519.94	811.07
	1.00	3.62	5.08	6.30	8.01	8.23	7.91
505.mcf_r	239.90	256.43	283.28	316.53	401.21	530.54	762.63
	1.00	3.74	5.08	6.06	7.18	7.24	7.55
520.omnetpp_r	435.74	511.02	542.09	578.41	676.04	907.40	1,245.20
	1.00	3.41	4.82	6.03	7.73	7.68	8.40
523.xalancbmk_r	315.59	336.51	365.24	389.87	450.76	592.92	910.37
	1.00	3.75	5.18	6.48	8.40	8.52	8.32
525.x264_r	160.18	166.14	168.46	172.35	173.33	225.22	295.33
	1.00	3.86	5.70	7.43	11.09	11.38	13.02
531.deepsjeng_r	240.86	245.02	262.50	266.03	274.44	380.10	445.32
	1.00	3.93	5.51	7.24	10.53	10.14	12.98
541.leela_r	416.59	422.10	433.34	431.74	441.48	544.79	670.33
	1.00	3.95	5.77	7.72	11.32	12.23	14.92
548.exchange2_r	259.32	266.25	270.77	273.46	280.39	413.07	501.53
	1.00	3.90	5.75	7.59	11.10	10.04	12.41
557.xz_r	319.02	346.69	403.33	410.68	437.40	557.22	670.22
	1.00	3.68	4.75	6.21	8.75	9.16	11.42

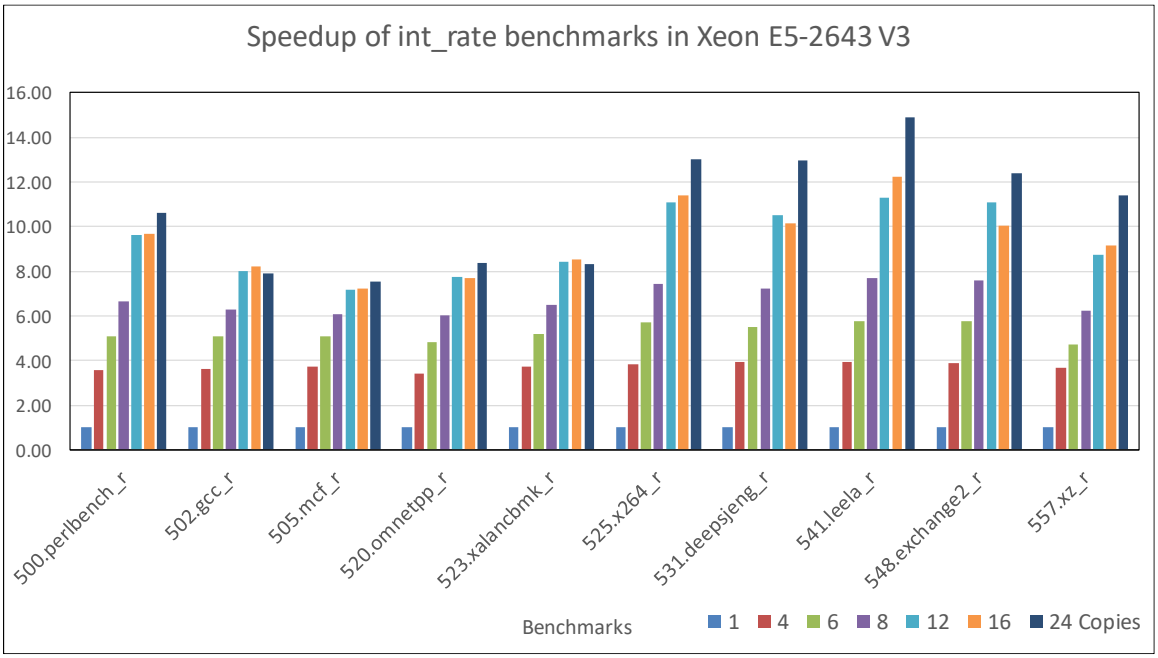

Figure 6.25 Speedup of *int_rate* Benchmarks in Xeon E5-2643 V3

Table 6.24 shows the *SPECrate2017_int_base* number across all the test systems. The Core processors outperform the Xeon processors on single copy benchmarks here also partly because of the higher operating clock frequencies. As the benchmarks scale well, the best performance is observed when the number of copies matches the number of physical cores. Figure 6.26 gives a visual representation of the *SPECrate2017_int_base* numbers for a different number of copies on all the test systems.

Table 6.24 *SPECrate2017_int_base* on all Test Systems

int_rate				
	mtsano	*clingmansdome*	*vidrak*	*mtleconte*
Thread_Count	Core i7-4770	Core i7-8700K	Xeon E3-1240 V2	Xeon E5-2643 V3
1	5.38	6.95	4.49	5.11
4	16.80	23.29	14.76	19.12
6	18.02	31.31	15.83	26.92
8	19.44	32.73	17.44	34.50
12		36.07		47.36
16				47.64
24				53.57

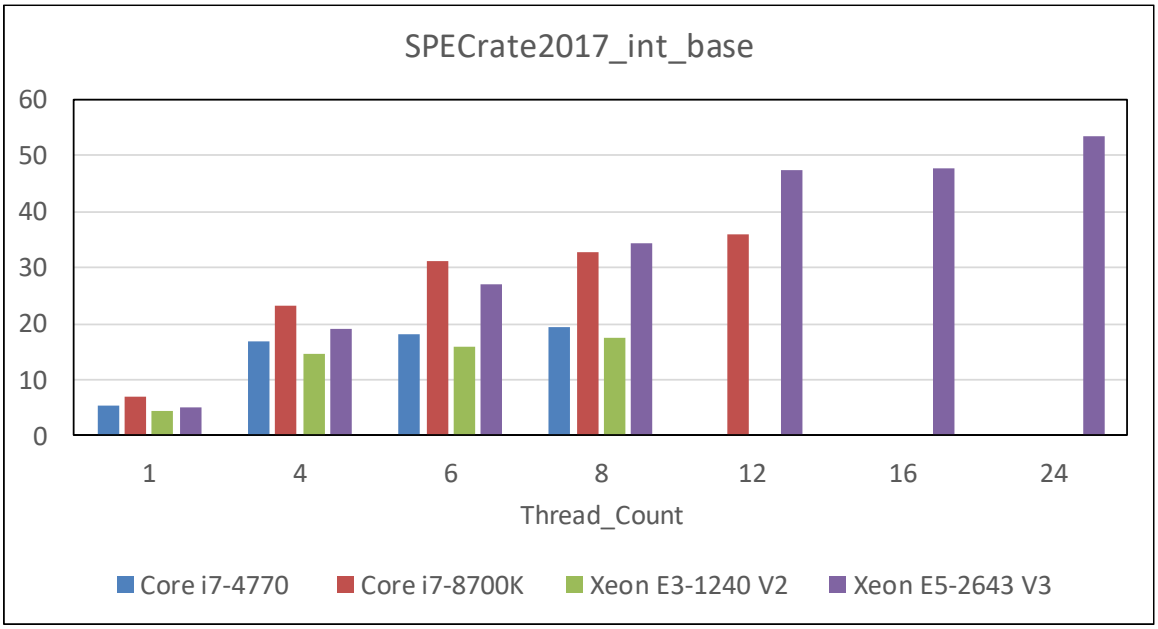

Figure 6.26 *SPECrate2017_int_base* Results on all Test Systems

A performance comparison across machines is made in Table 6.25. The speedups here are defined in Eq.6.3 and Eq.6.4. While comparing the two core i7's (*mtsano and clingmansdome),* the scaled speedup of *clingmansdome* over *mtsano* is minimal at ~3%. However, the speedup increases to ~25%, with an increase in the number of copies because of the higher core count in *clingmansdome*. Single core performance improvement of 18% is seen for *mtleconte* over *vidrak* and the speedup increases substantially as *mtleconte* has a second processor. Cross-comparison of machines in the case of *vidrak* and *mtleconte,* the newer architecture in *mtsano* contributed to ~25% gains. In the case of the two hexa-core machines, *clingmansdome and mtleconte,* the newer i7 fails to keep up when copy count increases. For throughput a multi-processor system would fare better especially when the scaled speedup for single copy application was relatively the same. Figure 6.27 and Figure 6.28 show excerpts from auto-generated reports from SPEC CPU2017 runs of the *int_rate* benchmarks on *clingmansdome* for one and six copies, respectively.

Table 6.25 Relative Speedups for *int_rate* on Different Test Machines

int_rate								
	clingmansdome to mtsano		*mtsano to vidrak*		*mtleconte to vidrak*		*clingmansdome to mtleconte*	
Copies	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup	Speedup	Scaled Speedup
1	1.29	1.02	1.20	1.23	1.14	1.18	1.36	1.06
4	1.39	1.04	1.14	1.24	1.30	1.31	1.22	0.99
6	1.74	1.32	1.14	1.25	1.70	1.74	1.16	0.95
8	1.68	1.25	1.11	1.25	1.98	2.03	0.95	0.77
12							0.76	0.63

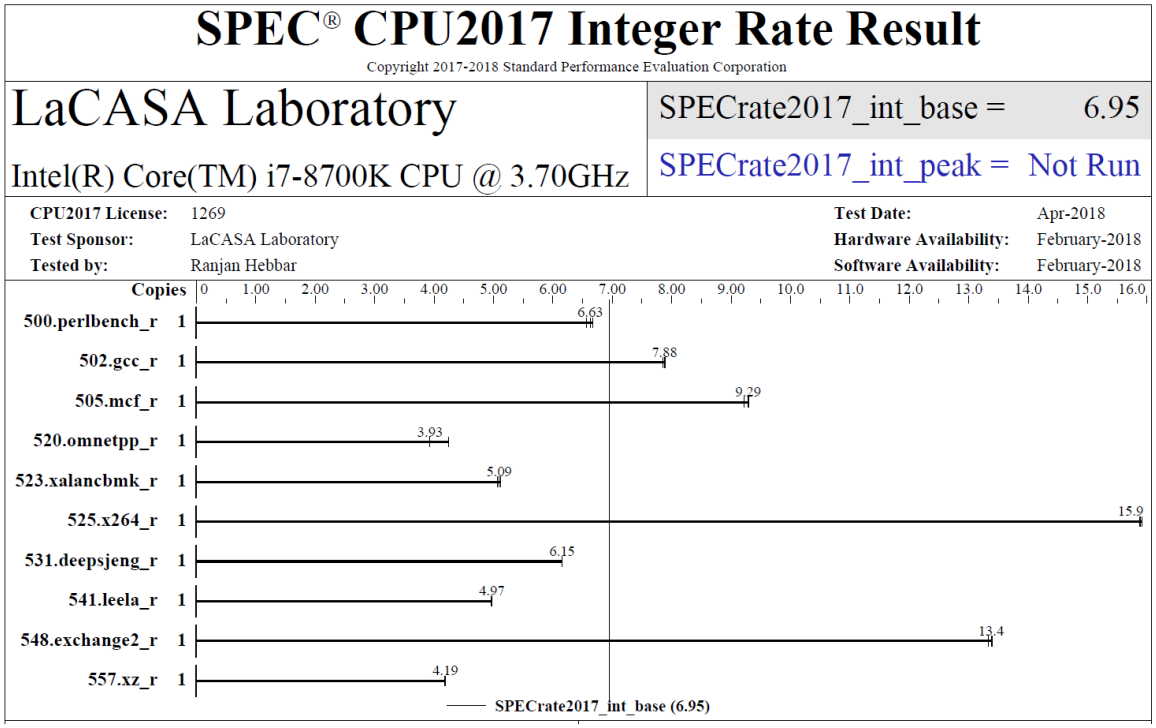

Figure 6.27 SPEC CPU2017 Report Excerpt for Single-Copy *int_rate*

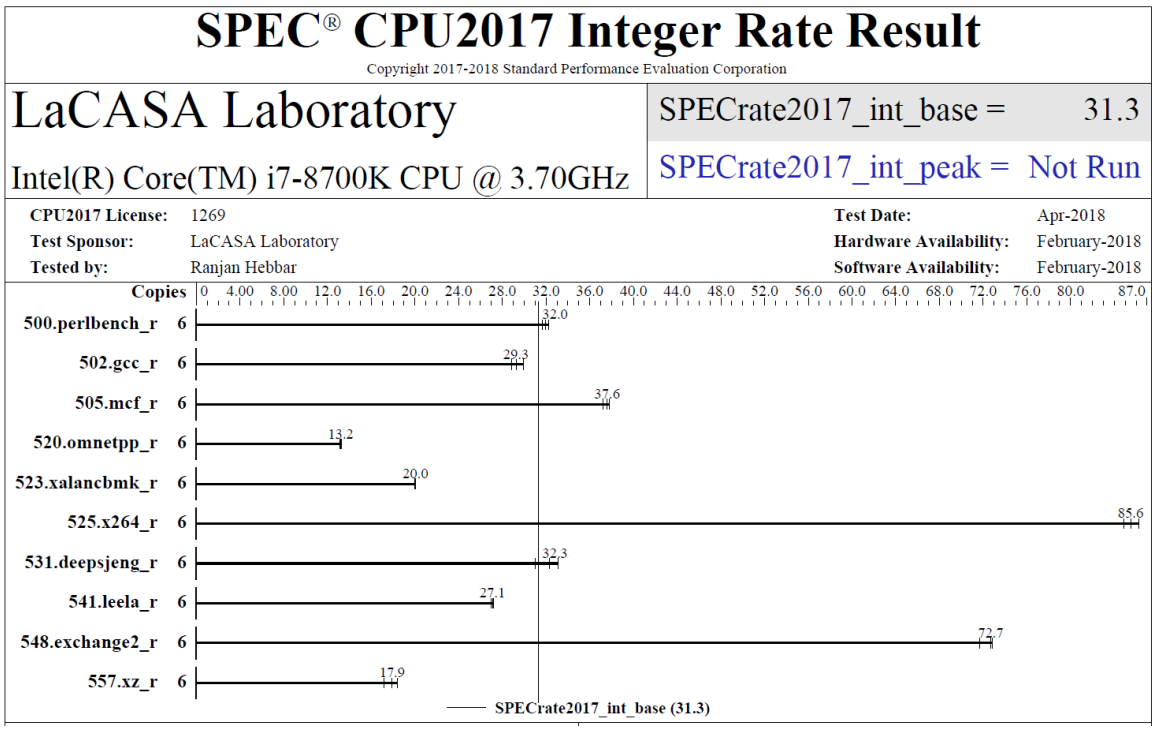

Figure 6.28 SPEC CPU2017 Report Excerpt for Six-Copy *int_rate*

6.2 Impact of Hardware Prefetching

This section discusses the performance impact of hardware prefetching on modern processors using the SPEC CPU2017 benchmarks. The SPEC benchmarks are executed on the Core i7-8700K used in *clingmansdome*, first with hardware prefetching disabled and then with hardware prefetching enabled. The experiments are repeated while varying the number of threads. The speedup, calculated as the ratio of the execution times when a given benchmark is run without and with hardware prefetching, is used to quantify the performance impact.

Figure 6.29 shows the speedup for single-threaded speed benchmarks. The speedups for individual benchmarks range from 1.08 (638.imagick_s) to 2.26 (603.bwaves_s) for *fp_speed* and from 0.96 (623.xalancbmk_s) to 3.34 (602.gcc_s) for *int_speed*. These results suggest that hardware prefetching has a significant impact on performance for both the *fp_speed* and *int_speed* benchmarks, though *fp_speed* benchmarks benefit a bit more on average. In some applications, hardware prefetching improves performance by over 3 times (602.gcc_s). Interestingly, hardware prefetching degrades the performance of 623.xalancbmk_s – a benchmark that converts text documents using tree-based data structures.

Figure 6.30 shows the speedup when the number of threads N=6. The advantages of enabling prefetching are less prevalent when the thread count increases. The speedups range from 0.93 (654.roms_s) to 1.30 (644.nab_s) for *fp_speed* and from 0.96 (632.xalanbmk_s) to 3.36 (602.gcc_s) for *int_speed*.

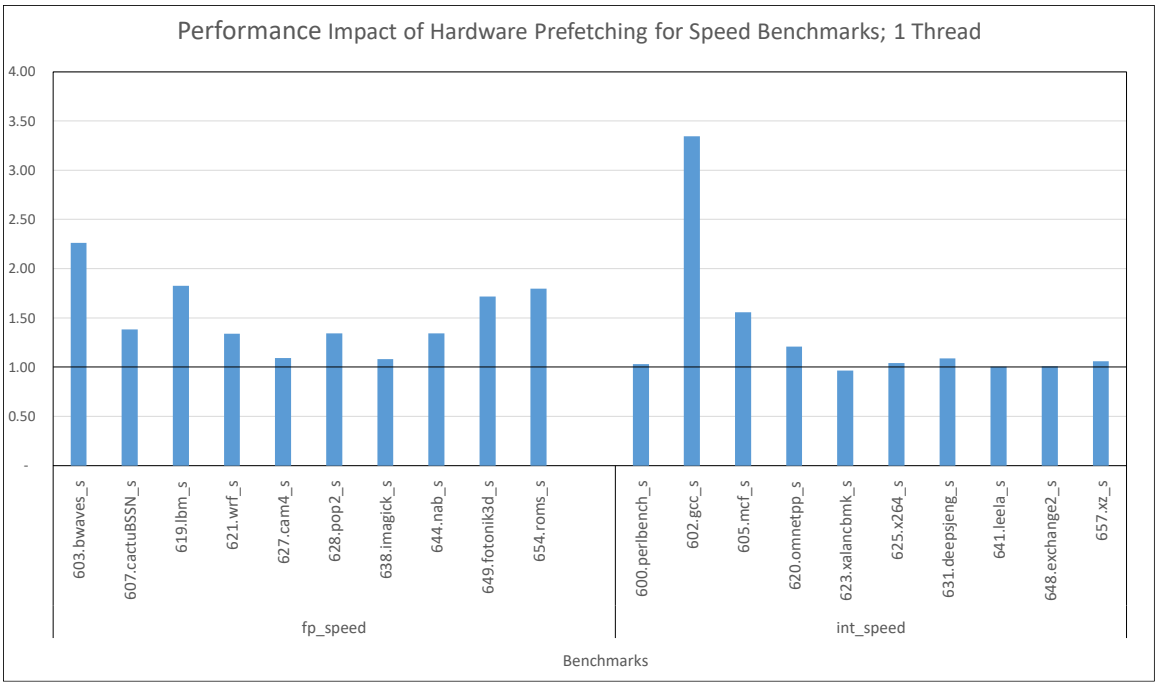

Figure 6.29 Impact of Hardware Prefetching on 1-Thread Speed Benchmarks

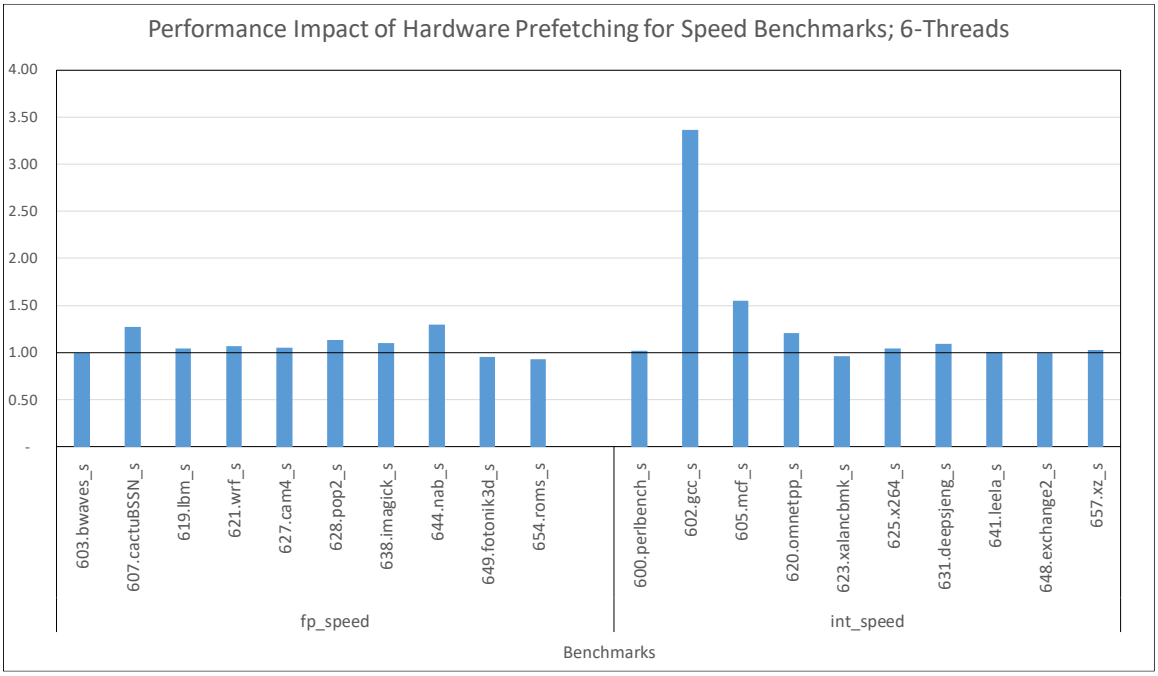

Figure 6.30 Impact of Hardware Prefetching on 6-Thread Speed Benchmarks

Figure 6.31 shows the speedup for the single-copy SPEC rate benchmarks. The speedups of individual benchmarks range from 0.98 (511.povray_r) to 2.37 (503.bwaves_r) for *fp_rate* and from 0.97 (523.xalancbmk_r) to 1.60 (502.gcc_r) for *int_rate*. The individual speedups are smaller than the equivalent ones observed in speed benchmarks because the rate benchmarks use smaller size data inputs.

Figure 6.32 shows the speedups for the SPEC rate benchmarks executed with six copies. The advantages of enabling prefetching do not seem to affect multiple copy runs. Some benchmarks suffered performance degradations from enabling prefetching (554.roms_r). The speedups range from 0.78 (554.roms_r) to 1.29 (544.nab_r) for *fp_rate* and from 0.91 (520.omnetpp_r) to 1.41 (502.gcc_r) for *int_rate*.

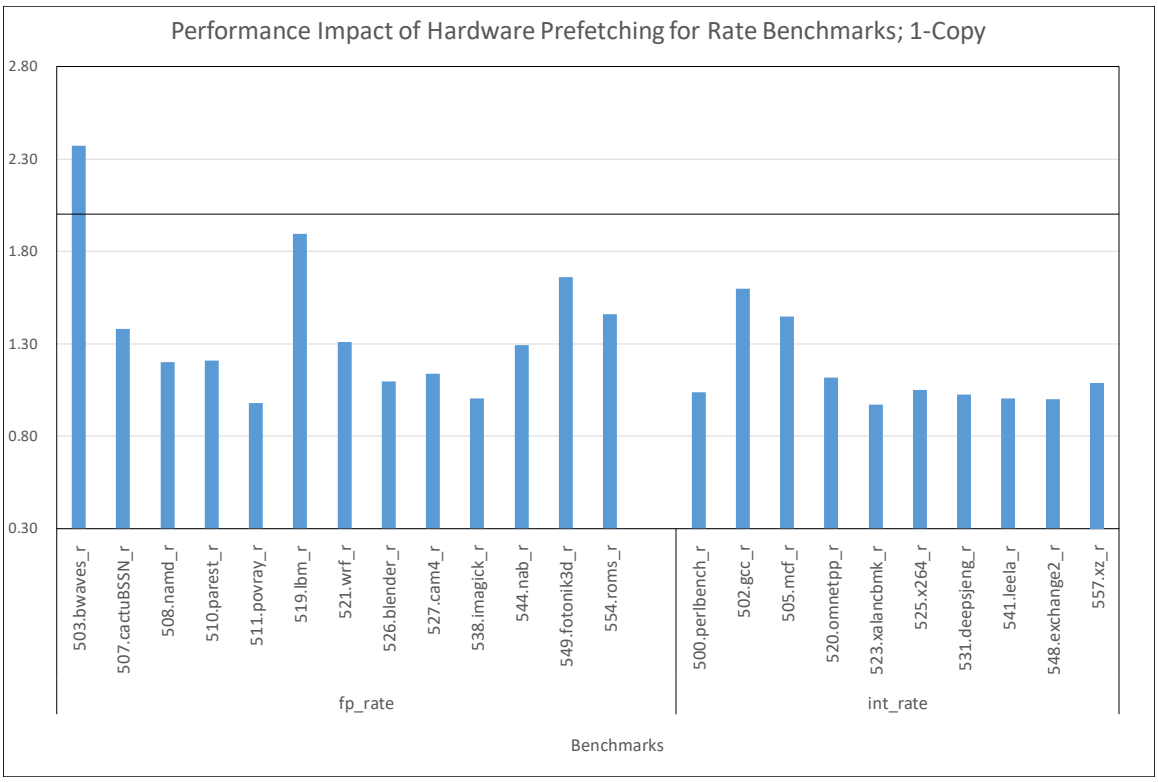

Figure 6.31 Impact of Hardware Prefetching on 1-Copy Rate Benchmarks

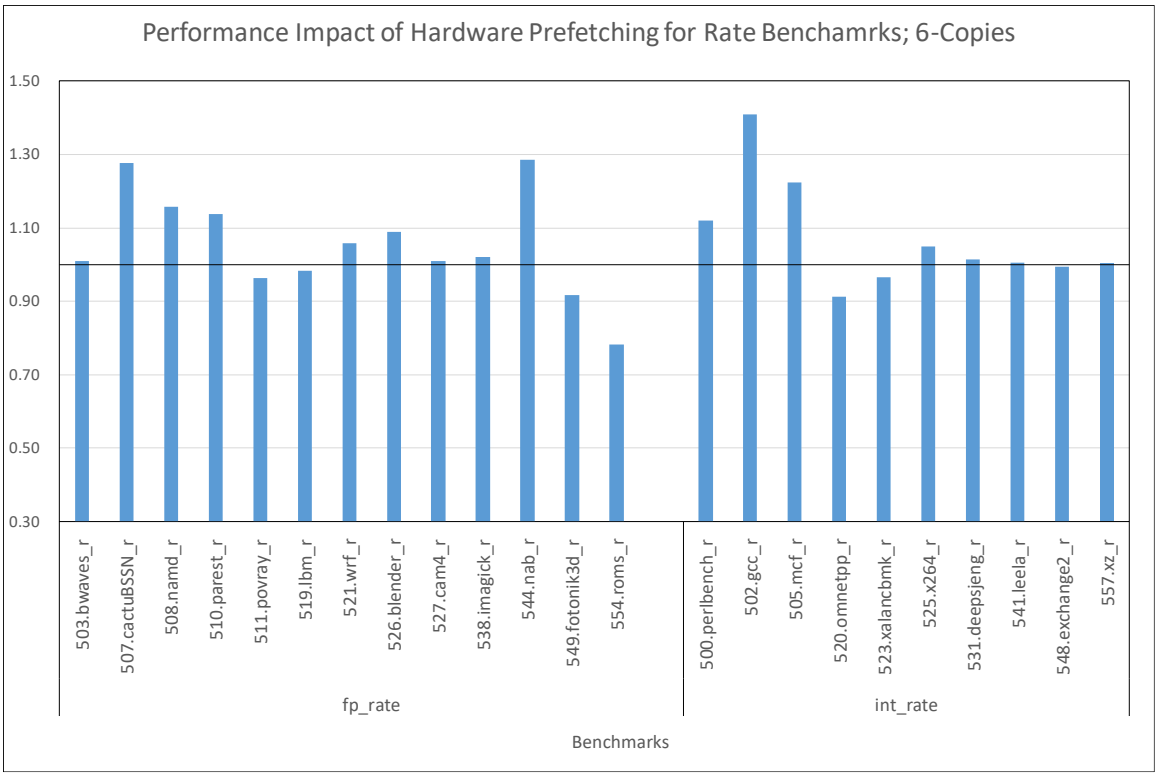

Figure 6.32 Impact of Hardware Prefetching on 6-Copy Rate Benchmarks

Table 6.26 shows the *SPEC_ratio_base* metric obtained for SPEC runs when prefetching is enabled and disabled, respectively. Figure 6.33 gives a graphical presentation of the results. Overall, hardware prefetching improves performance for most of the benchmarks. Single-threaded benchmarks seem to have the most benefit as thread communication and cache evictions is not a factor. When the system is at full load, prefetching could actually hurt performance by removing useful data from caches. For multiple copies, prefetching could evict data from the shared cache that could be used by other applications hurting the overall performance even if for the same application prefetching improved performance for single copy execution.

Table 6.26 *SPEC2017_ratio_base* for Prefetching Evaluation

	1 Thread/Copy		6 Threads/Copies	
Suite	Without Prefetching	With Prefetching	Without Prefetching	With Prefetching
fp_speed	7.31	10.81	22.69	24.4
int_speed	6.58	8.07	7.40	9.01
fp_rate	8.82	11.82	31.10	32.49
int_rate	6.22	6.95	29.49	31.31

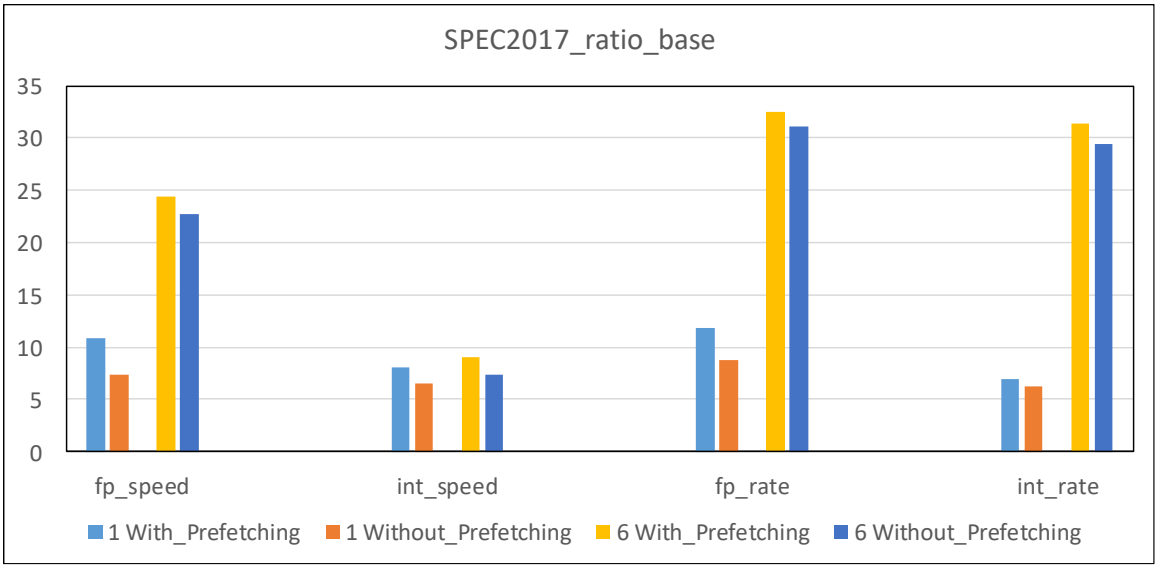

Figure 6.33 *SPEC2017_ratio_base* Numbers for Prefetching Evaluation

CHAPTER 7

CONCLUSIONS

Standardized benchmark suites, designed to provide a fair and structured performance evaluation of modern computer systems, have been widely used in both academia and industry. SPEC CPU2017 is the most recent incarnation of benchmarks that focus on the evaluation of processor performance. It includes a number of benchmarks designed to evaluate processor speed and throughput for both integer and floating-point benchmarks. These benchmarks are significantly more complex than those used in previous instances of SPEC CPU benchmarks. In addition, for the first time, SPEC CPU suites include a number of benchmarks that are parallelizable and thus can harness the performance of modern multi-core processors.

This thesis focuses on performance studies on modern processors using SPEC CPU2017. It describes a number of measurement-based studies that analyze characteristics of these benchmarks when executed on a series of modern Intel Core i7 and Xeon processors. These measurements are performed using SPEC CPU2017 run utilities and Linux utilities to interface processors' performance monitoring units, such as *perf* and *likwid*. The Intel's Top-down Microarchitecture Analysis Method is performed using *Intel VTune Amplifier*. The results obtained from these studies can be broadly grouped into following categories:

(a) Characterizing benchmarks by providing a top-view that includes SPEC metrics, execution time, the power consumed, and the average clock frequency.

(b) Analyzing scalability of parallelizable speed benchmarks as a function of the number of threads.

(c) Analyzing scalability of rate benchmarks as a function of the number of cores.

(d) Comparing the performance of machines featuring different processors.

(e) Performing Intel's Top-down Microarchitecture Analysis Method.

(f) Analyzing performance impact of hardware prefetching.

With numerous parameters affecting performance, the Intel Top-down Micro-architectural Analysis Method shows that the *fp_speed* benchmarks are bound by memory hierarchy, whereas the *int_speed* benchmarks are bound by bad speculation and front-end stalls. In the case of the *fp_rate* benchmarks, stresses on the memory hierarchy with an increase in the number of copies are even more emphasized.

A new metric called *PerfEE Speedup* is introduced that helps in finding the best combination of benchmarks to run to achieve optimal performance and energy consumption. Using this metric as a criterion, it is found that the *fp_speed* benchmarks with thread count equaling the number of physical cores achieve an optimum performance and energy.

Comparative analysis of performance shows that architectural improvements in newer Intel processors Core i7 processors (8^{th} generation vs. 4^{th} generation) can be attributed to ~20% of performance gains when the clock frequency is normalized. Xeon processors achieve significantly better throughput for the rate benchmarks.

Future research can branch into several directions. Whereas this research has been conducted on different machines, they all used the same operating system and Intel's compiler. One interesting question is to evaluate the effectiveness of different

compilers and within the same compiler different optimization levels and their impact on performance. Another direction is to expand the use of Intel's Top-view Microarchitecture Analysis Method on different processor models. Next, SPEC CPU2017 throughput analysis combines multiple copies of the same benchmark. It would be interesting to explore the impact on performance when combining different benchmarks, perhaps integer and floating-point benchmarks together.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5 edition. San Francisco, CA: Morgan Kaufmann, 2011.

[2] "SPEC - Standard Performance Evaluation Corporation." [Online]. Available: https://www.spec.org/. [Accessed: 19-Mar-2018].

[3] "SPEC CPU® 2006." [Online]. Available: https://www.spec.org/cpu2006/. [Accessed: 19-Mar-2018].

[4] "SPEC CPU® 2017." [Online]. Available: https://www.spec.org/cpu2017/. [Accessed: 19-Mar-2018].

[5] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 271–282.

[6] "Intel® 64 and IA-32 Architecture's Optimization Reference Manual," p. 672, Jun. 2016.

[7] "Intel Tick-Tock Model," *Intel*. [Online]. Available: https://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html. [Accessed: 12-Apr-2018].

[8] S. Damaraju *et al.*, "A 22nm IA multi-CPU and GPU System-on-Chip," in *2012 IEEE International Solid-State Circuits Conference*, 2012, pp. 56–57.

[9] B. Heaney, "DAC 2012 Keynote: Designing a 22nm Intel® Architecture Multi-CPU and GPU," p. 26, 2012.

[10] M. Milenkovic, A. Milenkovic, and J. Kulick, "Microbenchmarks for determining branch predictor organization," *Software: Practice and Experience*, vol. 34, no. 5, pp. 465–487, Apr. 2004.

[11] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," *Software Practice & Experience*, vol. 34, no. 4, pp. 465–487, Apr. 2004.

[12] "White Paper- Introduction to Intel Architecture (Haswell)." Intel.

[13] "Intel® Turbo Boost Technology 2.0," *Intel*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. [Accessed: 02-Jun-2018].

[14] S. T. Gurumani and A. Milenkovic, "Execution Characteristics of SPEC CPU2000 Benchmarks: Intel C++ vs. Microsoft VC++," in *42nd ACM Southeast Conference*, 2004, p. 6.

[15] "Intel® Core™ i7-4770 Processor (8M Cache, up to 3.90 GHz) Product Specifications," *Intel® ARK (Product Specs)*. [Online]. Available: https://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz. [Accessed: 19-Mar-2018].

[16] "Intel® Core™ i7-8700K Processor (12M Cache, up to 4.70 GHz) Product Specifications," *Intel® ARK (Product Specs)*. [Online]. Available: https://ark.intel.com/products/126684/Intel-Core-i7-8700K-Processor-12M-Cache-up-to-4_70-GHz. [Accessed: 24-Mar-2018].

[17] "Intel® Xeon® Processor E3-1240 v2 (8M Cache, 3.40 GHz) Product Specifications," *Intel® ARK (Product Specs)*. [Online]. Available: https://ark.intel.com/products/65730/Intel-Xeon-Processor-E3-1240-v2-8M-Cache-3_40-GHz. [Accessed: 19-Mar-2018].

[18] "Intel® Xeon® Processor E5-2643 v3 (20M Cache, 3.40 GHz) Product Specifications," *Intel® ARK (Product Specs)*. [Online]. Available: https://ark.intel.com/products/81900/Intel-Xeon-Processor-E5-2643-v3-20M-Cache-3_40-GHz. [Accessed: 19-Mar-2018].

[19] "Perf : Linux profiling with performance counters," *Perf Wiki*. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed: 19-Mar-2018].

[20] "Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B," *Intel*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html. [Accessed: 19-Mar-2018].

[21] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," *Competence in High Performance Computing 2010*, Apr. 2010.

[22] "Intel® VTune™ Amplifier 2018 User's Guide," *Intel Developer Zone*. [Online]. Available: https://software.intel.com/en-us/vtune-amplifier-help-introduction. [Accessed: 28-Mar-2018].

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

[24] "Thread Affinity Interface." [Online]. Available: https://software.intel.com/en-us/node/684320. [Accessed: 19-Mar-2018].

[25] A. Dzhagaryan and A. Milenković, "Impact of thread and frequency scaling on performance and energy in modern multicores: a measurement-based study," in *52nd Annual ACM Southeast Conference (ACMSE'14)*, Kennesaw, GA, 2014, pp. 1–6.

[26] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley, "Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling ∗," *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pp. 319–332, Mar. 2011.