PMU-EVENTS-DRIVEN DVFS TECHNIQUES FOR IMPROVING ENERGY EFFICIENCY IN MODERN PROCESSORS

by

RANJAN HEBBAR

A DISSERTATION

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

HUNTSVILLE, ALABAMA

2021

In presenting this dissertation in partial fulfillment of the requirements for a doctoral 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 dissertation.

________________________________ ____________________
(student signature) (date)

DISSERTATION APPROVAL FORM

Submitted by Ranjan Hebbar in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation 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 dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering in Computer Engineering.

Committee Chair

(Dr. Aleksandar Milenkovic) (date)

(Dr. B. Earl Wells) (date)

(Dr. David Coe) (date)

(Dr. Jeffrey H. Kulick) (date)

(Dr. Mohammad Haider) (date)

Department Chair

(Dr. Ravi Gorur) (date)

College Dean

(Dr. Shankar Mahalingam) (date)

Graduate Dean

(Dr. Sean Lane) (date)

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree: Doctor of Philosophy in Engineering
College/Dept.: Engineering/Electrical & Computer Engineering
Name of Candidate: Ranjan Hebbar
Title: PMU-Events-Driven DVFS Techniques for Improving Energy Efficiency in Modern Processors

Energy-efficient computing is one of the most important challenges computer designers and operators are facing today, exacerbated by the ever-increasing demands for faster, smaller, lighter, and more affordable computing. The processor is the primary driver of the overall system power consumption of a computer system. Typical power management techniques rely on either running the processor at a fixed clock frequency or utilizing dynamic voltage and frequency scaling (DVFS) techniques that adjust the processor's clock frequency in runtime based on its current level of activity.

In this dissertation, we first describe the results of our measurement-based study that evaluates the impact of the state-of-the-art power management techniques on performance (P), energy efficiency (EE), and their product (PxEE) in an Intel Core i7 processor, running SPEC CPU2017, Parsec-3.0, and SPECpower_ssj2008 benchmark suites. The results of this study indicate that the state-of-the-art DVFS power management techniques heavily favor performance, resulting in poor energy efficiency. For example, we find that the processor operates at the highest clock frequency even when 90% of all processor cycles are stalls, resulting in wasted energy. To remedy this problem, we introduce, implement, and evaluate the effectiveness of four new DVFS-based power management techniques driven by the following metrics derived from the processor's performance monitoring unit (PMU): (i) the percentage of all pipeline slot stalls (*FS-PS*), (ii) the percentage of all cycle stalls (*FS-TS*), (iii) the percentage of memory-related cycle stalls (*FS-MS*), and (iv) the number of last level cache misses per kilo instructions (*FS-LLCM*), respectively. The proposed techniques linearly map these metrics into available processor clock frequencies.

The results of the experimental evaluation show that the proposed techniques significantly improve EE and PxEE metrics relative to the state-of-the-art approaches. Further, we find that the proposed techniques are especially effective for memory-intensive benchmarks, wherein EE improves from 121% to 183% and PxEE from 100% to 141%. We elucidate the advantages and disadvantages of each of the proposed techniques and offer guidelines on when to use them.

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

ACKNOWLEDGMENTS

The work presented in this dissertation 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 support at every stage of this work and for creating an inspirational work environment in the LaCASA laboratory. He inspired me personally and professionally with his patience and his interest towards learning.

I will be always grateful to Dr. Ravi Gorur, Chair of the Electrical and Computer Engineering Department, for encouraging me to pursue research, and for providing me with financial support through the teaching assistantship during the course of my study.

I would also like to thank Dr. Coe, Dr. Haider, Dr. Kulick, and Dr. Wells for agreeing to serve on my committee and providing valuable feedback.

I would like to express my appreciation to Dr. Mounika Ponugoti, Dr. Prawar Poudel, Dr. Armen Dzhagaryan, Mr. Igor Semenov, and Mr. Amir Ramezani for their constant support in the LaCASA laboratory.

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

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Modern computing is continually evolving shaped by constant advances and changes in technology, applications, and market trends. Over the past six decades, semiconductor technology nodes have gotten smaller and more refined, resulting in an exponential increase in the number of transistors on a single die. Fueled by this phenomenal growth, 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. 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. Still, processors that power the contemporary laptop, desktop, and server computers remain one of the most important components in computing ecosystems.

Historically, improvements in the energy efficiency of modern processors were predominantly a byproduct of Moore's law. Shrinking technology nodes give smaller and faster transistors, resulting in more energy-efficient computing. However, recent trends indicate an end to Moore's Law. This is concerning as the energy consumption of data centers worldwide was estimated to be ~263 TWh in 2020 [4] and it is expected to grow to 1,137 TWh by 2030. This calls for renewed efforts in improving the energy efficiency of modern computers, especially those used in the largest cloud data centers.

A majority of high-end workstations and servers use x86 processors from Intel and AMD. Modern x86 processors have evolved to become extremely complex hardware structures, integrating multiple processor cores, multi-level cache structures, memory controllers that support multiple channels, a slew of hardware accelerators, and an interconnect network that connects all of these components on a single chip. Each processor core is highly pipelined with a superscalar out-of-order execution engine with speculative instruction execution, simultaneous multi-threading (SMT), hardware prefetching, advanced vectorization, and various other performance-enhancing structures. Consequently, computer architects have included hardware resources dedicated to monitoring and managing the operating states of the processor to ensure its safe, reliable, and efficient operation [19].

Dynamic Voltage and Frequency Scaling (DVFS) is a technique used in modern processors to adjust the clock frequency and the power supply voltage of specific modules based on their level of activity, thus reducing the power consumption and the heat generated by the processor. Each new generation of processors, starting from Intel's Haswell/Broadwell architecture [22] [31], has added more sophisticated hardware resources that support faster and more efficient DVFS techniques [52] [54]. Thus, modern processors support several performance states (a.k.a. P-states) that leverage DVFS and power states (a.k.a. C-states) that allow for unused modules to be turned off [68].

Algorithms for controlling the P- and C-states are carried out by either BIOS firmware or an OS driver, as defined in the Advanced Configuration and Power

Interface (ACPI) standard [69]. The control algorithms (in the further text referred to as governors) determine how the current processor state is monitored, what conditions warrant changes to the processor state, how the new state is determined, and how frequently these actions take place. Governors broadly fall into two categories: those that employ specific operating states (e.g., *performance*) or those that observe the current processor load and dynamically react to its changes (e.g., *ondemand/powersave*). The Linux recommended *ondemand* governor monitors the utilization of individual processor cores and uses it as the only factor in determining the cores' operating states [58] [70]. The governors send out requests to a dedicated unit on the processor called the Power Control Unit (PCU or P-Unit) to change the operating states of individual processor cores and other components at regular time intervals. This implementation is common across hardware and software vendors.

The state-of-the-art *ondemand* governor provides performance similar to the *performance* governor with lower power consumption during idle times. The current consensus reflected in the implementation of common governors is that running a processor at the highest possible clock frequency during program execution is the most energy-efficient strategy. However, several recent studies have shown that this approach is not optimal for all types of workloads, especially for those that are bounded by memory [15] [27].

Finding an efficient method to select an optimal operating frequency during a program's run-time remains a challenging problem. A number of prior research efforts have proposed analytical models [50] [57] and experimental methods [37] [71] to inform the design and implementation of energy-efficient governors. However, these proposals have not seen widespread adoption due to the added complexity, processing latency, and relatively modest gains.

1.1 Scope of This Study

This dissertation primarily focuses on the DVFS power management techniques in modern x86 processors to improve energy efficiency. First, we evaluate the effectiveness of the state-of-the-art OS governor (*ondemand)* by measuring its impact on performance (P), energy efficiency (EE), and their product (PxEE). The experimental evaluation is primarily carried out on a workstation with an Intel Core i7-8700K processor. To represent modern real-life workloads, we use the SPEC CPU2017 benchmark suites. We find that the *ondemand* governor tends to put the processor cores at the highest possible clock frequency, regardless of the properties of benchmarks being executed. While this policy maximizes performance for all types of benchmarks, it results in a significant amount of wasted energy, especially in the case of benchmarks bounded by the memory subsystem.

To address this problem, we propose, implement, and evaluate four new techniques that determine the P-state of the processor core using the following metrics derived from performance monitoring unit (PMU) events: (i) the total number of pipeline slot stalls (FS-PS), (ii) the total number of cycle stalls (FS-TS), (iii) the total number of memory-related cycle stalls (FS-MS), and (iv) the number of last level cache misses per kilo instructions (FS-LLCM). Each technique linearly maps the corresponding metric to the available P-states on a system. We also investigate the previous DVFS proposal that utilizes the cycles-per-instruction metric when determining the next P-state (FS-CPI).

The measurement-based studies performed in the dissertation rely on architectural support provided by the on-chip performance monitoring unit (PMU) that are part of modern processors' fabric. Initially, tools such as Linux utility *perf*

[72] and *Intel's VTune Amplifier* [73] are leveraged to profile the SPEC CPU2017 suites to better understand the impact of DVFS. Further, we utilize *likwid* [60] to measure the execution time and energy consumed by the processor for each of the benchmarks. We evaluate our proposed techniques by comparing them to the state-of-the-art *ondemand* governor, with metrics such as performance speedup (P.S), energy efficiency improvement (EE.I), the improvement in the product of performance and energy efficiency (PxEE.I).

To further validate the proposed techniques and in an effort to add additional diversity to our workloads, we use two more representative benchmark suits. First, a set of parallel benchmarks from Parsec-3.0 is used representing a somewhat lighter version of compute-intensive applications. Next, to represent server workloads, we use the SPECpower_ssj2008 benchmark suites and evaluate the techniques of interest by using the performance per watt metric on the test system.

1.2 Contributions

The main contributions of this dissertation are as follows.

- It quantitatively evaluates the effectiveness of the state-of-the-art power management technique in modern processors (the *ondemand* governor) and determines its shortcomings, especially in terms of its energy efficiency.
- It provides an in-depth analysis of the SPEC CPU2017 benchmarks using the Top-down Microarchitectural Analysis Method and classifies the benchmarks into three groups based on their characteristics.
- It introduces and implements four PMU-event-driven DVFS techniques that promise to provide significant energy-efficiency improvements.

- It experimentally evaluates the effectiveness of the proposed techniques and the existing state-of-the-art by considering performance, energy, efficiency, and the product of performance and energy efficiency. The experimental evaluation involves three different types of workloads, namely SPEC CPU2017, Parsec 3.0, and SPECpower_ssj2008.
- It provides insights into the inner workings of the proposed DVFS-based techniques and discusses their pros and cons relative to each other and the previously proposed FS-CPI technique.

1.3 Findings

The main finding of this dissertation is summarized as follows.

- The state-of-the-art governors provide the best possible performance albeit at the cost of poor energy efficiency. This is especially true for memory-bound benchmarks.
- The results of our experimental evaluation show that all of the proposed techniques provide significant improvements to EE and PxEE metrics when compared to the state-of-the-art ondemand governor, especially for the class of memory-intensive benchmarks. Considering all the SPEC CPU2017 benchmarks, the proposed techniques improve EE from 44% (FS-LLCM) to 92% (FS-PS), whereas PxEE improves from 31% (FS-LLCM) to 48% (FS-PS). The proposed techniques are especially effective for a class of memory-intensive SPEC CPU2017 benchmarks - EE improves from 121% (FS-MS) to 183% (FS-PS) and PxEE from 100% (FS-MS) to 141% (FS-PS).
- The proposed techniques also outperform the previously proposed FS-CPI. Relative to FS-CPI, the proposed techniques improve EE from 2% (FS-

LLCM) to 36% (FS-PS) when all benchmarks are considered together, and from 20% (FS-MS) to 54% (FS-PS) when memory-intensive benchmarks are considered alone.

- Considering Parsec-3.0 benchmark suits, the proposed techniques improve EE from 15% (FS-LLCM) to 58% (FS-PS) and PxEE from 5% (FS-PS) to 18% (FS-MS).
- In the case of SPECpower_ssj2008, Linux recommended *'OS-ondemand' to* provide the lowest performance-per-watt for a fully loaded system. All of the proposed techniques improve performance-per-watt as follows: FS-PS by 61%, FS-TS by 72%, FS-MS by 61%, and FS-LLC-MPKI by 24%.

1.4 Outline

The rest of the dissertation is organized as follows. CHAPTER 2 provides an overview of the current power management infrastructure in modern x86 processors. CHAPTER 3 explains the shortcomings of the state-of-the-art implementation and provides motivation for this study. CHAPTER 4 describes the proposed PMU-event-driven DVFS techniques aimed at increasing energy efficiency. CHAPTER 5 details the experimental setup, the tools employed, and the evaluation metrics used for the study. CHAPTER 6 provides an in-depth analysis of the primary workload used in the study. The SPEC CPU2017 benchmarks are classified into three distinct groups using the Top-down Microarchitectural Analysis Method. CHAPTER 7 provides the experimental results for all the proposed techniques. CHAPTER 8 discusses the related work in the field of power management through dynamic voltage and frequency scaling. CHAPTER 9 describes the various avenues for future work. Finally, CHAPTER 10 concludes the dissertation.

CHAPTER 2

BACKGROUND

Modern multicore processors have evolved to be extremely complex hardware structures that continue to advance by integrating an ever-increasing number of functional units aimed at achieving high performance. With billions of transistors on a single chip that can run at clock frequencies approaching 5 GHz, power consumption and thermal management have emerged as one of the most important design constraints. To address growing concerns related to thermal and power aspects in modern processors, manufacturers have incorporated hardware resources solely dedicated to power management.

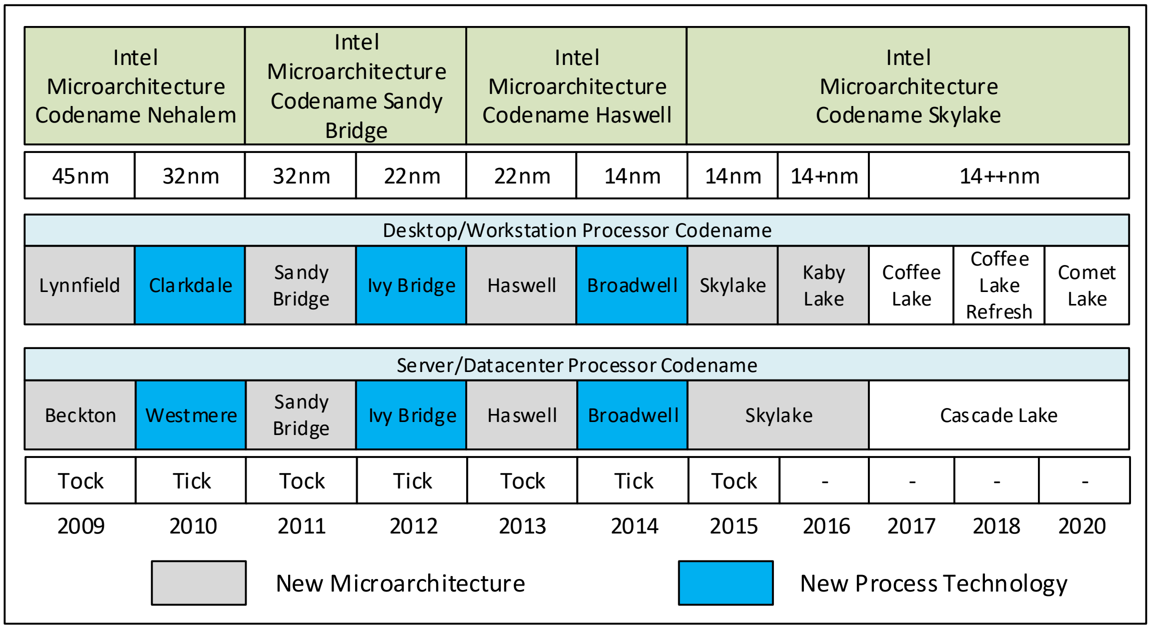
In the last 20 years, the complexity and sophistication of these resources have significantly increased, following an increase in the complexity of processors. Modern processors integrate multiple functional blocks on a single chip, including processor cores, interconnect, hardware accelerators (e.g., general-purpose graphics processing unit), a memory controller, and others. These functional blocks may be selectively turned on or off, or when active their operating points may be adjusted independently from the others.

This chapter provides detailed background about the state-of-the-art processor architectures and hardware and software aspects of power management. Specifically, Section 2.1 provides an overview of the Intel Skylake microarchitecture. Section 2.2 describes the processor power, consumption model. Section 2.3 describes the evolution of power management features in Intel processors over the years. Section 2.4 introduces the Advanced Configuration and Power Management (ACPI) standard used by hardware and operating systems (OS) vendors. Section 2.5 describes the power management hierarchy and its components. Finally, Section 2.6 explains the functioning of the most common DVFS based *ondemand* governor.

2.1 Intel Skylake Microarchitecture: An Overview

Intel processor releases are based on a "tick-tock" development process. At first, a "tock" comes with a new microarchitecture that uses the same technology node as the previous generation. 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 development costs. Figure 2.1 illustrates the "tick-tock" for 11 generations of Intel desktop (Core) and 8 generations of server processors (Xeon).

Continual transistor size reduction has played a key role in speed and energy improvements. But for four full generations of the Intel Core processors, the same technology node of 14 nm has been used with slight process refinements. This shows a break from the traditional "tick-tock" approach. Though the technology feature 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 for each new generation of processors.



Intel Microarchitecture Codename Nehalem		Intel Microarchitecture Codename Sandy Bridge		Intel Microarchitecture Codename Haswell		Intel Microarchitecture Codename Skylake				
45nm	32nm	32nm	22nm	22nm	14nm	14nm	14+nm	14++nm		
Desktop/Workstation Processor Codename										
Lynnfield	Clarkdale	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake	Kaby Lake	Coffee Lake	Coffee Lake Refresh	Comet Lake
Server/Datacenter Processor Codename										
Beckton	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake		Cascade Lake		
Tock	Tick	Tock	Tick	Tock	Tick	Tock	-	-	-	-
2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2020

▨ New Microarchitecture 🟦 New Process Technology

Figure 2.1 Intel's Tick-Tock Model for Desktop and Server Processor [74]

Each generation of Intel microarchitecture contains two variants, the microarchitecture for the Core processors and the Xeon processors. Though the internal core architecture is similar, the design of the Xeon processors is oriented towards prolonged usage, higher scalability, and lower power consumption. A Xeon processor is usually clocked at a lower clock frequency than the corresponding Core processor, in order to have a lower operating temperature.

The most recent iteration of Intel Core architecture comes under the name Skylake. Skylake is the successor to Broadwell in terms of the technology node and includes a number of improvements relative to the Haswell microarchitecture. Five generations of processors were built using the Skylake microarchitecture. This section provides a brief review of the Skylake microarchitecture.

2.1.1 Processor Core Microarchitecture

A physical core (also referred to as 'core') is a well-partitioned piece of logic capable of independently performing all functions of a processor. A single physical core may encompass one or more logical cores. The Intel Skylake microarchitecture specifies an out-of-order superscalar design that can dispatch up to six microinstructions to execution units per a single CPU clock cycle. The internal functional units can be segregated into the front-end and the back-end. 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.

Figure 2.2 gives the block diagram of the Skylake microarchitecture. The flow of instruction through the pipeline can be illustrated as follows. Initially, the branch prediction unit (BPU) chooses the next 16-byte block of instructions to execute. The processor then searches for instructions in the Decode ICache, first-level instruction cache (L1I), 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 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 the retirement of the faulting instruction.

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 next instruction to be executed but also based on the execution path. The BPU can efficiently predict the following types of branches:

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

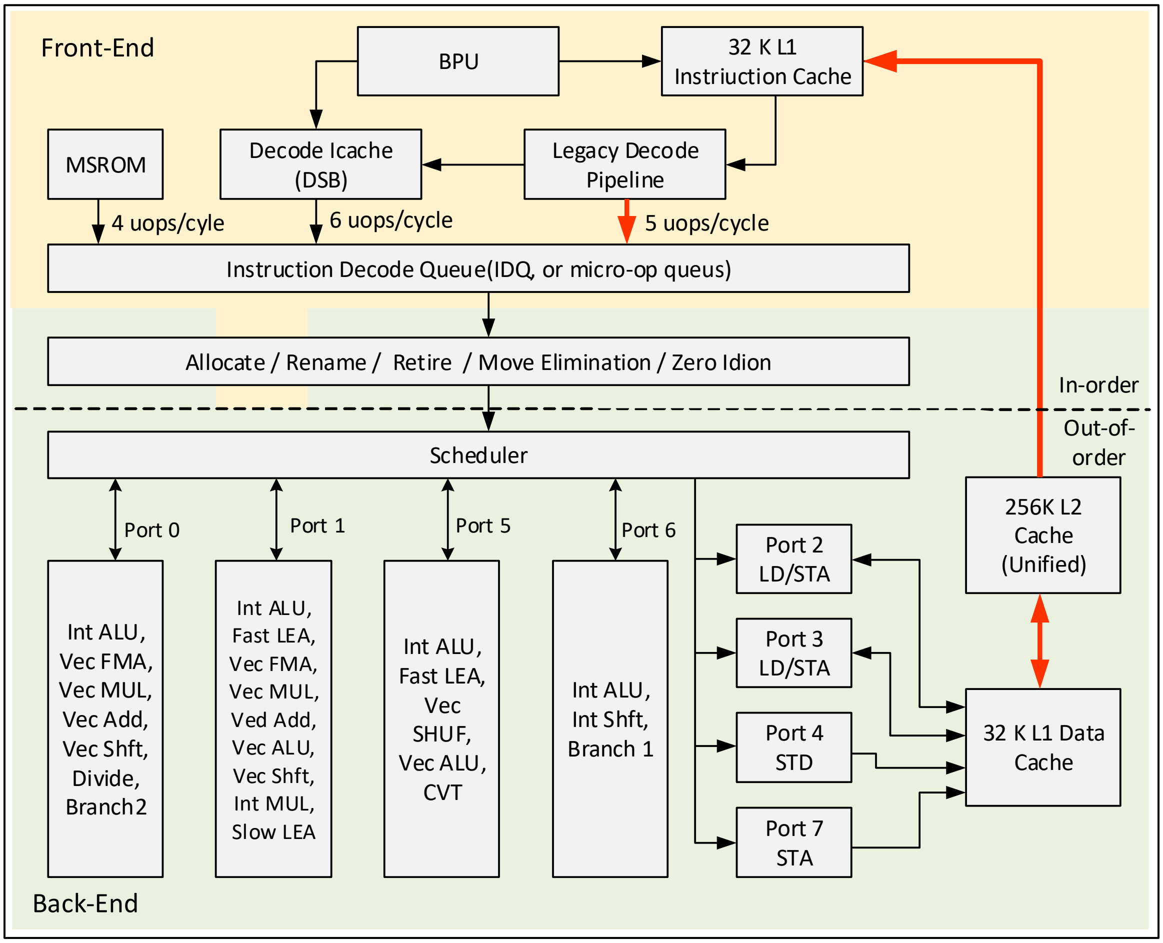

Front-End
BPU
32 K L1 Instriuction Cache
MSROM
Decode Icache (DSB)
Legacy Decode Pipeline
4 uops/cyle
6 uops/cycle
5 uops/cycle
Instruction Decode Queue(IDQ, or micro-op queus)
Allocate / Rename / Retire / Move Elimination / Zero Idion
In-order
Out-of-order
Scheduler
Port 0
Port 1
Port 5
Port 6
256K L2 Cache (Unified)
Int ALU, Vec FMA, Vec MUL, Vec Add, Vec Shft, Divide, Branch2
Int ALU, Fast LEA, Vec FMA, Vec MUL, Ved Add, Vec ALU, Vec Shft, Int MUL, Slow LEA
Int ALU, Fast LEA, Vec SHUF, Vec ALU, CVT
Int ALU, Int Shft, Branch 1
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
32 K L1 Data Cache
Back-End

Figure 2.2 Skylake Microarchitecture CPU Core Block Diagram [67]

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 [41]. Unfortunately, the branch predictor organization and operation on the Skylake architecture are not disclosed by the manufacturer. Studies have used experimental reverse engineering and it is found that the branch predictor unit used in the older generations of the Intel processors is a 4096-entry bimodal predictor [40] [61] [62].

The back-end, also known as the out-of-order (OOO) engine, detects dependency chains and sends those chains 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 out-of-order engine consists of three execution stacks, where each stack encapsulates a certain type of data: a general-purpose integer, a 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 eight micro-operations every cycle, one on each 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.

2.1.2 Cache Hierarchy

The cache hierarchy contains a first-level instruction cache, a first-level data cache (L1 DCache), and a second-level cache (L2), that are private to each processor core. The caches may be shared by two logical processors if the processor is hyper-threaded. The L2 cache is unified, containing both 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 below it.

The actual delay a CPU core sees when reading a data item depends on 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 cache line of interest and its state needs to be modified, there is a cache coherence lookup into the L1 DCache and L2 of these cores too.

Table 2.1 shows the size, associativity, and access times in the memory hierarchy of a typical Skylake based quad-core processor. The overall memory structure in Skylake is similar to its predecessor Broadwell/Haswell, except for the change in associativity of L2 to 4-way from the previous 8-way. The data access latency is dependent on the operating clock frequency of the core and uncore. A higher operating frequency would reduce the wall-clock period for the access latency.

Table 2.1: Memory Hierarchy Parameters in a Typical Skylake Processor

	Size	Associativity	Latency
L1 DCache	32 KB	8-way	4 cc
L1 ICache	32 KB	8-way	5 cc
L2 Cache	256 KB	4-way	12 cc
L3 Cache	8 MB	16-way	42 cc
DRAM	-	-	42 cc + 51 ns

Figure 2.3 shows the best-case access latency in ns for all levels in the memory hierarchy of the test machine while varying operating frequencies. These data are collected using the Hopscotch benchmark suite [1]. We can see how the changes in the processor clock frequency impact the access latency for L1, L2, L3, and DRAM. By lowering the processor clock frequency, expectedly the latencies increase at each level.

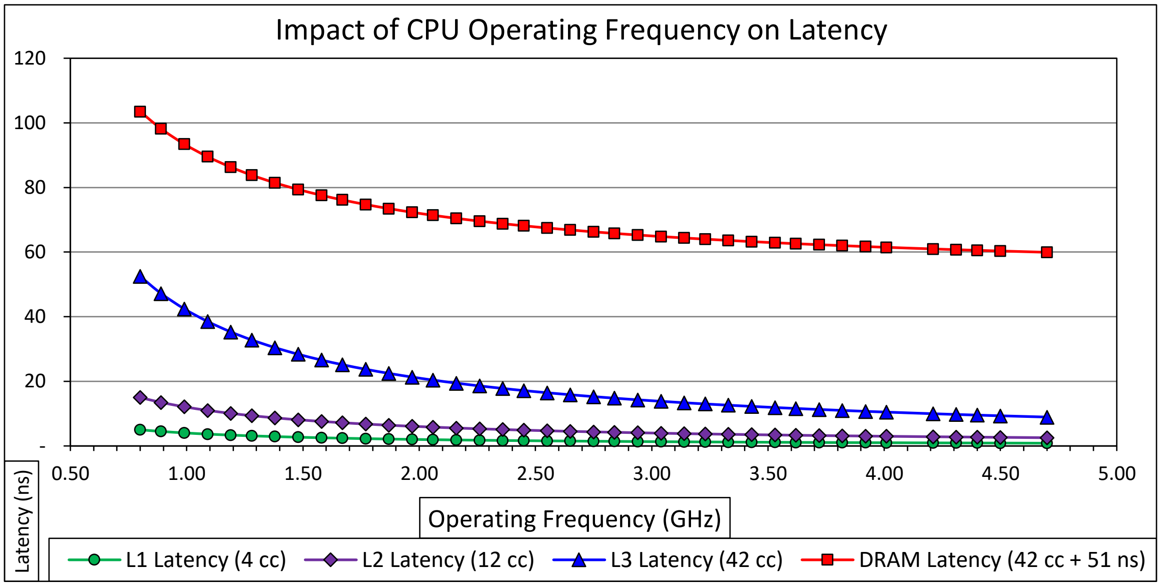


Figure 2.3 Measured Cache Hierarchy Access Latency

2.2 Processor Power Consumption Model

CPU power can be conceptually broken into (a) the logic power and (b) the I/O power. The two major components of the logic power are: (i) the power consumed by the clocks that run throughout the processor; (ii) power consumed by logic performing computation [21]. The power consumed by the logic elements performing computation can be further divided into dynamic power, short-circuit power, and leakage power as shown in Eq. 2.1 and Eq. 2.2 [42].

$$P_{CPU} = P_{dyn} + P_{sc} + P_{leak} \qquad \text{Eq. 2.1}$$

$$P_{CPU} = ACV^2f + \tau AVI_{short}f + VI_{leak} \qquad \text{Eq. 2.2}$$

The first component of the equation, dynamic power consumption is caused by the charging and discharging of the capacitive load on the output of each gate. It is proportional to the frequency of the system's operation, *f*; the activity of the gates in the system, *A,* a metric that captures how often gates are switching; the total capacitance seen by the gate's outputs, *C*; and the square of the supply voltage, *V*. The second component of the equation short-circuit power captures the power expended as a result of short circuit current, I_{short}, which momentarily, τ, flows between the supply voltage and ground when a CMOS logic gate's output switches. The third component measures the power lost from the leakage current regardless of the gate's state.

For a long time, dynamic power consumption was the major factor influencing total power consumption. The most effective way to save power was by reducing the supply voltage, *V*. The quadratic dependence on *V* means that the savings can be significant: Halving the voltage reduces the power consumption to one-fourth of its original value. Unfortunately, this savings comes at the expense of performance, or, more accurately, maximum-operating frequency, as shown in Eq. 2.3.

$$f_{max} \propto \frac{(V - V_{threshold})^2}{V} \qquad \text{Eq. 2.3}$$

The maximum frequency of operation is thus proportional to *V*. Reducing it limits the maximum frequency the circuit can run at. Reducing the power supply to one-fourth of its original value only halves the maximum frequency. However, reducing the voltage, *V*, in Eq. 2.3 requires a reduction in $V_{threshold}$. This reduction must occur so that low-voltage logic circuits can properly operate. However, reducing $V_{threshold}$ increases the leakage current, as shown in Eq. 2.4

$$I_{leak} \propto exp\left(\frac{q\,V_{threshold}}{kT}\right) \qquad \text{Eq. 2.4}$$

Eq. 2.1 represents the average power consumed by the CPU. Although the dynamic power consumed is perceived as a function of voltage, frequency, and temperature, each of these components has a direct and proportional impact on the behavior of every other parameter. The power consumption is proportional linearly to frequency and quadratically to voltage as shown in Eq. 2.5.

$$P_{dyn} \sim f * V^2 \qquad \text{Eq. 2.5}$$

In order to increase the operating frequency, the voltage has to be increased. The voltage required to run the CPU tends to increase with the square of the frequency in operating regions with a very high clock frequency. As the power consumed is directly dependent on voltage and frequency this relationship is critical for power management. At low frequencies, we can change the frequency with little impact on voltage, however, when operating at high frequencies, a small increase in frequency requires a large variation in voltage.

2.3 Evolution of Power Management in Intel Processors

Over the past two decades, power has become a primary design constraint in the design of modern processors. In response, computer architects have significantly improved the energy efficiency infrastructure in modern processors. With each new processor generation, additional energy efficiency features were introduced, resulting in power savings by at least a factor of four in idle systems. While these features improved the energy efficiency significantly, they also have a major influence on the performance of the processor. In this section, we will go over the evolution of the power management features on different microarchitectures from Intel over the years.

2.3.1 Nehalem Microarchitecture

The Intel Nehalem microarchitecture, released in late 2009, was the basis for the 1st generation of the Intel core processors. The corresponding processors were initially manufactured on a 45nm technology node and later upgraded to a 32nm technology node the next year.

2.3.1.1 Power Control Unit (PCU)

To tackle the growing problem of leakage power, which was responsible for roughly 1/3rd of the core power consumption, new power management features had to be developed. As a solution, the first on-chip power control unit or the package control unit (PCU), which was built using over a million transistors, and was introduced in the architecture. The PCU consolidated all the power management features present on the processor, including the ACPI interface that controls the P-states and the C-states to one module. The PCU runs proprietary firmware and provides interfaces to the BIOS or OS with a set of control and model-specific registers (MSRs).

Figure 2.4 shows the positioning of the PCU on the chip of a Nehalem processor. The controller is responsible for managing the power states of the processing cores using real-time sensors for temperature, current, and power. The on-chip power management improved voltage switching rates resulting in a P-state transition latency of ~100 ms. The P-state management in Intel Nehalem processors is called, *SpeedStep Technology* (software P-state management).

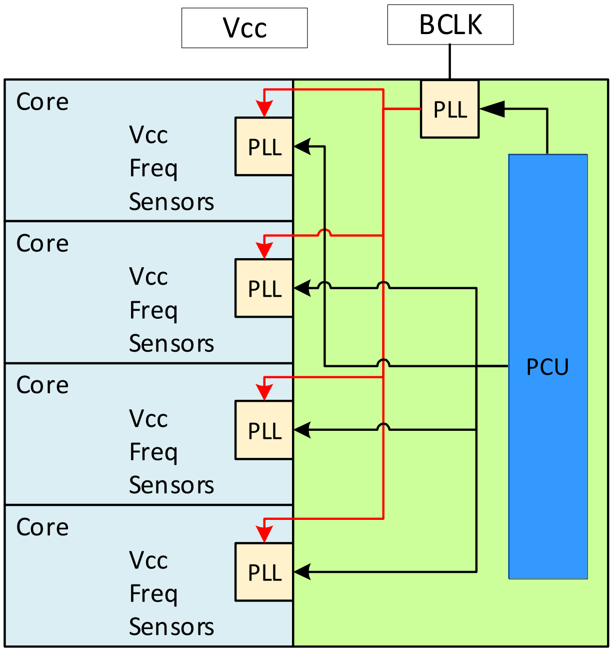


Figure 2.4 Integrated PCU on the Nehalem Processor

The *SpeedStep* implementation provides each physical core with its own integrated phase-locked-loop (PLL), enabling it to be clock gated independently, allowing for core-level C-states. The external clock source of 133 MHz is brought to the processor chip. A new power gate was designed for the Nehalem architecture. The outcome was for the first time; an un-used processor core power consumption can be completely reduced to zero by placing it into the C6 (“deep power down”) power state independently. It should be noted that though each physical core has an independent PLL, the operating voltage and the frequency of the cores are the same and they operate in the same voltage domain.

An added advantage of the modular design was the decoupling of the core and uncore domains. As a result, the uncore to be powered down when all cores enter the C6 sleep state. However, the uncore savings do not scale similarly to the core as even a single active core can wake the uncore from the sleep state.

2.3.1.2 Intel Turbo Boost Technology

The savings in the power budget paved the way for the introduction of the turbo-mode. The basic premise of the turbo-mode is to use the power budget surplus from turning off unused cores to temporarily increase the operating frequency of the active cores. Figure 2.5 provides an illustration of the turbo mode on a 4-core Nehalem processor. When all four cores are loaded, the processor operates at the specified thermal design power (TDP).

TDP, in watts, refers to the power consumption under the maximum theoretical load. However, in the case of a lightly threaded workload occupying only 2 cores, the remaining cores can be put to sleep, providing power and thermal headroom for turbo mode. All Nehalem processors were capable of at least boosting a single clock step (133 MHz) in turbo mode, even if all cores are active, for as long as the PCU does not detect any violation in the TDP. If the TDP levels are low enough, or if several cores are idle, the PCU can increase clock speeds by more than one clock step. However, the Turbo technology in Nehalem was limited to just two clock steps, providing a maximum turbo boost of 266 MHz above the nominal frequency [75].

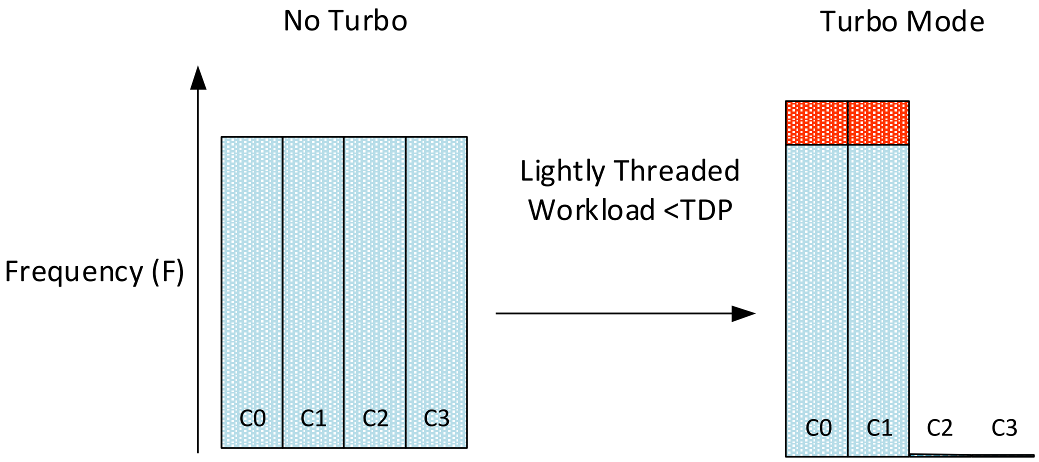


Figure 2.5 Illustration of Turbo Mode

2.3.2 Sandy Bridge Microarchitecture

The Intel Sandy Bridge microarchitecture, released in 2011 is an evolution of the Nehalem microarchitecture. It was the core microarchitecture for the 2nd and 3rd generation of the Intel core processors. The first wave of processors used the earlier 32 nm technology node and later upgraded to 22 nm under the code-name Ivy Bridge. The PCU, introduced in the Nehalem architecture, received several feature updates. Figure 2.6 shows the block diagram of the major functional blocks and the power-management control blocks and interconnect on the Sandy Bridge microarchitecture. The PCU resides in the system agent and is a combination of dedicated hardware state machines and an integrated microcontroller. A power-management link connects the PCU to different cores and functional blocks on the die via power management agents (PMAs). PMAs collect telemetry information such as power consumption and junction temperature and perform control functions such as P-state and C-state transitions. The PCU communicates to the external voltage regulator and embedded controller that performs system power-management functions. The PCU runs firmware that constantly collects power and thermal information, communicates with the OS, and performs various power-management functions and optimization algorithms.

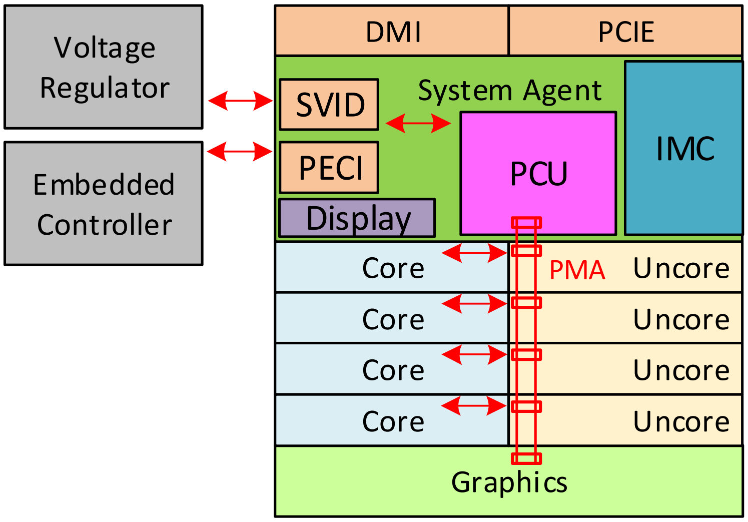


Figure 2.6 Sandy Bridge Power Management Block Diagram [52]

Sandy Bridge's package implements two independent variable power planes. The first one is a shared power plane, that feeds all CPU cores, the ring interconnect, and the last level cache (LLC). Embedded power gates turn each core on and off individually. The LLC's power gates can turn on or off portions of the cache in shallow package sleep states or all of the cache in deeper sleep states. All the cores and the ring share the same clock and perform dynamic voltage and frequency scaling together. The second power plane is the graphics processor. It has an independent power plane, whose voltage and frequency can be varied independently. It can also be turned off completely when the graphics are inactive. Additional fixed power planes control the system agent and I/O [52].

2.3.2.1 Intel Turbo Boost Technology 2.0

P-state management in Sandy Bridge processors is termed *Enhanced Intel SpeedStep Technology*. A major update came in the form of a revised functioning of the turbo-mode, called Intel Turbo Boost technology 2.0. The processors from the previous generation (Nehalem) limited the turbo mode to match the TDP budget, based on the assumption that the CPU reaches that TDP immediately upon enabling turbo mode. However, in reality, the CPU temperature changes more gradually – there is a period of time where the CPU is not dissipating its full TDP – this behavior is similar to a ramp function.

Sandy Bridge takes advantage of this by allowing the PCU to enable turbo-mode on active cores above the TDP budget for a short period of time (up to 25 seconds). The PCU keeps track of the available thermal budget while idle and spends it when CPU demand goes up. The longer the CPU remains idle, the more potential it has to ramp up above TDP during a high load period. During workload execution, the CPU can turbo above its TDP and step down, as the processor heats up, eventually settling down at its TDP [76].

In addition to the above-TDP-turbo, Sandy Bridge also supported more turbo bins than Nehalem and allowed for both CPU and GPU turbo to work in tandem. Workloads that are more GPU bound can result in the CPU cores clocking down and the GPU clocking up and vice-versa.

2.3.2.2 Running Average Power Limit (RAPL)

With the introduction of the above-TDP-turbo, a robust hardware mechanism was required to monitor and control power consumption on the chip to avoid thermal damage. The Running Average Power Limit (RAPL) interface was designed to limit on-chip power while ensuring maximum performance [66]. The interface supports fine-grain time measurement of power, energy, and temperature of sockets, individual cores, uncore structures as well as on-chip GPUs. The RAPL interface acts as an architectural power meter. It collects a set of architectural events from each Intel architecture core, the processor graphics, and I/O, and combines them with energy weights to predict the package's active power consumption.

In RAPL, platforms are divided into domains for fine-grained reporting and management. Figure 2.7 shows the major RAPL domain available on the processor. This includes the package domain (PKG) which incorporates the entire socket, the core domain (PP0) which includes all the CPU cores, the graphic domain (PP1) which includes the onboard graphics, and the memory domain (DRAM). The specific RAPL domains available in a platform vary across product segments.

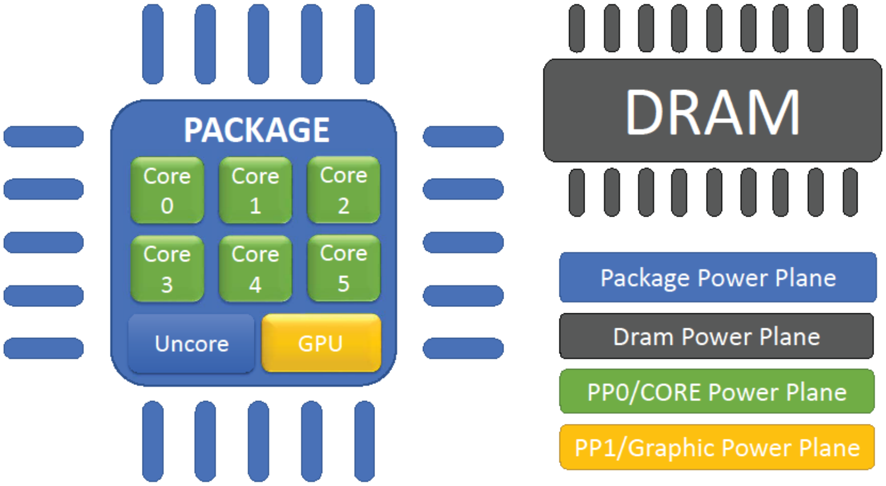


Figure 2.7 RAPL Power Domains

Each RAPL domain supports four different functionalities as shown below:

- ENERGY_STATUS for power monitoring.
- POWER_LIMIT and TIME_WINDOW for controlling power.
- PERF_STATUS for monitoring the performance impact of the power limit.
- RAPL_INFO contains information on measurement units, the minimum and maximum power supported by the domain.

Intel has validated the energy estimates provided by the RAPL interface to actual power consumption. Several studies have explored the effectiveness of on-chip power meters and explained hardware and software optimizations as a function of performance and energy efficiency [22]. Various tools make use of the RAPL interface to enable power and energy measurements of different power domains [64] [60].

2.3.3 Haswell Microarchitecture

The Intel Haswell microarchitecture introduced in 2013 is the core microarchitecture for the 4th and 5th generation of the Intel Core processors. The fourth-generation used the 22 nm technology node and the fifth upgraded to 14 nm under the code-name Broadwell. The Haswell microarchitecture was optimized for idle power consumption and consequently, several new power management features were added.

2.3.3.1 Per-Core Power Management and Independent Uncore Scaling

Intel processors from the Haswell microarchitecture were the first x86 processors that incorporated fully integrated voltage regulators (FIVR) on the die [9]. Additionally, server-class processors included separate voltage regulators for every processor core, enabling fine-grained P-state control. The on-chip voltage regulators also paved the way for uncore frequency scaling (UFS), enabling the processor to control the frequency of the uncore components (e.g., last-level caches) independently of the core frequencies. Prior Intel processor generations used either a fixed uncore frequency (Nehalem and Westmere) or a common frequency for cores and uncore (Sandy Bridge and Ivy Bridge). The uncore frequency has a significant impact on on-die cache-line transfer speeds as well as on memory bandwidth [31]. At the microarchitecture level, Intel added more power gating and low power modes. The additional power gating gives the PCU fine-grained control over shutting off parts of the core that are not used.

Furthermore, a major focus on vectorization resulted in the expansion of an advanced vector instruction set (AVX), supporting 256-bit wide data paths. However, AVX instructions draw more current and a higher voltage is needed to sustain operating conditions. To facilitate this, the core signals the PCU to provide additional voltage and slows the execution of AVX instructions. To maintain the limits of the TDP, the increasing voltage may cause a drop in clock frequency. Hence, the Haswell CPU family uses a lower clock frequency for workloads with a substantial portion of AVX instructions [22]. To cope with the huge difference between the power consumption of scalar and AVX instructions, a new base and Turbo Boost frequencies called AVX base/Turbo was introduced, as shown in Figure 2.8.

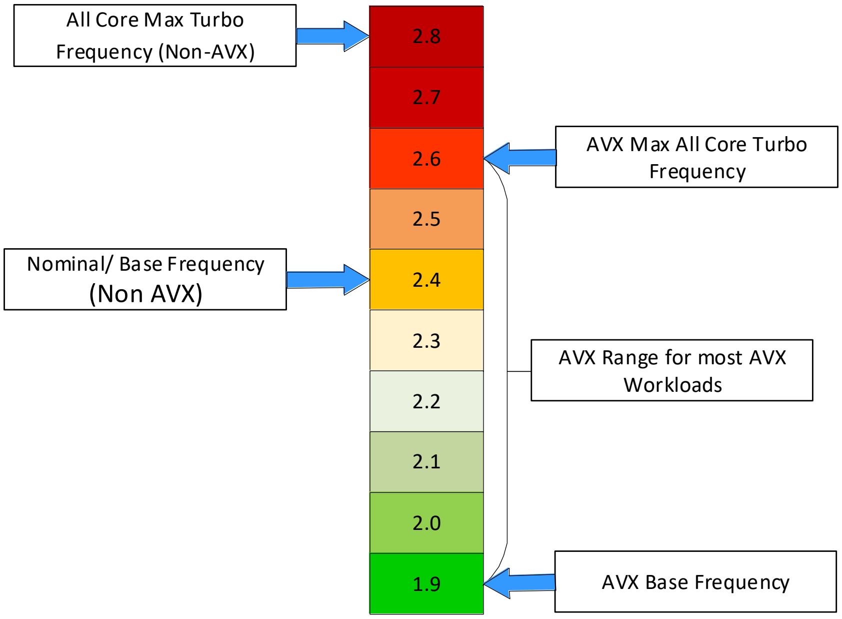

All Core Max Turbo Frequency (Non-AVX)
2.8
2.7
2.6
AVX Max All Core Turbo Frequency
2.5
Nominal/ Base Frequency (Non AVX)
2.4
2.3
AVX Range for most AVX Workloads
2.2
2.1
2.0
1.9
AVX Base Frequency

Figure 2.8 AVX Frequency Range in a Haswell Processor

Turbo-boost in Haswell/Broadwell processors saw several updates. Figure 2.9 illustrates the operation of turbo-mode on a 4-core processor. The processor will have a certain number of turbo bins, controlled by the PCU, available based on the rated TDP. Monitoring the CPU load, thermal headroom, and power budget, the PCU allocates these bins to one or more processor cores. This revision to the turbo mode includes the introduction of the Energy Efficiency Turbo (EET) [6].

High turbo frequencies—typically only limited by power or thermal constraints—tend to hurt energy efficiency, especially if the performance increase is negligible. The EET feature attempts to reduce the usage of turbo frequencies that do not significantly increase the performance. EET monitors the number of stall cycles and uses this information as well as the energy performance bias (EPB) setting to select a turbo-frequency that is predicted to be optimal.

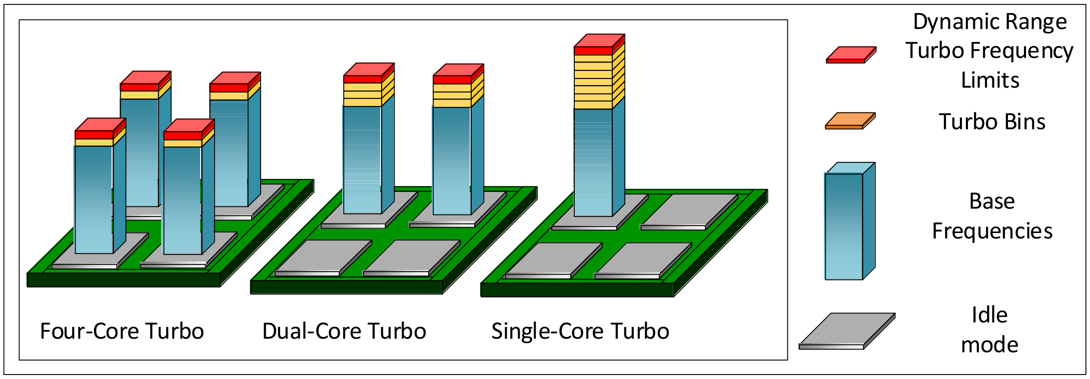


Figure 2.9 Turbo Operation in Haswell/Broadwell Processors

2.3.3.2 Hardware P-state Management

All prior generation processors relied on the OS to be in control of the on-chip power management features such as selecting the P-states and the C-states based on CPU utilization. This causes congestion in the OS control loop, which interrupts the workload regularly. To tackle this problem Hardware-Controlled P-states (alias

Hardware Power Management (HWPM or SpeedShift) was introduced in the Broadwell generation of Intel processors. The hardware-controlled P-states mechanism transfers the decision of frequency scaling from the OS to the hardware and acts autonomously. Furthermore, it increases the responsiveness because the hardware control loop can be executed more frequently without perturbation.

2.3.3.3 Intel Turbo Boost Technology 3.0

Due to variations in their manufacturing process, individual cores in the same die may have varying efficiency characteristics. As a consequence, during turbo-mode, some cores may reach a higher operating frequency while other cores may not, which in turn influences the performance of a single thread depending on the hardware core that executes on. To overcome this problem, the Turbo Boost Max 3.0 (TBM3) feature was introduced with the Broadwell processors. Its basic premise is to improve single-thread performance by executing the workload on the processor core that delivers the best power and performance. The PCU can automatically select the best performing core and ask the schedular to execute the workload on the given core.

2.3.4 Skylake Microarchitecture

The Intel Skylake microarchitecture was released at the end of 2015 and was the core architecture for the five generations of the Intel processor series (from 6th generation to 10th generation). Skylake was a "tock" in Intel's cycle, hence it used the same 14-nm technology node as Broadwell with some process refinements. Figure 2.10 shows a block diagram of an Intel Skylake processor with four different power domains, as follows: processor cores, uncore, graphics, and system agent. The PCU is in charge of power management; it includes a microcontroller that runs proprietary firmware and provides interfaces to the BIOS or OS [77].

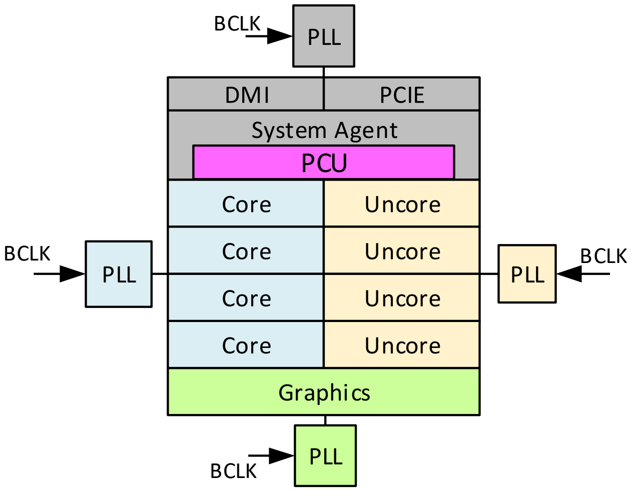


Figure 2.10 Illustration of Power Domains on an Intel Processor

The PCU monitors the state of individual power domains and carries out power management requests, including power gating of individual domains and adjusting their frequencies and power supply voltages. The voltage regulators and an external clock source of 100 MHz are brought to the processor chip. On-chip PLLs generate internal clock frequencies for the individual power domains. Whereas Figure 2.10 shows a single voltage domain for all four processor cores, server processors may support separate voltage domains for individual physical processor cores.

2.3.4.1 Energy Efficiency Mechanism

The FVR introduced in the Haswell processor was removed and the voltage regulators were moved back to the motherboard in the Skylake processors. Like its predecessors, Skylake processors support per-core P-states and Uncore Frequency Scaling. This enables fine-grained control over performance and energy efficiency decisions. The Energy Performance Bias (EPB) indicates whether to balance the profile for runtime or power consumption or something in between. The Energy-Efficient Turbo (EET) mechanism was inherited from the Haswell microarchitecture. The hardware P-state management (a.k.a. Intel Speedshift) saw a major update. While the Broadwell processors hardware acts mostly autonomously, Skylake processors provide interfaces for a collaboration with the OS through interrupts. With the HWP interface, the OS can define a performance and power profile, and set a minimal, efficient, and maximal frequency. The OS can also override the hardware in selecting a P-state.

Table 2.2 shows the P-state transition on the latest processors from the Skylake microarchitecture. Compared to Speed Step- P-state transitions, the Speed Shift terminology improves transition times by having the operating system relinquish some or all control of the P-States and handing that control off to the processor. This has a couple of noticeable benefits. First, it is much faster for the processor to control the changes in clock frequency, compared to OS control. Second, the processor has much finer control over its states, allowing it to choose the most suitable performance level for a given task. Specific jumps in frequency are reduced to around 1 ms with Speed Shift's CPU control from 10-30 ms on OS control and going from the lowest P-state (Pn-energy-efficiency state) to the lowest P-state (P0-maximum performance can) be done in around 35 ms, compared to around 100 ms with the legacy implementations. This improvement in transition time is especially beneficial for latency-sensitive application and interrupt handling.

Table 2.2: P-state Transition Latency Reported by Intel

	SpeedStep	SpeedShift
P-state Transition	~10-15 ms	~1 ms
The transition from Pn to P0	~100 ms	~35 ms

In summary, each new generation of the processor builds on the energy-efficiency features of the prior generation. Utilizing the multitude of hardware features focused on power management, operating system vendors, over the years have tried to optimize application performance and save power. A number of various governors are built to target different use cases. Generally, the governors follow a strategy of “race to idle”, which relies on finishing execution quickly in order to save power. However, we learn through experimentation that this strategy is not ideal for all sorts of applications.

BIOS/OS developers utilize the available hardware structures to build high-level control algorithms for power management. Major computing companies developed an open industry specification called *Advanced Configuration and Power Interface* (ACPI) to maintain uniformity across processor vendors, OEMs, and OS providers [78]. ACPI establishes common interfaces for power management in a variety of computer systems.

2.4 ACPI Power & Performance States

The primary objective of power management techniques is to reduce overall power consumption when possible, without affecting performance. Two primary ways to reduce power consumption in modern processors are to either turn unused components off or to throttle used components based on their load. To facilitate these actions, modern processors feature power states (C-states) that facilitate turning off individual processor components when idle and performance states (P-states) that facilitate clock frequency and voltage throttling.

Figure 2.11 illustrates C-states and P-states as defined by the ACPI standard. The C0-state corresponds to the processor active mode, where all components are turned on and component clocks are active. Within this state, multiple P-states are available, enabling dynamic changes of the processor clock frequency and power supply voltage. The P0 state corresponds to the processor's highest operating clock frequency in the so-called turbo mode [10]. The P1-state typically corresponds to the nominal or base processor clock frequency. Turbo Boost is a technology initially introduced by Intel that opportunistically allows the processor to run faster than the nominal frequency if the processor operates below power, temperature, and current limits. The maximum Turbo Boost frequency depends on the number of active cores, workload, operating environment, and platform design. (Note that Turbo Boost is not the same as overclocking). Max Turbo Boost frequency is dependent on the number of active cores, workload, operating environment, and platform design. Higher P-states (P2-Pn) progressively lower processor clock frequency and power supply below their nominal levels.

Higher C-states (C1-Cn) progressively turn off unused components, entering deeper sleep modes, thus eliminating both the switching and leakage components of power consumption. C1 is the first idle state, a.k.a. *Halt*. In C1 the processor clock is gated, i.e., the clock is prevented from reaching the core(s), effectively shutting them down. However, the clock can be restored almost instantaneously (with a few clock cycles delay) to return to the active state. Higher C-states (C2-Cn) offer larger power savings, albeit at the cost of increased wake-up time. Each new generation of modern processors introduces a larger number of C- and P-states, faster and more efficient transitions between the C states, and a richer set of functions for power management [22] [54].

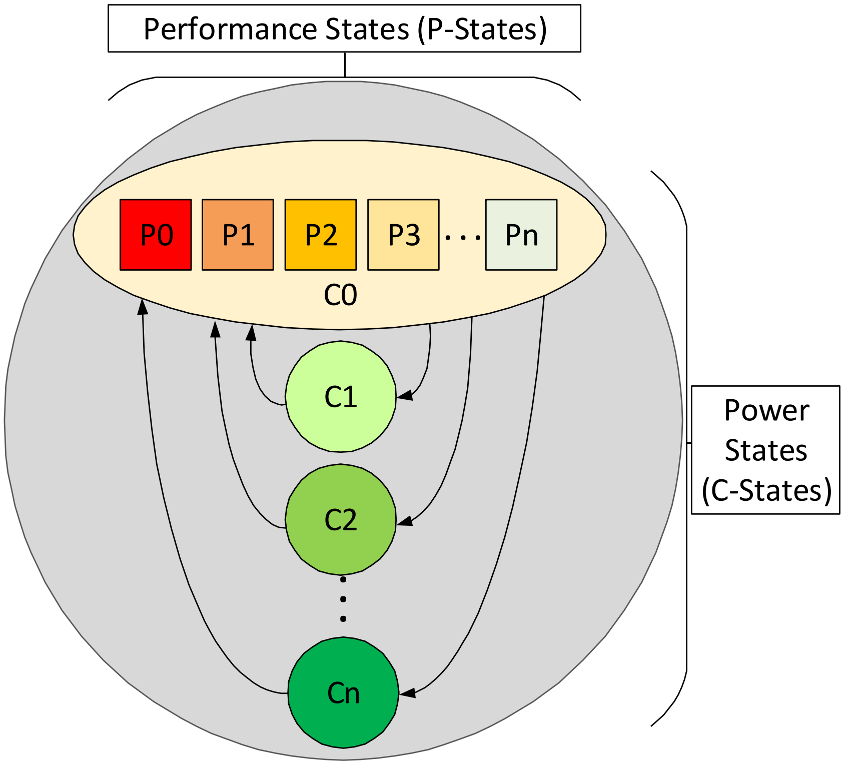


Figure 2.11 Processor Power States (C-states) & Performance states (P-states)

2.5 CPU Power Management

Figure 2.12 provides a hierarchical view of the various power management components, from hardware to userspace interface. Starting from the bottom up, the hardware level encompasses the power control unit (PCU/P-unit) with a set of control and model-specific registers (CSRs and MSRs). During an initial handshake at bootup, the processor provides information to the BIOS about available P- and C-states. Further communication to inspect the current state or initiate a state change is carried out through the status and control registers (shown at the bottom of the figure). The BIOS can typically support multiple system profiles that can favor performance, energy efficiency, or allow for dynamic power-saving techniques. Considering the latter, the power management control is transferred to the operating system. This profile is often referred to as *Performance Per Watt OS* or *OS Control Mode*.

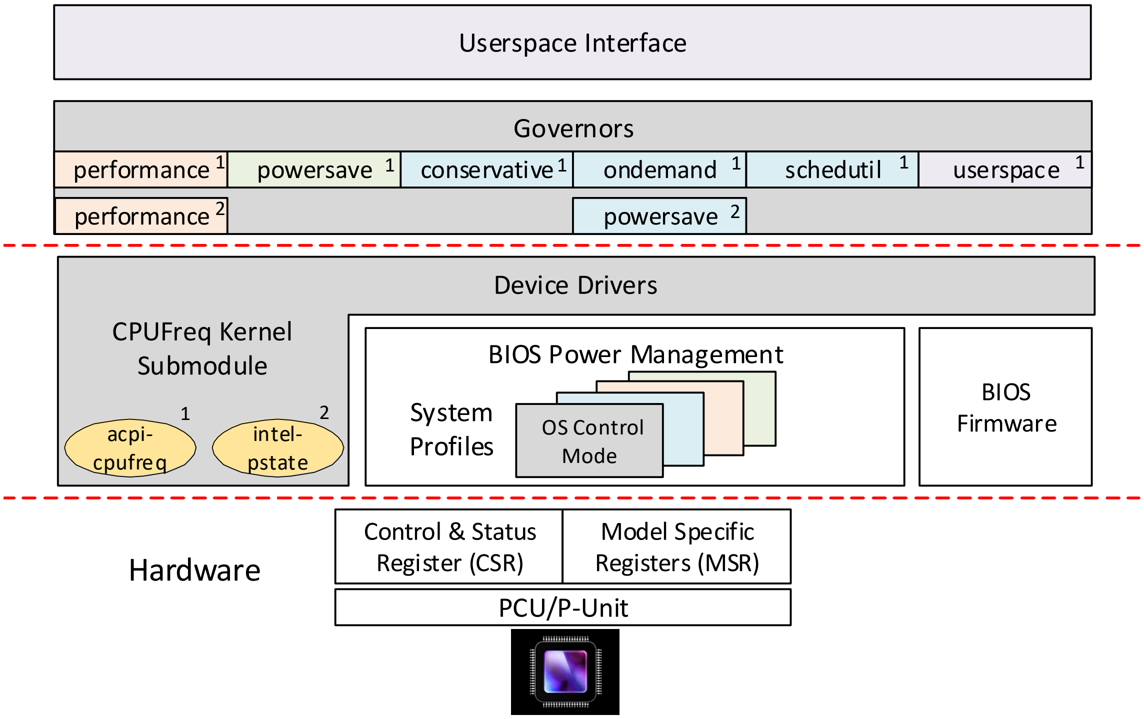


Figure 2.12 A Hierarchy of Power Management Components

To abstract out differences between various hardware implementations across multiple generations of processors, vendors provide transition drivers such as the Intel P-state driver (the default for Intel processors) and the CPUFreq driver (the default for AMD processors). These drivers act as an interface between the PCU and a set of defined governors residing in the OS. Governors implement a particular policy that determines when and how the processor frequency and voltage are scaled.

Generic governors supported by the Linux *acpi-cpufreq* driver are shown in Figure 2.12 and they can be broadly classified into two groups, static frequency selection governors and DVFS-based governors. The static frequency governors, such as the *performance* and *powersave* governors, set the processor frequency to the highest (P0) and lowest (Pn) available clock frequency, respectively. The *performance* governor is utilized for latency-sensitive workloads to minimize their response time and execution times. However, this policy can quickly lead to overheating and it tends to be wasteful when the system is idle or underutilized. On the other side, the *powersave* governor will guarantee the lowest-power operation, at the expense of increased execution time. It should be noted that running at the lowest clock frequency may significantly increase the execution time so that the overall energy exceeds the energy required at other operating points.

To bridge the gap between the *performance* and *powersave* governors, the governors that employ DVFS are utilized. The *ondemand* governor automatically selects the highest frequency when the average processor load exceeds a certain threshold. The governor keeps track of the average processor load determined by the scheduler. If the load falls below a certain threshold the clock frequency is lowered accordingly. The conservative governor is similar to the *ondemand* one; the only difference is that changes in the clock frequency occur more gradually. The *schedutil* governor is also similar to *ondemand* and allows for scheduler-driven processor frequency selection. Finally, the *userspace* governor allows the user to set a specific clock frequency statically.

The *ondemand* generic governor is recommended when using the *acpi-cpufreq* driver. When using the recent *intel_pstate* driver, only two governors are supported referred to as the *performance* and *powersave*. However, although these two governors share the names of the generic governors, they behave differently. They both provide dynamic voltage and frequency scaling, similar to the generic *schedutil* or *ondemand* generic governors. In the rest of the paper, we will exclusively use the governors that rely on the *intel_pstate* driver.

2.6 Functioning of a DVFS-based Governor

The governors that employ DVFS such as *ondemand* use CPU utilization as the primary metric in determining appropriate P-states [46]. The CPU utilization is separately provided by the scheduler for each CPU core at a fixed time interval, typically 1 ms. The CPU utilization metric is calculated as the percentage of time spent in the non-idle thread for a given time interval, as shown in Eq. 2.6.

$$\%\, CPU\ Utilization = \frac{time\ in\ non\ idle\ thread}{time\ intervel} * 100 \qquad \text{Eq. 2.6}$$

Figure 2.13 illustrates the P-state selection mechanism on a single core over a period of time-based on CPU utilization. In this example, we assume that the processor supports 11 P-states, P0 to P10. The scheduler monitors and updates the CPU utilization every 1 ms and the *ondemand* governor linearly maps the CPU utilization to the available P-states and sends a request for the next P-state every 10 ms. However, a transition between the P-states takes a finite amount of time. This latency has been reduced over many generations of processors and is currently ~10 ms. The same technique is applied to all the cores visible to the operating system.

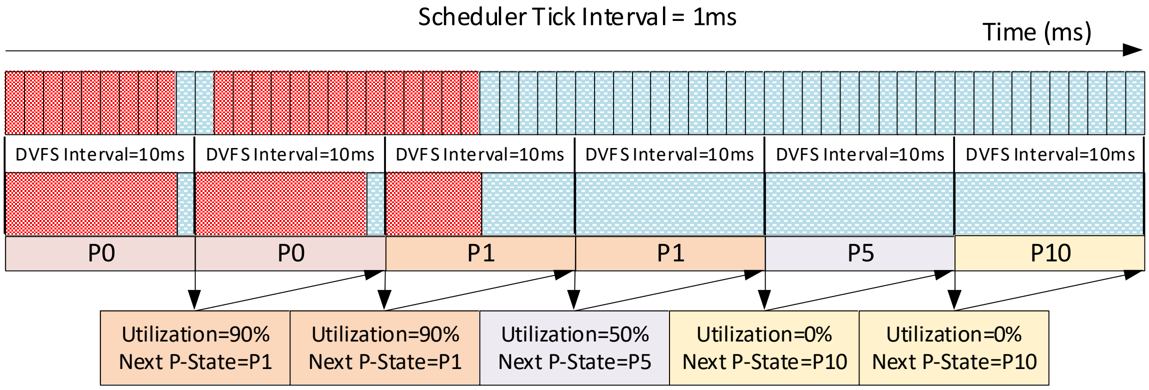

Scheduler Tick Interval = 1ms
Time (ms)
DVFS Interval=10ms
DVFS Interval=10ms
DVFS Interval=10ms
DVFS Interval=10ms
DVFS Interval=10ms
DVFS Interval=10ms
P0
P0
P1
P1
P5
P10
Utilization=90% Next P-State=P1
Utilization=90% Next P-State=P1
Utilization=50% Next P-State=P5
Utilization=0% Next P-State=P10
Utilization=0% Next P-State=P10

Figure 2.13 CPU Utilization Metric Breakdown

Figure 2.14 illustrates the P-state selection mechanism on a 4-physical core processor utilizing the *ondemand* governor. We assume that core 0 has utilization of 100% in the given interval; thus, it is mapped to the P0 state. Similarly, core 1 with the utilization of 80% is mapped to P2, core 2 with the utilization of 0% to P11, and core 3 with the utilization of 60% to P4. If the processor supports core level P-state management, then cores 0-3 operate in states P0, P2, P10, and P4, respectively. However, if the processor only supports socket level P-state management, then the lowest-numbered P-state among all the cores is selected for the entire processor (P0 in our example). This request is then sent to the P-unit through the corresponding driver.

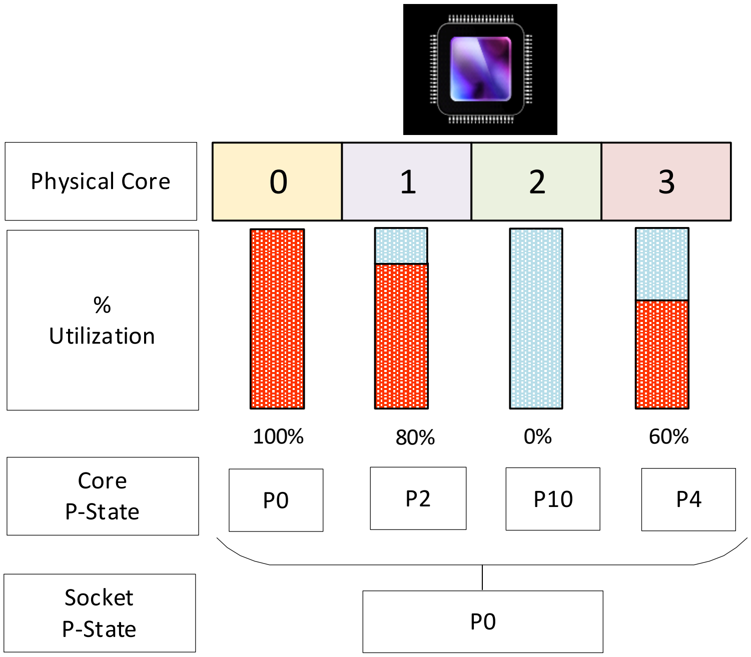


Figure 2.14 Core-wise P-state Voting Mechanism

CHAPTER 3

MOTIVATION

The current consensus reflected in the implementation of the most frequently used governors is that running the processor at the highest possible clock frequency during program execution is the most energy-efficient strategy. However, several studies have shown that this approach is not optimal for all types of workloads, especially for those that are bound by memory [15] [27].

To illustrate the problems with the CPU utilization metric discussed in 2.6, let us consider an example illustrated in Figure 3.1. Assume a processor supports 11 P-states, P0-P10. A CPU is utilized for 9 ms by a thread out of 10 ms in a DVFS interval, resulting in a 90% utilization rate. Consequently, the ondemand governor selects the P1-state. However, the utilization metric does not look into whether the thread performs any useful computation or not.

For example, it could be that out of 9 ms, only 3 ms are spent in doing useful computation. The rest are wasted processor clock cycles due to mispredictions in the processor front-end, structural hazards in the back-end, stalls due to memory reads and writes, or other stalls. This results in wasted CPU clock cycles that continue to consume energy without providing any returns. This problem is present in processors operating in all domains, from hand-held devices to datacenter servers. To the best of our knowledge, none of the current state-of-the-art governors deal with this issue even if the energy-saving settings are turned on.

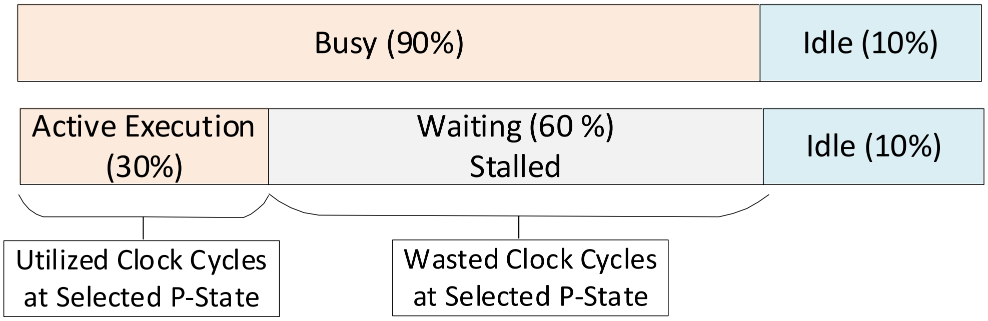

Busy (90%)
Idle (10%)
Active Execution (30%)
Waiting (60 %) Stalled
Idle (10%)
Utilized Clock Cycles at Selected P-State
Wasted Clock Cycles at Selected P-State

Figure 3.1 Limitations of the CPU utilization metric.

To quantify the impact of the voltage and frequency operating points on the execution time of different types of benchmarks, we consider three floating-point speed benchmarks from the SPEC CPU2017 benchmark suite: *638.imagick*, *628.pop2*, and *649.fotonik3d*. The benchmarks are picked from the SPEC CPU2017 floating-point speed suite where the user has the ability to select the number of OpenMP threads to run.

In this case, the benchmarks are run with 6 threads to fully load a test machine with 6 processor cores. These benchmarks exhibit different characteristics, being *compute-intensive* (*638.imagick*), *balanced* (*628.pop2*), and *memory-intensive* (*649.fotonik3d*) [26]. *Compute-intensive* refers to benchmarks that are bound by the available on-chip compute resources. *Balanced* benchmarks are bound by both the available compute resources and the memory subsystem, where performance depends on both compute resources, memory size, and bandwidth. *Memory-intensive* applications are bound by the memory subsystem, where performance is dependent on the available memory size and bandwidth alone.

Figure 3.2, Figure 3.3, and Figure 3.4 show the program execution time (primary *Y*-axis) and the total number of clock cycles needed (secondary *Y*-axis) as a function of statically selected operating points (frequency, voltage) across the entire socket for *638.imagick*, *628.pop2*, and *649.fotonik3d*, respectively. The number of clock cycles is further divided into clock cycles that actively issue a micro-operation and clock cycles that are stalls.

In the case of *638.imagick* (Figure 3.2), the total number of clock cycles and the percentage of the stalled cycles remain constant, regardless of the clock frequency. Consequently, the program execution time proportionally decreases as the clock frequency increases. In the case of *628.pop2* (Figure 3.3), the total number of clock cycles needed to execute the benchmark increases with an increase in the clock frequency. This increase is mainly driven by a significant increase in the number of stalled cycles caused by memory. Consequently, the program execution times plateaus at ~2.7 GHz.

Finally, in the case of *649.fotonik3d* (Figure 3.4), the total number of clock cycles increases almost 4-fold as the clock frequency increases from 0.8 GHz to 4.3 GHz. Here the program execution time plateaus at ~1.7 GHz. Thus, processors running at higher clock frequency will waste energy without any benefit to overall performance. Yet, the default *ondemand* governor would run all three benchmarks at the maximum clock frequency.

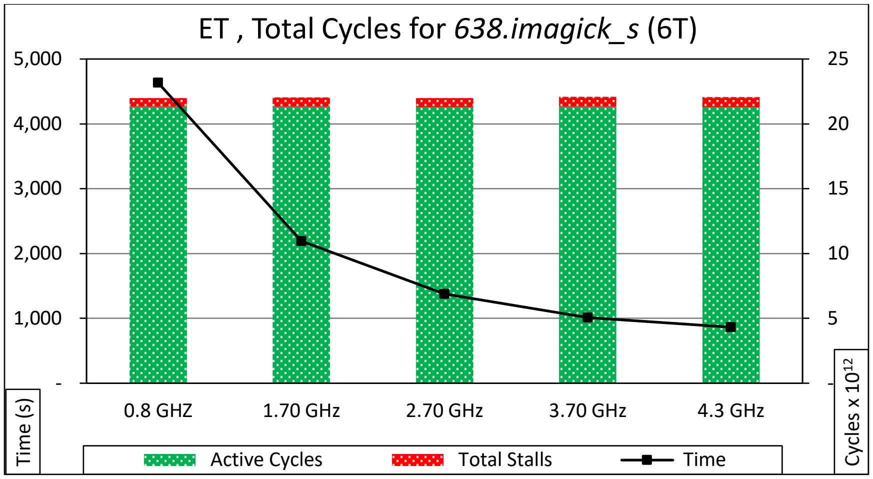


Figure 3.2 Impact of Frequency Scaling on Compute Intensive Benchmark

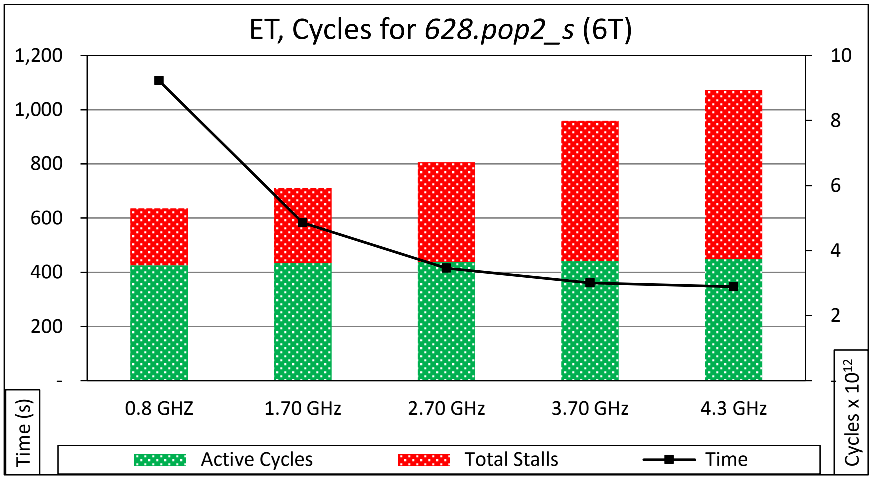


Figure 3.3 Impact of Frequency Scaling on Balanced Benchmark

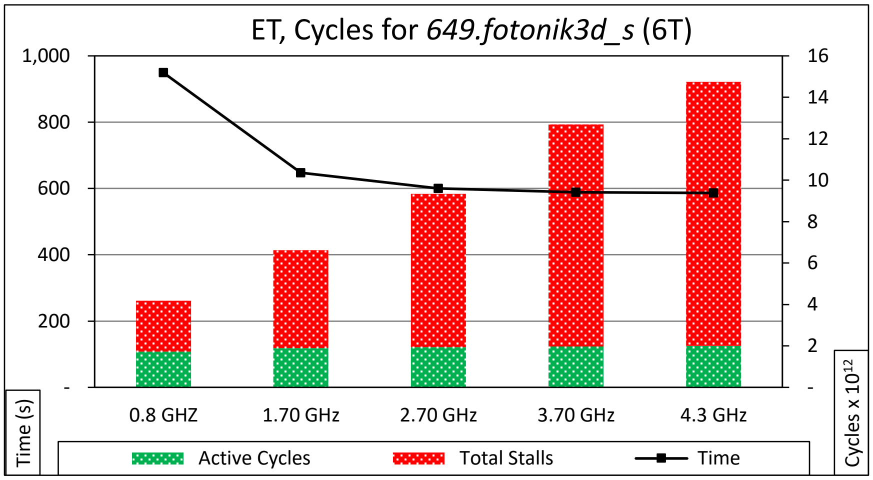


Figure 3.4 Impact of Frequency Scaling on Memory Intensive Benchmark

To remedy this problem, we introduce a new class of DVFS-based governors that do not use processor utilization as the primary metric in selecting P-states. Rather, we propose considering a range of different events from performance monitoring units that can help us dynamically select P-states that will reduce energy consumption while providing minimal performance degradation.

CHAPTER 4

PMU-EVENTS-DRIVEN DVFS TECHNIQUES

This section describes the proposed PMU-event-driven DVFS techniques and their implementation. Section 4.1 introduces the performance monitoring unit (PMU) and the top-down microarchitectural analysis method (TMAM) derived from the PMU events. Section 4.2 describes our proposed techniques for runtime DVFS. We propose four techniques that use the metrics derived from the Performance Monitoring Unit (PMU) events to determine P-states. The first two techniques evaluate utilization at the microarchitectural level, by using pipeline stalls or total cycle stalls. The next two techniques focus on the memory subsystem by using the memory-related stalls or the last level misses per kilo instruction to determine P-states. Section 4.3 discusses the previously proposed CPI-based frequency scaling technique and its limitations. Section 4.4 details the implementation of the proposed techniques.

4.1 Performance Monitoring Unit Event-Based Analysis

Modern processors integrate multiple components on a single chip, including, out-of-order superscalar processor cores with private L1 and L2 caches, interconnect, shared L3 caches, hardware accelerators (e.g., GPGPU), and a memory controller.

Multiple micro-operations can be executed and retired concurrently in a single clock cycle (~5 in most modern x86 processors). Writing effective software that takes full advantage of complex hardware structures is a challenging proposition. To cope with this challenge, software developers often rely on dedicated on-chip hardware resources called performance monitoring units (PMUs). Performance monitoring was introduced in the Pentium processor with a set of model-specific counters.

PMUs can help software developers find bottlenecks in their programs, understand how their programs utilize available hardware resources and guide their optimization efforts. A PMU typically consists of several counters dedicated to counting various hardware and software-triggered events. Each processor core includes several fixed-purpose counters (e.g., counting clock cycles and instructions) and several programmable general-purpose counters. The programmable counters can be used to count one of the hundreds of available events. The events can be broadly classified into hardware events (e.g., cache misses, branch mispredictions) and software events, from the OS and kernel (e.g., page faults, context switches) [79]. Modern processors support uncore PMUs, those that reside outside processor cores and can count events related to the memory controller, interconnect, or shared L3 caches.

4.1.1 Top-down Microarchitectural Analysis Method

Modern superscalar processors can be conceptually divided into the front-end and the back-end. The front-end is responsible for fetching and decoding instructions into micro-operations for execution. The back-end is responsible for scheduling, execution, and retiring of instructions. The Top-down Microarchitectural Analysis Method (TMAM) introduced by A. Yasin provides a practical way to quickly identify true bottlenecks in Intel processors [65]. In this method, we assume that each CPU core on each clock cycle has a fixed number of pipeline slots available as shown in Figure 4.1. The TMAM analysis looks at the issue stage of the pipeline, which is right in between the front-end and the back-end. Therefore, in any instance, it is possible to determine the maximum number of pipeline slots that can be issued. In this example, a 4-wide CPU is shown executing instructions for 10 clock cycles, resulting in 40 *pipeline slots*.

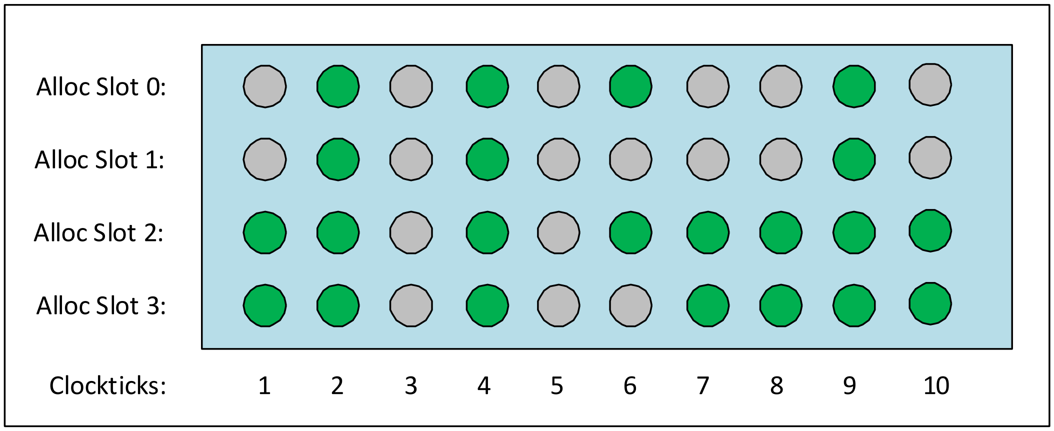


Figure 4.1 Illustration of pipelines slot utilization on a 4-wide CPU

If a pipeline slot retires a micro-operation, it is useful (shown in green) and if it does not retire a micro-operation it is attributed to a *stall* (shown in grey). Thus, in this example, 18 out of 40 slots are stalled, indicating that the code efficiency from the microarchitecture perspective is only 55% (22/40). An alternative form of evaluating code effectiveness is by observing the total cycle stalls. A particular CPU clock cycle is considered *a stall* when no micro-operation is issued across all available slots. From the illustration in Figure 4.1, 2 out of the 10 cycles are stalled. This indicates a clock cycle utilization of 80% (8/10).

The TMAM analysis breaks up all pipeline slots into four categories as shown in Figure 4.2: (i) Pipeline slots containing useful work that is issued and retired (*Retiring*); (ii) Pipeline slots containing useful work that is issued but flushed (*Bad Speculation*); (iii) Pipeline slots that could not be filled with useful work due to problems in the front-end such as limited buffer sizes and low decode bandwidth (*Front-End Bound*); and (iv) Pipeline slots that could not be filled with useful work due to unavailability of functional units and data hazards in the backend (*Back-End Bound*) [73].

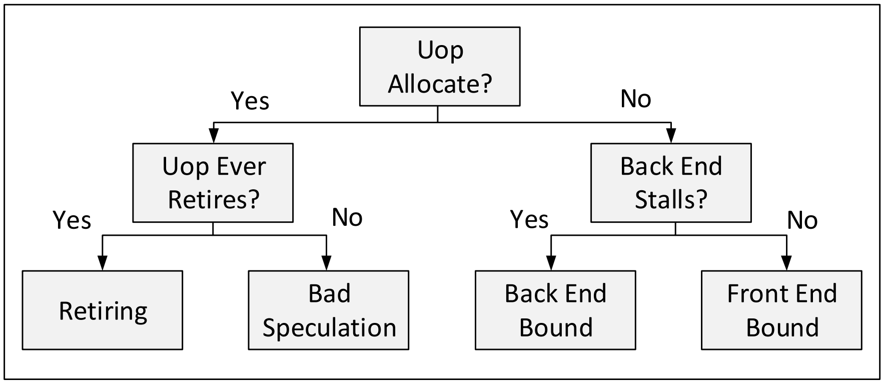


Figure 4.2 TMAM slot classification hierarchy

The *Retiring* metric represents a fraction of pipeline slots utilized by useful work, i.e., μOps (micro-operations) that eventually get retired. μOps perform basic operations on data stored in one or more registers, including transferring data and performing arithmetic or logical operations on registers. Ideally, all pipeline slots would be attributed to the *Retiring* category. *Retiring* of 100% would indicate that the maximum possible number of retired μOps per clock cycle has been achieved. Maximizing *Retiring* typically increases the *Instruction-Per-Cycle* (IPC) metric. A lower IPC indicates bottlenecks that should be addressed for better performance.

Bad Speculation captures a fraction of pipeline slots wasted due to incorrect speculations. This includes slots used to issue μOps that eventually do not get retired and slots for which the issue-pipeline was blocked due to recovery from earlier incorrect speculation.

The *Front-End Bound* metric captures a fraction of pipeline slots where the processor's front-end undersupplies its back-end. Within the front-end, a branch predictor predicts the next address to fetch, cache-lines are fetched from the memory subsystem, cache-lines are split into instructions, and lastly, instructions are decoded into micro-operations (μOps). The *Front-End Bound* metric denotes pipeline slots that are not utilized because the front-end failed to deliver μOps, even though the back-end could have accepted them.

The *Back-End Bound* metric captures a fraction of pipeline slots where no μOps are being delivered due to a lack of required resources in the back-end for accepting new μOps. The back-end is a portion of the processor core where an out-of-order scheduler dispatches ready μOps into their respective execution units, and, once completed, these μOps get retired according to program order. For example, stalls due to data-cache misses or stalls due to the divider unit being overloaded are both categorized as *Back-End Bound*. The *Back-End Bound* stalls are further broken down into two subcategories: (i) Core Bound stalls and (ii) Memory Bound stalls.

Core Bound stalls. *Core Bound* stalls are caused by a less-than-optimal use of the available execution units in the CPU. This metric captures the impact of stalls caused by a shortage of uncore resources or data dependencies. Hence it may indicate the CPU may have exhausted all the Out of Order (OOO) resources, certain execution units are overloaded or dependencies in the program's data- or front-end is limiting the performance (e.g., FP-chained long-latency arithmetic operations).

Memory Bound stalls: This metric shows how memory subsystem issues affect performance. *Memory Bound* captures a fraction of pipeline slots where pipelines are being stalled due to load or store instructions. This accounts mainly for incomplete in-flight memory demand loads in addition to less common cases where stores could imply back pressure on the pipeline.

4.2 Proposed DVFS Techniques

We propose four techniques that use the architectural events derived from the PMUs to determine P-states. PMUs in each core are programmed to count specific events for a given period of time. The first two techniques evaluate core utilization using microarchitectural metrics defined in 4.1, namely the pipeline slot stalls (*DVFS based on Pipeline Slot Stalls or FS-PS*) and the total cycle stalls (*DVFS based on Total-Stalls or FS-TS***)**. The next two techniques evaluate the utilization of the memory subsystem by using the memory-related cycle stalls (*DVFS based on Memory-Stalls or FS-MS*) and the last level cache misses per kilo instructions (*DVFS based on LLC Misses PKI or FS-LLCM*).

FS-PS: The first technique selects the P-state based on the pipeline slot stalls. The pipeline slot stalls are a metric that accurately captures the CPU's pipeline utilization. The number of available pipeline slots in a given time interval can be divided into (i) pipeline slots that issue micro-operations and (ii) stalled/unused pipeline slots, as shown in Figure 4.3. The pipeline slot stall ratio is computed as the number of unused slots divided by the total number of available slots in the time interval (Eq. 4.1). In the illustration from Figure 4.1, 18 out of 40 available slots do not issue any useful micro-operation, resulting in a pipeline stall ratio of 0.45.

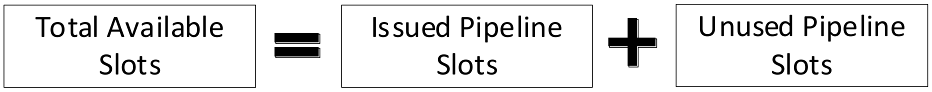


Figure 4.3 CPU Pipeline Slots Breakdown

$$\mathit{Pipeline\ Stall\ Ratio} = 1 - \frac{\mathit{Issued\ Pipeline\ slots}}{\mathit{Total\ Pipeline\ Slots}} \qquad \text{Eq. 4.1}$$

By profiling a range of representative workloads, we find that the pipeline slot stall ratio is always larger than 0.10. Hence, the ratio range between 0.1 and 1.0 is linearly mapped onto available P-states. While this metric accurately assesses the pipeline occupancy, it has one weakness. If the code does not have enough work to fill in all of the slots in a single cycle (e.g., due to data dependencies), the pipeline slot stall ratio will be relatively high, which will in turn lower the clock frequency. However, this may not be advantageous for either performance or energy efficiency. Figure 4.4 illustrates one such scenario where the pipeline slot stall ratio is 0.8.

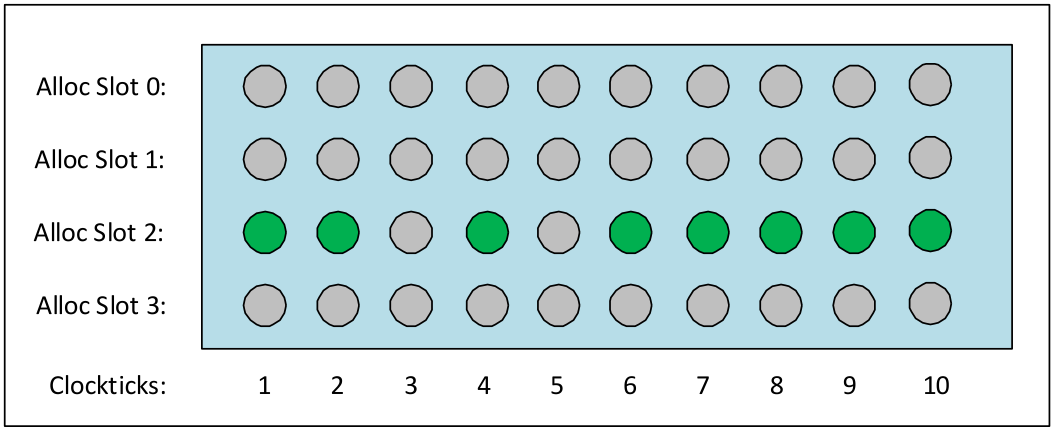


Figure 4.4 Pipeline slot occupancy resulting in a high pipeline slot stall ratio.

In this case, FS-PS will throttle towards a P state with a low clock frequency; this will, in turn, lower performance with no tangible benefits for energy efficiency. For example, this happens when stalls are caused by data loads satisfied by upper levels in the cache hierarchy.

FS-TS: The second technique selects the next P-state based on the total cycle stalls, promising to overcome the shortcomings of FS-PS. The processor clock cycles while executing instructions can be divided into (i) those that contain at least one pipeline slot that actively issues and retires a micro-operation and (ii) those that contain stalls across all available slots as shown in Figure 4.5. The total cycle stall ratio is computed as the number of unused cycles divided by the total number of CPU cycles in a given time interval as shown in Eq. 4.2. Thus, the total cycle stall ratio for a scenario shown in Figure 4.4 is 0.2 (2 out of 10 cycles are completely unused). Consequently, unlike FS-PS, FS-TS will ensure that the CPU runs at a relatively high clock frequency as long as there are not too many adjacent clock cycles without any useful micro-operations that can be issued.

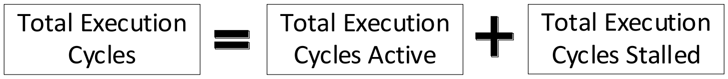


Figure 4.5 Total Execution Cycle Breakdown

$$Total\ Stall\ Ratio = \frac{Total\ Stalled\ Cycles}{Total\ Cycles} \qquad \text{Eq. 4.2}$$

FS-MS: The next proposed technique focuses only on the memory subsystem. It selects the next P-state based on the ratio of memory-related cycle stalls. The total cycle stalls in the back-end can be divided into (i) core-related cycle stalls and (ii) memory-related cycle stalls, as shown in Figure 4.6. The memory cycle stall ratio is computed as the number of cycles stalled due to the memory hierarchy divided by the total number of CPU cycles in a given time interval, as shown in Eq. 4.3. We observe through workload profiling that the memory-related cycle stall ratio is always lower than 0.90. Hence the ratio ranging from 0.0 to 0.9 is mapped linearly onto the available P-states.

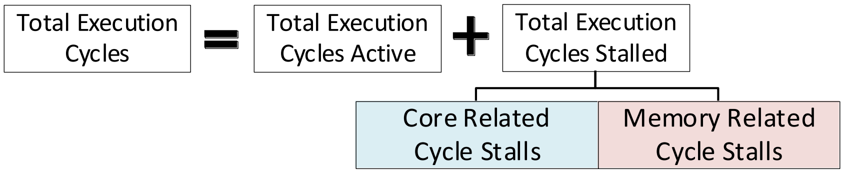

Total Execution Cycles
=
Total Execution Cycles Active
+
Total Execution Cycles Stalled
Core Related Cycle Stalls
Memory Related Cycle Stalls

Figure 4.6 Total Stall Cycle Breakdown

$$Memory\ Related\ Stall\ Ratio = \frac{Total\ Memory\ Related\ Stalled\ Cycles}{Total\ Cycles} \qquad \text{Eq. 4.3}$$

FS-LLCM: This technique utilizes the stalls in the memory hierarchy, specifically the ones caused due to off-chip requests. Figure 4.7 shows the breakdown for the memory-related stall cycles. The memory requests can be resolved in the upper levels of the cache hierarchy (e.g., LLC) or may require access to DRAM (off-chip). The number of stalls imposed by the requests resolved in DRAM can be orders of magnitude larger than the number of stalls imposed by the requests resolved in caches.

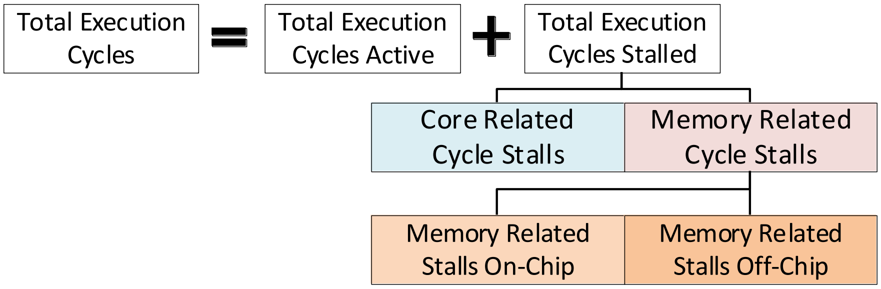

Total Execution Cycles
=
Total Execution Cycles Active
+
Total Execution Cycles Stalled
Core Related Cycle Stalls
Memory Related Cycle Stalls
Memory Related Stalls On-Chip
Memory Related Stalls Off-Chip

Figure 4.7 Memory Stall Breakdown

$$LLC\ MPKI = \frac{LLC\ Misses}{Retired\ Instructions} * 1000 \qquad \text{Eq. 4.4}$$

FS-LLCM is based on using the misses in the last level cache memory per kilo instructions (Eq. 4.4) to determine the next P-state. A miss in the last level cache correlates with an increased number of stall cycles. We observe through workload profiling that the LLC MPKI typically ranges from 0 to 100. The actual LLC-MPKI is then linearly mapped onto the available P-states.

4.3 DVFS based on CPI (FS-CPI)

Johnson et al. proposed a frequency scaling technique that uses cycles-per-instruction (CPI) to determine the next P-state [34]. The proposal uses PMU events *cycles* and *instructions* to determine the CPI of each active thread and groups them into low-CPI and high-CPI threads. Each of these groups is then scheduled onto different cores/sockets with different operating frequencies. For comparison with our proposed techniques, we implement a version of this proposal (*DVFS based on CPI or FS-CPI*). As specific implementation guidelines for CPI ranges and mappings are not specified, we define conditions similar to our techniques for a fair comparison. The CPI range depends on processor microarchitecture, the number, and characteristics of P-states, and the workload characteristics. In our case, the test system has 40 P-states. Through profiling various workloads, we observe that the CPI can be as high as 6.28. Hence, we select the CPI range from 0 to 6, before mapping it onto the available P-states. Experiments with other ranges are performed as well, but the results turned out to be inferior when compared to the selected range.

The CPI is a useful metric for assessing system performance. However, it could sometimes be misleading in modern superscalar processors. Modern processors support a number of vector instruction set extensions, with the most recent AVX2 that can process 512 bits of data in a single operation. Such instructions do significantly more work in a single clock cycle than corresponding scalar instructions. The use of vector instructions generally shortens the time needed to complete a task. However, since a single vector instruction does a lot of work, the CPI for a vectorized program typically exceeds the CPI of an equivalent scalar program. This phenomenon, where the non-vectorized code has lower CPI but poorer performance, has been observed in prior research [28]. It is better to use fewer vector instructions that do more work than to use many scalar instructions that retire faster [2][80]. This can be illustrated using a simple example. Figure 4.8 illustrates the scalar addition of two vectors with 64 elements, where each element is 1 byte in size. Assuming each scalar addition takes 1 cycle, 64 clock cycles are required to complete 64 operations, resulting in a CPI of 1. However, if we vectorize the same code as shown in Figure 4.9, the whole operation can be completed in one instruction which could take nearly two clock cycles, resulting in a CPI of 2. Though the CPI is higher for the vectorized code, it takes significantly less time. Thus, the CPI as a metric fails to account for such intricacies.

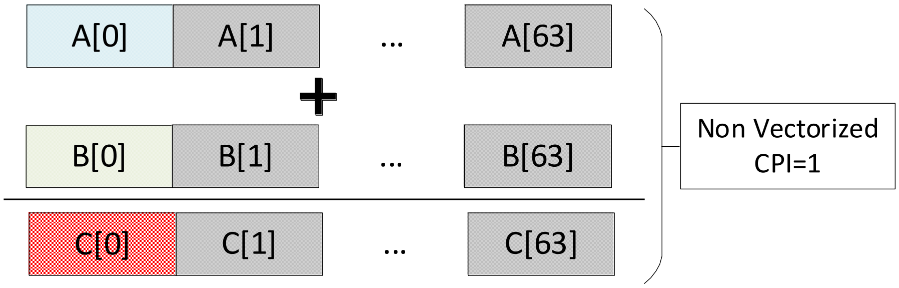


Figure 4.8 CPI of Non-Vectorized Code

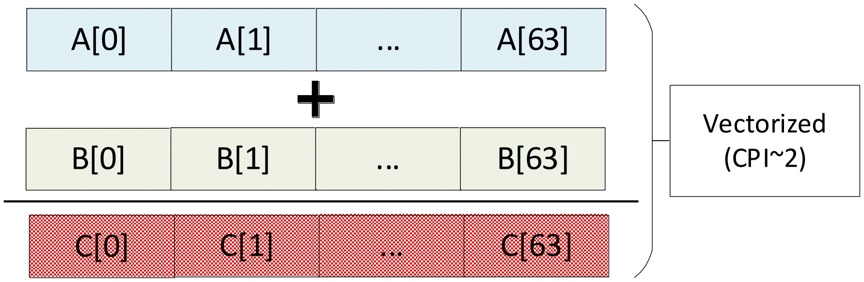


Figure 4.9 CPI of Vectorized Code

4.4 Implementation of the Proposed Techniques

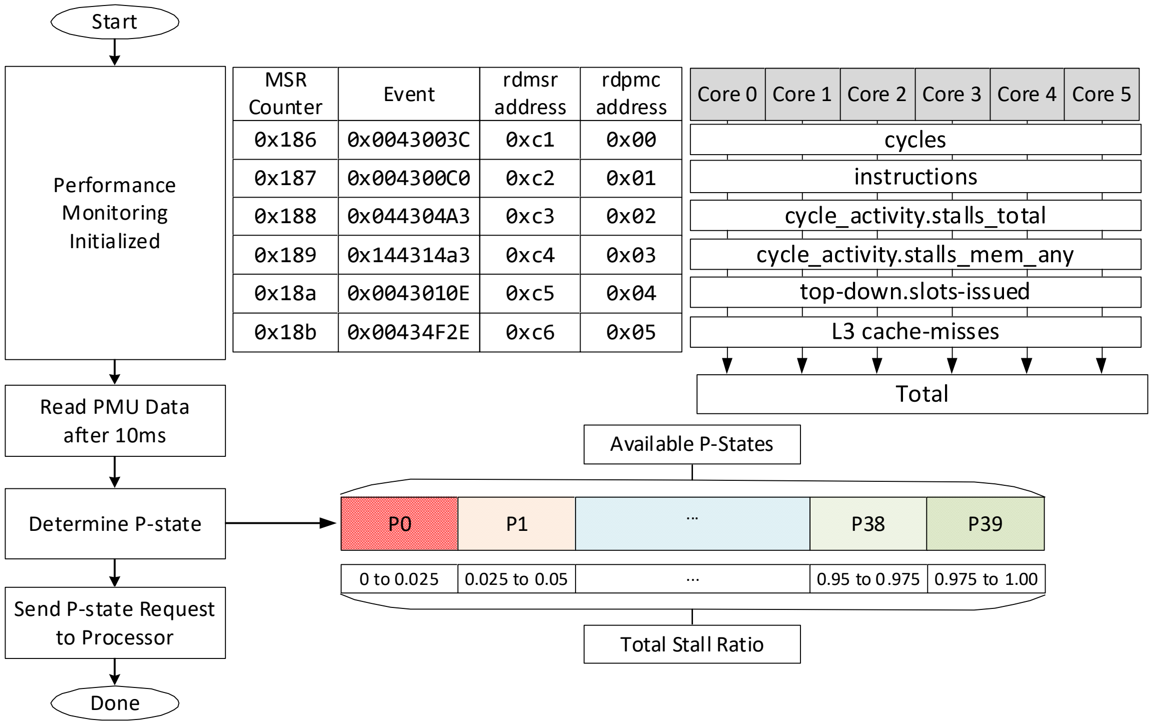
Figure 4.10 illustrates our implementation of the proposed techniques. All the proposed techniques use the same framework. They differ only in the events used to calculate the metrics of interest. The PMUs are initialized and programmed to count specific events in each physical core. The use of 'rdpmc' machine instruction reduces the latency to a few clock cycles when reading the PMU events. Events such as cycles, instructions, the total stall cycles, the total memory stall cycles, the total number of used pipeline slots, and the total number of L3 misses are counted using general-purpose counters. PMU events are collected concurrently across all physical CPU cores in the system.

Metrics such as the total pipeline slot stall ratio, the total cycle stall ratio, total memory-related cycle stall ratio, the LLC misses PKI, and the average CPI are computed periodically. The performance monitoring interval is set to 10 ms. This interval matches the period used by the current governors.

All the techniques employ a linear mapping of events onto the P-states, including P0 (turbo frequencies). The use of P0 ensures that *compute-intensive* benchmarks will not experience any performance degradation. The next P-state is determined for each processor core and then applied to individual cores if the core-level P-state management is supported. Alternatively, the lowest-numbered P-state is selected and applied to all the cores, if only the socket-level P-state management is supported, as shown in Figure 2.14. The implementation of the proposed algorithm has a worst-case execution time of ~13 ms (when running in the highest numbered P-state), 10 ms monitoring interval plus 3 ms to compute the metrics of interest, determine the next P-state and issue a request for the new P-state.

The implementation of this algorithm increases the CPU power consumption by 1 W during nominal operating conditions when processor cores are idle. We also note that the frequency of algorithm implementation is an important aspect. For the given workloads, which take significant time and do not change phase often, an invocation period of 100 ms provides good results. For workloads with frequent phase changes, a smaller invocation period, e.g., ~10ms, is beneficial. However, implementing the technique at the hardware level would provide the best possible results.

Next, it should be noted that though all the techniques are implemented on an Intel processor, similar PMU infrastructures exist in AMD and ARM processors. However, specific event names and access methods/tools may vary. Thus, the proposed techniques can be used in non-Intel architectures.



MSR Counter	Event	rdmsr address	rdpmc address
0x186	0x0043003C	0xc1	0x00
0x187	0x004300C0	0xc2	0x01
0x188	0x044304A3	0xc3	0x02
0x189	0x144314a3	0xc4	0x03
0x18a	0x0043010E	0xc5	0x04
0x18b	0x00434F2E	0xc6	0x05

Figure 4.10 Implementation of the Proposed DVFS Techniques

CHAPTER 5

EXPERIMENTAL ENVIRONMENT

This chapter gives an overview of the experimental environment, tools used for measurements, and metrics used for evaluation. Section 5.1 describes the test system used for experiments and experimental conditions. Section 5.2 describes various metrics used in the study. Section 5.3 covers all the tools used in the study. Finally, Section 5.4 introduces the workloads used in the study. All the measurements are carried out on the test system in the LaCASA Laboratory at UAH [81].

5.1 System under Test

The study primarily utilizes a workstation with an Intel x86 processor. The test system is built around an Intel 8th generation Core i7-8700K (code name Coffee-Lake) manufactured using Intel's 14nm++ technology node [82]. The processor core architecture is based on the Skylake architecture with minor updates and refinements. Figure 5.1 shows the die map of the processor used in this study. The processor includes six processor cores (hexa-core), a shared L3/LLC cache partitioned to ~2 MiB per core, a graphical processing unit, a memory controller, a system agent, and I/O interfaces, all connected through an on-chip ring interconnect.

The processor supports hyperthreading, thus providing twelve logical cores when hyperthreading is enabled. However, throughout the study, we disable hyperthreading for measurement purposes as hyperthreading does not contribute to the performance of SPEC CPU2017 benchmarks and several measurements require it to be disabled [23]. The integrated memory controller is in a dual-channel configuration with a maximum bandwidth of 41 GiB/s to external DRAM memory. The processor clock frequency ranges from 0.8 GHz to 4.3 GHz (Turbo Mode when all cores are active) or 4.7 GHz (when only one core is active). The nominal frequency is set to 4.3 GHz.

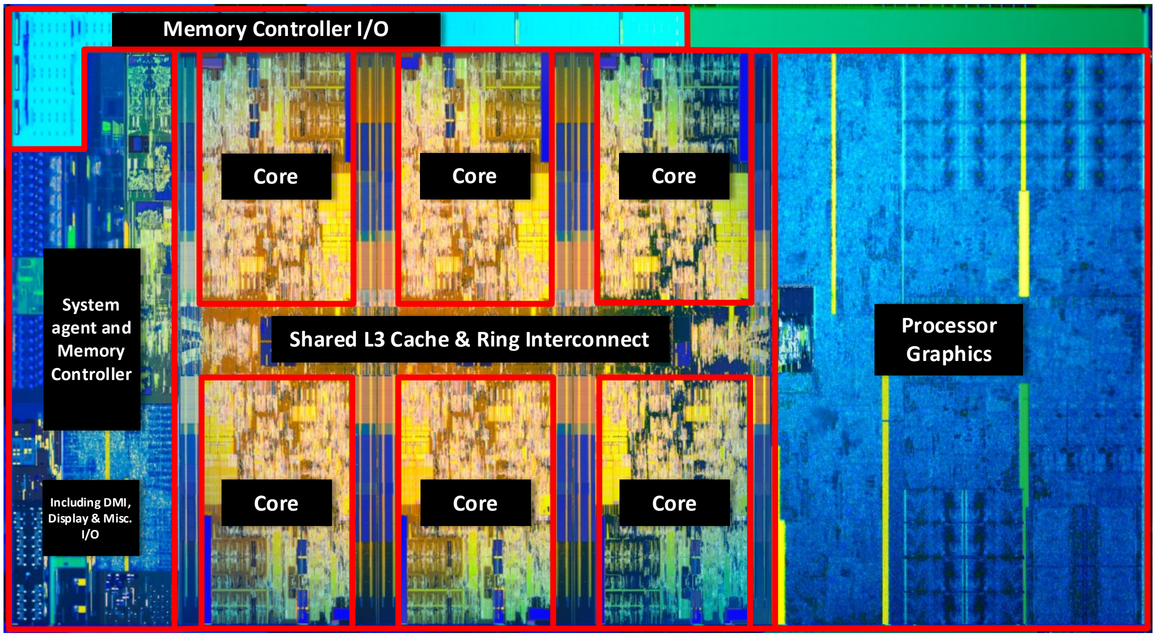

Memory Controller I/O
Core
Core
Core
System agent and Memory Controller
Shared L3 Cache & Ring Interconnect
Processor Graphics
Including DMI, Display & Misc. I/O
Core
Core
Core

Figure 5.1 Die Map of a Hexa-Core Coffee Lake Processor

Table 5.1 provides the workstation parameters. The workstation has a total system DRAM of 32 GiB configured as dual-channel. The system runs Ubuntu 18.04 LTS with Linux kernel 4.15.0. It has sufficient power and cooling requirements. The highest observed CPU operating temperature of 55° C and no thermal throttling is observed throughput the experimental evaluations. The processor's base operating frequency is 3.70 GHz and an all-core turbo frequency of 4.30 GHz. Note that the proposed techniques were tested in multiple x86 machines and provide similar results. The test machine shown here has state-of-the-art power management with 40 P-states and hence was chosen.

Table 5.1 Test System Parameters

Processor	Core i7-8700K
Lithography	14 nm
Intel Codename	Coffee-Lake
Physical Core Count	6
Logical Core Count	12
CPU Max Freq.	4.70 GHz
CPU Nom. Freq.	3.70 GHz
CPU Min Freq.	0.80 GHz
Number of P-States	40 (P0-P39)
DRAM	32 GB
DRAM Freq.	2,400 MHz
DRAM Bandwidth	41.6 GB/s (2-Channels)
TDP (watts)	95 W

5.2 Metrics for Evaluation

In this study, we evaluate the impact of the proposed techniques on performance (P) and energy efficiency (EE). The performance of a benchmark is defined as the reciprocal of its execution time. Energy-efficiency of a benchmark is defined as the reciprocal of the energy consumed to execute the benchmark. As we are evaluating the effectiveness of the proposed techniques relative to the state-of-the-art ondemand governor, a reference measurement set is established for each benchmark, B_i, by measuring its execution time, $T(Bi, OD_{GOV})$, and energy consumed, $E(Bi, OD_{GOV})$ when the *ondemand* governor is used.

To compare performance under different governors, we define performance speedup, $P.S$, calculated as shown in Eq. 5.1, where $T(Bi, PG_{GOV})$ is the execution time of a benchmark *Bi* when run using the proposed governor, *PG_{GOV}*. This metric captures the impact of a proposed governor on performance relative to the default *ondemand* governor. For example, a P.S of 0.5 indicates that the benchmark takes two times longer to execute under *PG_{GOV}* than under *OD_{GOV}*.

Similarly, we calculate the energy efficiency improvement *EE.I* for each benchmark, as shown in Eq. 5.2, where *E(Bi, PG_{GOV})* is the energy consumed by the benchmark Bi when run suing the proposed governor *PG_{GOV}*. Please note that both P.S and EE.I are a higher-is-better type of metrics.

$$P.S\ (B_i, PG_{GOV})\ = \frac{T(B_i, OD_{GOV})}{T(B_i,\ PG_{GOV})} \qquad \text{Eq. 5.1}$$

$$EE.I\ (Bi, PG_{GOV}) = \frac{E(B_i, OD_{GOV}\ )}{E(B_i, PG_{GOV})} \qquad \text{Eq. 5.2}$$

Whereas *P.S* and *EE.I* capture the effectiveness of the proposed techniques in regard to performance and energy efficiency, respectively, we use their product, *PxEE.I*, to capture their overall effectiveness in a single number. *PxEE.I* is defined as shown in Eq. 5.3 and it assumes that both performance and energy efficiency are equally important. This is also a higher-is-better metric, and it captures the overall effectiveness of the proposed techniques relative to the *ondemand* governor. Thus, if one cares only about performance, P.S should be used. If one cares only about energy efficiency, EE.I should be used. Finally, if one cares about both, PxEE.I metric should be used.

$$PxEE.I(B_i, PG_{GOV}) = \frac{T(B_i, OD_{GOV}) * E(B_i, OD_{GOV}\ )}{T(B_i, PG_{GOV}\ ) * E(B_i, PG_{GOV})} \qquad \text{Eq. 5.3}$$

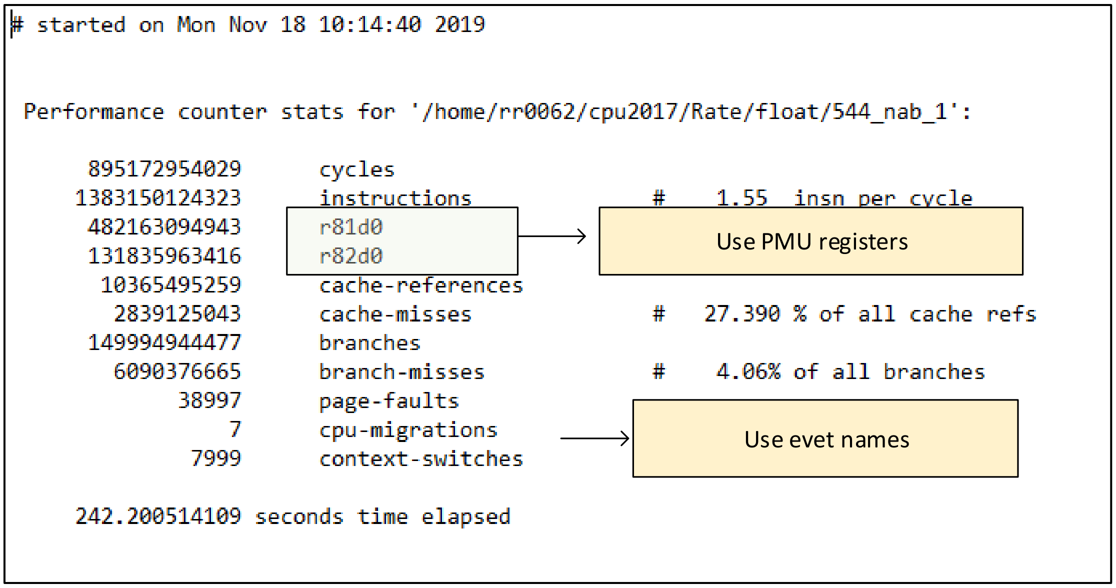
5.3 Tools

The study uses various tools in different sections of the study that primarily leverage the PMUs to collect various events during benchmark execution. Tools such as *Linux perf* [72] and *Intel VTune Amplifier* [73] are utilized to characterize and classify the workloads and *likwid* [60] is used to measure the program execution time and the processor power consumption during experimental evaluation.

5.3.1 Linux perf

Modern processors have dedicated hardware counters for performance monitoring as part of the PMU. They form a basis for profiling applications that trace dynamic control flow and identify hotspots. Linux *perf* is a profiler tool present in all Linux-based systems after kernel version 2.6. It abstracts the hardware differences in different processor generations and vendors by virtualizing the counter mechanism and providing a simple command-line interface with a list of measurable events. However, this process adds an overhead of about 100 ms for any measurement [63].

The tool and underlying kernel interface can measure events coming from PMUs, i.e., their hardware counters, or from the kernel. Some examples of micro-architectural events are, the number of clock cycles, instructions retired, L1 cache misses, and so on as shown in Figure 5.2. They vary with each processor type and model. Other events are counted using Linux kernel counters, and they are thus called software events. *Perf* has been consistently used in several performance evaluation studies for architectural evaluation and code optimization [16]. The study uses *perf* for characterizing the workload and for verifying the implementation of the proposed techniques.



```
# started on Mon Nov 18 10:14:40 2019


 Performance counter stats for '/home/rr0062/cpu2017/Rate/float/544_nab_1':

      895172954029      cycles
     1383150124323      instructions              #    1.55  insn per cycle
      482163094943      r81d0
      131835963416      r82d0
       10365495259      cache-references
        2839125043      cache-misses              #   27.390 % of all cache refs
      149994944477      branches
        6090376665      branch-misses             #    4.06% of all branches
             38997      page-faults
                 7      cpu-migrations
              7999      context-switches

     242.200514109 seconds time elapsed
```

Figure 5.2 Illustration of event access in *perf*

5.3.2 Likwid

Energy measurements are conducted using a set of lightweight command-line tools called *likwid* (Like I Knew What I'm Doing) [60]. It is targeted towards performance-oriented programming in a Linux environment, does not require any kernel patching, and is suitable for Intel and AMD processor architectures. The tools can be roughly grouped into three categories, such as system information and control, performance and energy profiling, and micro-benchmarking. Individual tools allow developers to explore memory hierarchy, access performance monitoring counters, control clock frequencies, and control architectural features, e.g., hardware prefetching. Specifically, we use *likwid-powermeter*, a tool that accesses RAPL counters for measuring power and energy [64], and *likwid-setfrequencies*, a tool that allows for setting the core and uncore clock frequencies. We use likwid for time and energy measurements.

5.3.3 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: (i) Advanced Hotspots, (ii) HPC Performance Characterization, (iii) Memory Access Analysis, and (iv) General Exploration. When the number of events exceeds the number of available PMU counters, the tool multiplexes events and uses sampling. Depending on the number of multiplexed events, the reliability of measurements can vary. If the reliability is less than 70%, then the results are not to be considered acceptable [73]. Our experimental results had a measurement reliability of over 95%.

Advanced Hotspot analysis is used 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 the instrumentation of memory allocations/de-allocations and getting static/global variables from the symbol information. Memory Access analysis 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).

5.4 Workloads

This section introduces the various workloads used in the dissertation. The study primarily uses the SPEC CPU2017 benchmark suite [83] to evaluate the proposed techniques. In addition, the study uses the Parsec-3.0 benchmark suite [84] representing now a bit aged workload, and the SPECpower_ssj2008 benchmark [85] to represent transactional workloads in servers.

5.4.1 SPEC CPU2017

The SPEC CPU suites have been widely used in academia and industry for the past few decades to evaluate the performance of processors, memory, and compilers [29]. The SPEC CPU2017 benchmark suites are the SPEC's latest, sixth generation of CPU benchmarks. The CPU2017 suites incorporate major updates relative to the previous generation, CPU2006. They include significantly larger workloads signifying the evolution of computing capacity [25], parallel programs using OpenMP to accommodate multiple core and thread models, and optional metrics for measuring power consumption [30].

The SPEC CPU2017 contains 43 benchmarks, organized into four suites [83] [8]. The *fp_speed/fp_rate* and *int_speed/int_rate* suites (shown in Table 5.2 and Table 5.3) include benchmarks with predominantly floating-point and integer data types, respectively, designed to stress the speed (speed suites) and throughput (rate suites) of modern computer systems. 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 compilation flags, run rules, and the size of the input workloads. Generally, speed benchmarks require more memory than their rate counterparts. The SPECspeed benchmarks need large stacks [23].

Table 5.2: SPEC CPU Floating-point Benchmarks

SPECrate	SPECspeed	Lang.	Application Area
503.bwaves	603.bwaves_s	Fortran	Computational Fluid Dynamics
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	Physics: General Relativity, Numerical Relativity
508.namd_r		C++	Scientific, Structural Biology, Molecular Dynamics
510.parest_r		C++	A finite element solver
511.povray_r		C++, C	Computer Visualization: Ray tracing-
519.lbm_r	619.lbm_s	C	Computational Fluid Dynamics
521.wrf_r	621.wrf_s	Fortran, C	Weather Research and Forecasting
526.blender_r		C++, C	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	Atmosphere General Circulation Model (AGCM)
	628.pop2_s	Fortran, C	Climate modeling: Wide-scale ocean modeling
538.imagick_r	638.imagick_s	C	Image manipulation
544.nab_r	644.nab_s	C	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	Computational Electromagnetics (CEM)
554.roms_r	654.roms_s	Fortran	Regional Ocean Modeling System (ROMS)

Table 5.3: SPEC CPU2017 Integer Benchmark

SPECrate	SPECspeed	Lang.	Application Area
500.perlbench_r	600.perlbench_s	C	Programming language: Perl interpreter
502.gcc_r	602.gcc_s	C	C Language optimizing compiler: GNU C compiler
505.mcf_r	605.mcf_s	C	Combinatorial optimization
520.omnetpp_r	620.omnetpp_s	C++	Discrete Event simulation
523.xalancbmk_r	623.xalancbmk_s	C++	XSLT processor for transforming
525.x264_r	625.x264_s	C	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	Artificial Intelligence: Alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	Artificial Intelligence (Monte Carlo simulation)
548.exchange2_r	648.exchange2_s	Fortran	Artificial Intelligence: Recursive solution generator
557.xz_r	657.xz_s	C	General data compression

The benchmarks are derived from a wide variety of application domains and are written in C, C++, and Fortran programming languages. The SPEC CPU2017 provides a comparative measure of integer and/or floating-point compute-intensive performance on a machine. Upon completion of execution, the user is provided with a number generated by the 'runcpu' utility that compares the performance to the SPEC reference machine [8]. This is convenient for quick analysis and a good starting point.

A single copy of a speed benchmark (name ending with a suffix "_s"), *SBi*, is run on a test machine using the reference input set; the *SPECspeed (SBi)* metric reported by the running script is calculated as the ratio of the benchmark execution times on the reference machine and the test machine as shown in Eq. 5.4.

$$SPECspeed(SB_i, N_T) = \frac{T(Ref, 1)}{T(Test, N_T)} \qquad \text{Eq. 5.4}$$

A composite single number is also reported for an entire suite; it is calculated as the geometric mean of the individual SPECspeed ratios of all benchmarks in that suite. When running speed benchmarks, a performance analyst has an option to specify the number of OpenMP threads, N_T, as many benchmarks support multi-threaded execution. Multiple copies (N_C) of a rate benchmark (name ending with a suffix "_r"), *RBi*, are typically run on a test machine, and the *SPECrate (RBi,* N_C*)* metric is defined as the ratio of the execution times of a single copy on the reference machine and N_C-copy on the test machine, multiplied by the number of copies as shown in Eq. 5.5.

$$SPECrate(RB_i, N_C) = \frac{N * T(Ref, 1)}{T(Test, N_C)} \qquad \text{Eq. 5.5}$$

5.4.2 Parsec 3.0

The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite composed of multithreaded programs. The suite was designed to be representative of shared-memory programs for chip-multiprocessors.

It consists of 9 applications and 3 kernels which were chosen from a wide range of application domains. The workloads were selected to include different combinations of parallel models, machine requirements, and runtime behaviors. The benchmarks cover a wide range of computer tasks such as financial analysis, computer vision, engineering, enterprise storage, animation, similarity search, data mining, machine learning, and media processing. Benchmarks vary in type of parallelization model (data-parallel or pipelined), working set, and communication intensity.

All benchmarks are written in C/C++. Characterization studies have evaluated the use of Parsec-3.0 benchmarks and have analyzed the parallelization, the working sets and locality, the communication-to-computation ratio, and the off-chip bandwidth requirements of its workloads [7] [11]. Several prior studies have used the Parsec suite simulations and experimental evaluations.

5.4.3 SPECpower_ssj2008

SPECpower_ssj2008 is an industry-standard benchmark designed for experimental power and performance evaluation of server computers. The workload is scalable, multi-threaded, and portable across a wide range of operating environments. It exercises CPUs, caches, memory hierarchy, and the scalability of symmetric multiprocessor systems (SMPs), as well as implementations of the Java Virtual Machine (JVM), Just-In-Time (JIT) compiler, garbage collection, threads, and some aspects of the operating system. Although the workload is derived from the SPECjbb2005 benchmark suite, the two workloads are not comparable because of basic differences in the transaction mix, transaction scheduling, and timing.

The execution of the benchmark consists of two phases, (a) calibration and (b) running of a series of target loads. Initially, a series of calibration measurements are performed to find the maximum throughput of the server. The calibration run, by default, uses three intervals of 240 seconds each. The benchmark uses a system call to determine the number of logical cores available on the system and creates a matching number of emulated warehouses. Transactions are scheduled in batches of 1000 transactions per warehouse. Once a batch of transactions is processed, the next batch is issued after a period of time, thus modulating the machine load. However, during the calibration, the bathes are issued continuously to determine the maximum throughput the machine can sustain – it is equivalent to the 100% load level.

After calibration, the benchmark run consists of a sequence of eleven load levels from 100% to 0% (idle) in 10% increments. The whole benchmark run takes about 70 minutes to complete on the test machine. The results include the total number of ssj operations for each load level and the corresponding power consumption and operating temperature. To compute a power-performance metric across all load levels, the measured transaction throughputs for each load level are added together and then divided by the sum of the average power consumed for each level. The result is a figure of merit called "overall ssj_ops/watt." This ratio indicates the effectiveness of the system under test and its energy efficiency.

CHAPTER 6

SPEC CPU2017 CHARACTERIZATION ANALYSIS

The SPEC CPU2017 benchmark suites are used as the primary workload in this study. To gain deeper insights on the impact of dynamic voltage and frequency scaling (DVFS) a comprehensive evaluation is conducted on the test system. During the SPEC CPU2017 evaluation, the processor is set to the fixed nominal operating frequency of 3.7 GHz. Section 6.1 discusses briefly the results of a compiler evaluation performed to select the primary compiler for the study [28]. Section 6.2 shows the results of the top-down microarchitectural analysis used to classify the benchmarks into specific groups [25] [26]. Finally, Section 6.3 shows the impact of static frequency scaling on different classes of benchmarks [27].

6.1 Compiler Evaluation

Modern compilers are extremely complex software that translates programs written in high-level languages into binaries that execute on the underlying hardware. Compilers play a key role in bridging the gap between abstract high-level source code used by software developers and the advanced hardware structures. The selection of a compiler depends on parameters such as accessibility, support for hardware, the efficiency of the compiler, and backward compatibility.

To select the compiler to build the CPU2017 suites, we consider the three most prevalent compilers used in industry and academia, as follows: (a) the Intel Parallel Studio XE-18 (IPS), (b) the LLVM Compiler Infrastructure project, and (c) the GNU Compiler Collection [28]. We evaluate their effectiveness by comparing three important metrics, as follows: (a) the total time needed to compile benchmarks (build times), (b) the size of the executables (code sizes), and (c) the execution times for *speed* benchmarks and throughput for *rate* benchmarks (performance). Note that we were unable to successfully compile and run all benchmarks across all compilers. The discussion for the entire benchmark suite contains only the benchmarks that have results across all the compilers. Any benchmark that does not have results across all compilers is omitted from the discussion and summary view of the suite.

6.1.1 Executable Size

Table 6.1 shows the size of the SPEC CPU2017 suites in terms of kilo lines of code (KLOC) and the total executable size generated by the three compilers. The size of the benchmark executables varies widely for different compilers.

Table 6.1: Executable Size (Lower is Better)

Suites	*KLOC*	*Executable size [KB]*		
		ips	*llvm*	*gnu*
fp_speed	916	22,868	16,952	46,346
fp_rate	3048	56,595	50,595	282,627
int_speed	2484	35,265	24,684	181,820
int_rate	2484	35,409	24,701	181,702
Total	8968	150,137	116,192	692,495

The LLVM compilers consistently create the smallest executables for all the suites. GNU compilers produce the largest executables in all cases. LLVM generates executables that are ~1.28x and ~5.92x smaller than the corresponding ones created by IPS and GNU, respectively.

6.1.2 Build Times

SPEC CPU2017 configuration files that govern the process of compiling benchmarks (compiler and libraries used, optimization switches, and others) allow us to specify the number of processor cores that can be utilized during compilation. Thus, we consider build times for all the benchmarks when using one processor and when using six processor cores.

The build times are shown in Table 6.2. LLVM has a smaller executable size, but it has significantly longer build-times in comparison to GNU and IPS. This is especially true for floating-point benchmarks. Though the GNU compilers produce executables that are significantly larger in size, the build times are shorter than the build times of LLVM. IPS on the other hand produces executables as small as LLVM and it does that in build times that are comparable to the GNU build times. The number of CPUs used in building the benchmarks plays a significant role in build times for GNU and LLVM, however, the IPS does not appear to benefit much when using multiple cores in the building process.

While considering build times, the GNU compiler collection is the best choice. Overall, when a single processor is used to build executables, GNU build times are ~11.16x shorter than build times of LLVM and ~1.03x shorter than IPS. When six processor cores are used to build benchmarks, GNU build times are ~10.34x (LLVM) and 4.41x (IPS) shorter.

Table 6.2: Build Times (Lower is Better)

Suits	*Build Time (1-CPU) [s]*			*Build Time (6-CPU) [s]*		
	ips	*llvm*	*gnu*	*ips*	*llvm*	*gnu*
fp_speed	490	2,911	251	426	775	79
fp_rate	880	11,203	940	534	2,521	224
int_speed	635	7,151	677	855	1,477	151
int_rate	608	7,157	678	852	1,480	151
Total	2,613	28,422	2,546	2,667	6,253	605

6.1.3 Performance

Finally, we look at the overall performance of executables created by each of the compilers. Table 6.3 shows the overall SPECratio calculated for the entire suite. Regarding benchmark performance, IPS is the clear winner with its ability to exploit hardware features of the x86 ISA. Considering the geometric mean of the SPEC ratios of all the benchmarks, we find that IPS executables run ~37% faster than LLVM and ~46% faster than GNU executables for single-threaded executions. When we consider six-threaded executions, IPS executables run ~22% faster than the corresponding ones for LLVM and ~30% faster than the GNU executables.

Table 6.3: SPECratios (Higher is Better)

Suits	*Performance (1-T/C)*			*Performance (6-T/C)*		
	ips	*llvm*	*gnu*	*ips*	*llvm*	*gnu*
fp_speed	10.05	7.54	6.01	20.04	18.98	15.40
fp_rate	11.79	7.91	7.15	34.00	28.00	27.00
int_speed	8.08	6.06	6.28	9.00	7.00	7.00
int_rate	6.95	5.28	5.46	31.00	25.00	25.00

In summary, executables created by IPS outperform those created by LLVM and GNU for all benchmarks. The performance of LLVM and GNU are comparable with LLVM doing better for floating-point benchmarks and GNU showing slightly better performance for the integer benchmarks. We observe that a vast majority of the submitted results in SPEC also use IPS results. Hence the rest of the study uses IPS compiled executables exclusively.

6.2 TMAM Results of SPEC CPU2017 Benchmarks

Now we use the to-down microarchitectural analysis method (TMAM) to classify the IPS compiled executables on the workstation. Figure 6.1 shows the results of TMAM for all the CPU2017 benchmarks executed with six thread/copies ($N_{T/}N_{C}$=6) as well as the average instruction per cycle (IPC) on the secondary *y*-axis. With TMAM, the product of the number of pipeline slots and the number of clock cycles needed to execute a benchmark constitutes 100% of possible pipeline slots. Each pipeline slot is then marked as either *Retiring* (orange), *Bad Speculation* (gray), *Front-End Bound* (yellow), or *Back-End Bound* stalls. The *Back-End Bound* stalls are further broken down into (i) *Core Bound* stalls (royal blue) that are caused by pressures on execution units or lack of instruction-level parallelism, and (ii) *Memory Bound* stalls (light blue) that are caused by stalls related to caches and memory subsystems. Memory latency and limited memory bandwidth are major factors contributing to a large number of *Memory Bound* slots.

The benchmarks are organized based on the overall IPC and the percentage of slots bound by the back-end, especially the memory sub-component. Observing the runtime behavior and resource requirements for each of the benchmarks, they can be classified as *compute-intensive*, *balanced*, and *memory-intensive* [26] [25]. The first group which is *compute-intensive* has a higher percentage of retiring slots. The bottlenecks are generally associated with the front-end and are generally core bound in the back-end. They have a low dependence on the memory sub-component. This results in a high IPC. Such benchmarks linearly scale with operating core-frequency and do not see noticeable benefits in lower operating frequency [27]. The second group called *balanced* is bound by both the front-end and the back-end. Such application has a lower percentage of retiring instructions resulting in a lower IPC. The benefits of frequency scaling for such benchmarks are contingent on where the bottleneck originates. If a significant number of stalls are resolved on-chip (e.g., data is found in the upper-level cache), lowering CPU clock frequency due to a high stall ratio would have a negative impact. On the other side, if a significant number of stalls is resolved off-chip, i.e., in DRAM, the lower CPU clock frequency may be beneficial. Finally, the last group is called *memory-intensive.* This group has a large dependency on the memory hierarchy resulting in an extremely low IPC. A significant portion of the pipeline slots are stalls.

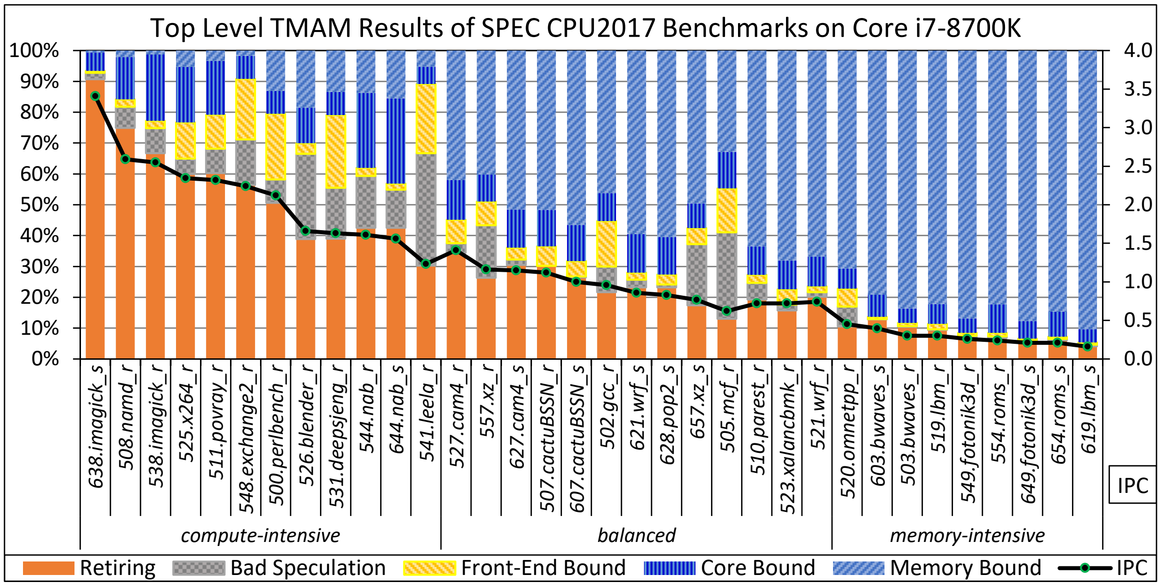


Figure 6.1 TMAM Analysis of SPEC CPU2017 Benchmarks

As the proposed DVFS techniques are geared towards *memory-intensive* applications with a significant number of stalls, it is vital to understand the breakdown of the memory bottlenecks. To address this, we consider the breakdown of benchmark execution using a clock cycles view. A clock cycle is considered stalled when no micro-operation is issued during that cycle across all slots. The origins of these stalls can be further divided into L1 bound, L2 bound, LLC/L3 bound, DRAM bound, and store bound. The L1-bound metric shows how often the execution was stalled without missing the L1 data cache. The L2 and L3 bound metric shows how often the core was stalled in L2 and L3 respectively. The DRAM bound metric shows how often the CPU was stalled in the main memory. The Store Bound metric shows how often the CPU was stalled on store operation.

Figure 6.3 shows the clock cycle view for all the CPU2017 benchmarks. As discussed earlier, the memory dependency of the *compute-intensive* group is minimal. Only a small fraction of the total execution cycles are memory hierarchy stalls. Next, in the case of the *balanced* benchmarks, the stalls in the memory hierarchy significantly increase, especially in DRAM. ~50% of all execution cycles are stalls in the memory hierarchy for the *balanced* benchmarks. It is also important to note that *523.xalancbmk_r* has a significant portion of stalls that are serviced in the last level cache. Finally, the *memory-intensive* benchmarks have a significant portion of the execution clock cycles being memory hierarchy stalls. Over ~80% of execution cycles are spent waiting for data to arrive. We observe that a major portion of the stall comes from either L1 or DRAM.

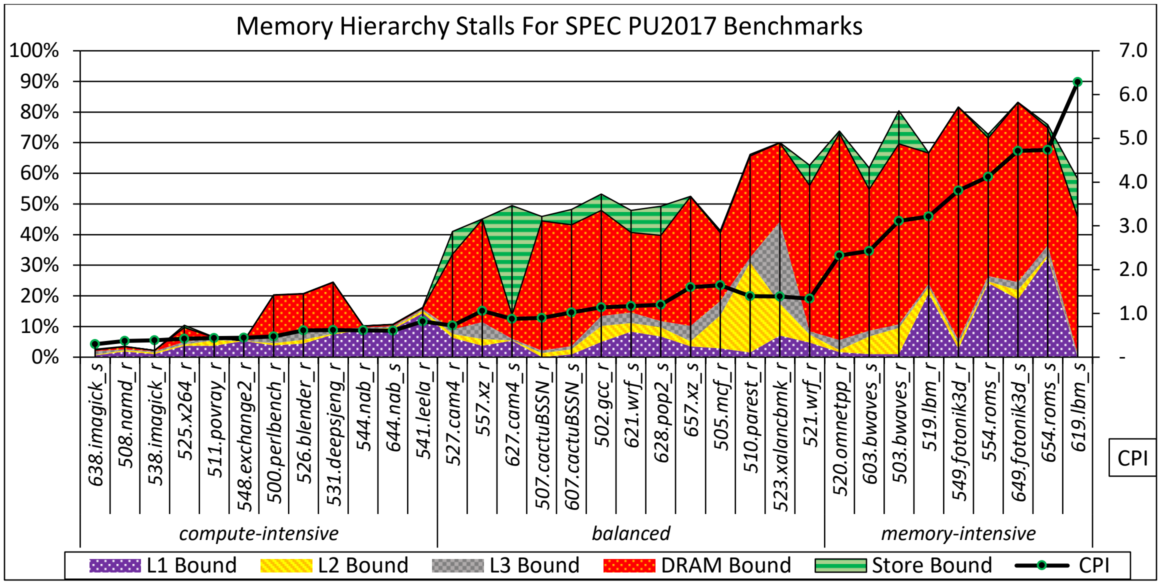


Figure 6.2 Memory Hierarchy Related Cycle Stall Breakdown

The DRAM-bound metric further enables us to identify bandwidth and latency-related issues in main memory. DRAM bandwidth is the rate at which data can be read from or stored into the main memory by the processor. DRAM bandwidth bound metric specifies the number of cycle stalls due to the inability of main memory bandwidth. Figure 6.3 explores the off-chip stalls by providing stalls that occur due to memory bandwidth and latency in DRAM. Overall, memory bandwidth is the biggest factor affecting the performance of these benchmarks.

In the case of the *compute-intensive* benchmarks, as expected we see minimal stalls from the main memory. The available bandwidth is not a constraint on the performance of such benchmarks. Next, in the case of the *balanced* benchmarks, the bandwidth-related stalls account for ~20% of the execution cycles. Considering the *memory-intensive* group of benchmarks, ~50% of all execution cycles are stalls in DRAM due to bandwidth limitations. This is concerning when we notice that over the past decade, that single-core memory bandwidth allocation has not improved significantly. This is a major bottleneck in modern systems.

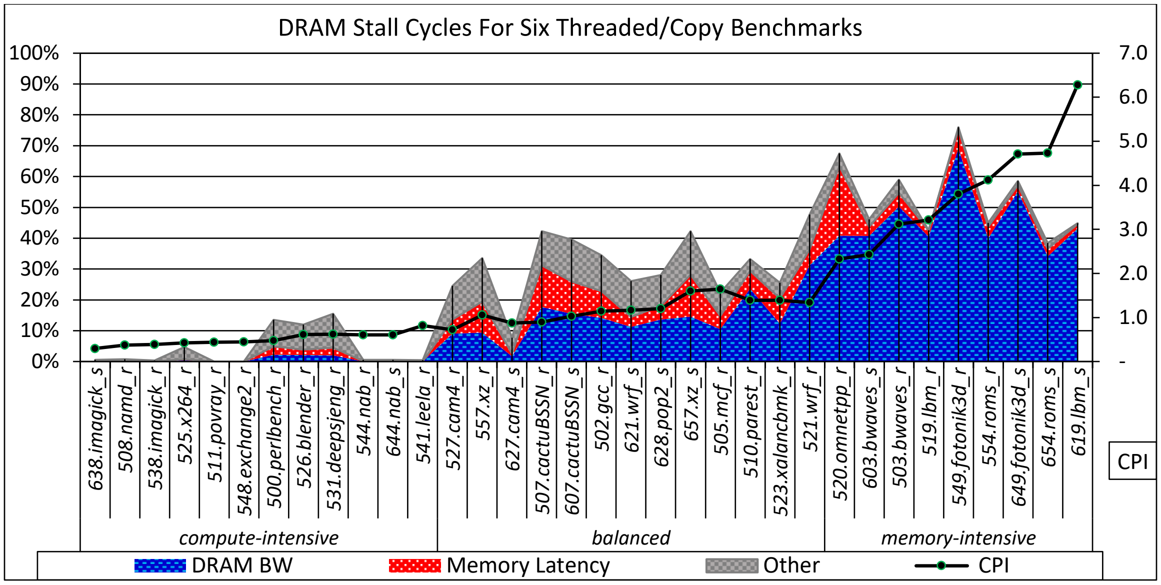


Figure 6.3 Main Memory/Off-Chip Stall Breakdown

Further, to demonstrate the bandwidth utilization of the benchmarks we show the main-memory bandwidth consumption for each of the benchmarks. Figure 6.4 shows the average and the maximum DRAM bandwidth consumed at any point during the execution of each of the SPEC CPU2017 benchmarks.

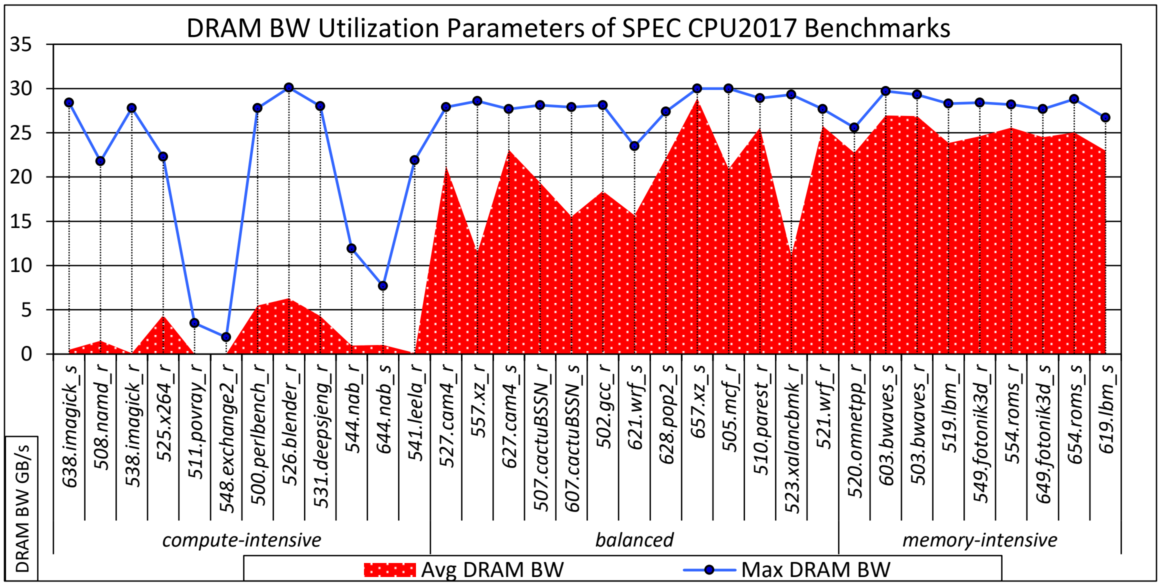


Figure 6.4 DRAM Bandwidth Utilization Parameters

Overall, we see that the *compute-intensive* benchmarks have a low average DRAM bandwidth utilization. The *balanced* benchmarks have a moderate bandwidth demand, well under the machine capacity. Finally, the *memory-intensive* benchmarks consistently reach the full test machine capacity and as a result, have significant stalls in the DRAM as shown in the previous section.

In summary, based on the run-time analysis of the SPEC CPU2017 benchmarks are classified into three distinct groups as shown in Figure 6.5; (a) *compute-intensive* when the benchmark performance is generally compute-bound; (b) *balanced* when benchmarks are bound both by the compute and memory resources; and (c) *memory-intensive* when the benchmarks are heavily bound by the memory subcomponent of the system.

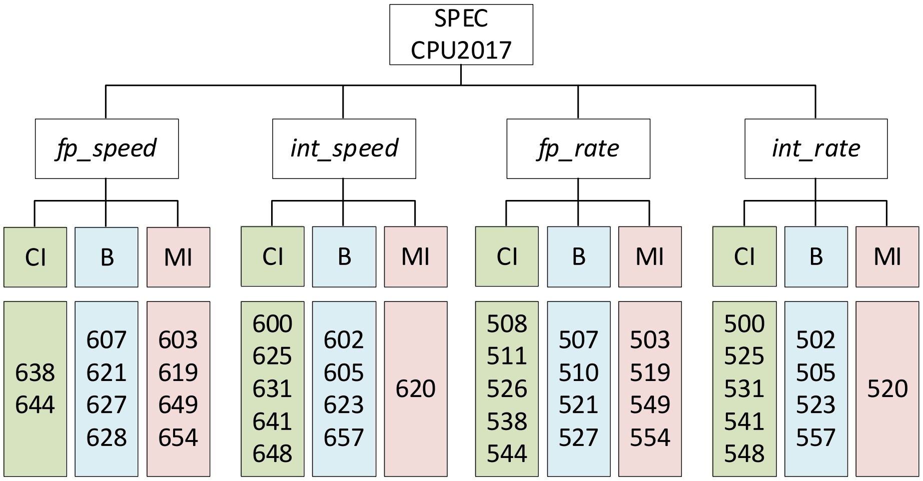


Figure 6.5 SPEC CPU2017 Classification Summary

6.3 Impact of Static Frequency Selection on P and EE

In this section, we explore the impact of clock frequency on performance and energy efficiency by setting the operating points at fixed values, as follows: 0.8 GHz, 1.7 GHz, 2.7 GHz, 3.70 GHz, 4.00 GHz, and 4.30 GHz (Turbo mode). The test machine is fully loaded running six-thread or six-copy SPEC CPU2017 benchmarks. The benchmark execution times and energy consumed is measured for each operating point. For each benchmark, *Bi*, the ratio of the benchmark's execution time while running the core's nominal frequency of 3.7 GHz and the benchmark's execution time while running at a particular frequency *F* is evaluated as follows: *T(Bi, 3.7 GHz)/T(Bi, F)*. This metric is equivalent to the normalized performance for a benchmark *Bi* when running at a frequency, *F*, *P(Bi, F)/P(Bi, 3.7 GHz)*. Similarly, normalized energy-efficiency is defined as *E(Bi,3.7 GHz)/E(Bi, F)*.

Figure 6.6 illustrates how benchmarks' execution times vary with processor clock frequency. Straight horizontal lines with lighter shade represent the ratios of processor clock frequency *F/3.7* GHz, thus serving as indicators of expected performance. The results indicate that increasing processor clock frequency above the nominal frequency is beneficial for a small group of *compute-intensive* benchmarks. For instance, the performance gain of *compute-intensive* benchmarks is ~16% (out of 16% theoretical) when running at 4.30 GHz and ~8% (out of 8% theoretical) when running at 4.00 GHz. The gains are lower for the *balanced* group with 6%, when running at 4.30 GHz. On the other side, by lowering the processor clock frequency below the nominal frequency, the performance is expectedly reduced. However, the performance losses of the benchmarks that are bound by memory are lower than expected. For example, the performance loss for the *memory-intensive* benchmarks is ~3% (27% is theoretically possible) when running at 2.7 GHz and ~11% when running at 1.70 GHz (out of 54%).

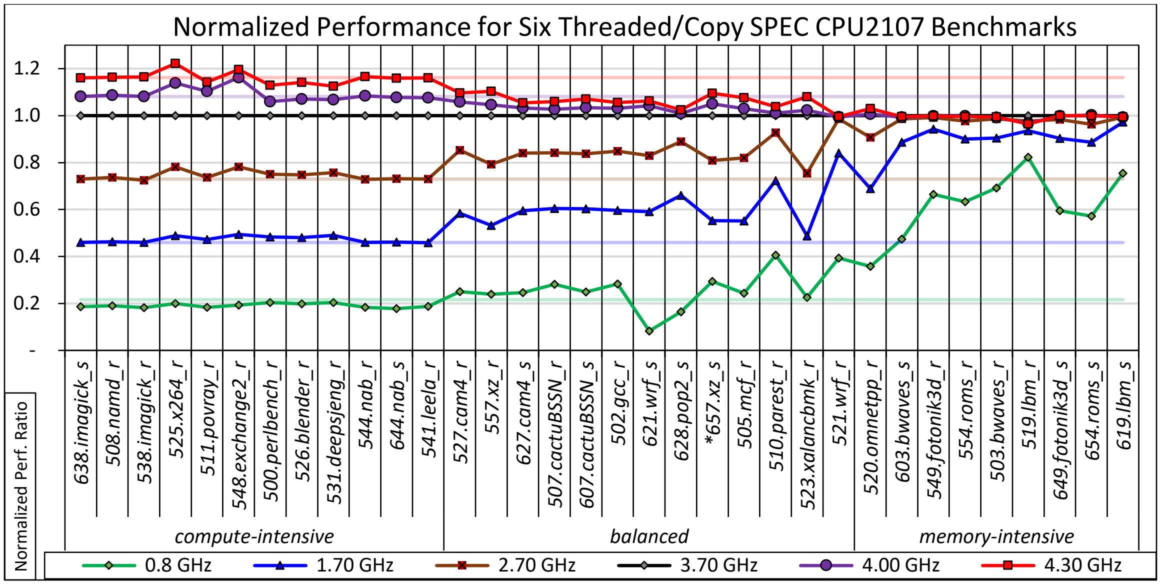


Figure 6.6 Normalized P for SPEC CPU2017 as a Function of Clock Frequency

Figure 6.7 illustrates how the total package energy varies as a function of the processor clock frequency. The results show that by increasing the processor clock frequency above the nominal (lines representing 4.0 GHz and 4.3 GHz), the total energy increases proportionally and thus does not improve energy efficiency even for benchmarks that see significant performance gains. Running at 1.7 GHz and 2.7 GHz improves energy efficiency for all benchmarks, regardless of their characteristics. However, the energy efficiency improvements are the largest for the *memory-intensive* benchmarks. Thus, the *memory-intensive* group sees a relative energy efficiency improvement of 88% at 1.70 GHz. The effects of running at the lowest clock frequency of 0.8 GHz are mixed. Whereas all compute-intensive benchmarks and many memory-intensive benchmarks see an overall loss in energy efficiency because of prolonged execution time, the benchmarks in the memory-intensive group see improvement in energy efficiency even at this operating point.

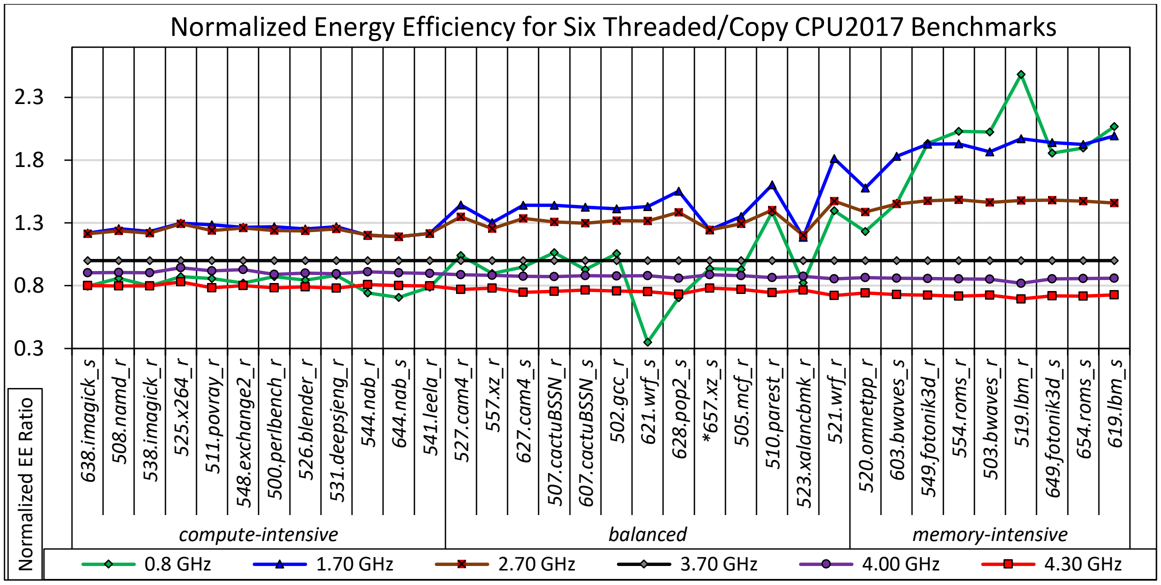


Figure 6.7 Normalized EE for SPEC CPU2017 as a Function of Clock Frequency

Figure 6.8 shows the effect of frequency scaling on the combined metric PxEE when normalized to the nominal frequency of 3.70 GHz across all benchmarks. For each benchmark, the ratio of the *PxEE* when running at the nominal frequency and the *PxEE* when running at frequency *F* is calculated. As expected, *compute-intensive* and *balanced* groups do not see noticeable benefits in scaling frequency. However, the *memory-intensive* benchmarks see significant benefits in running at lower operating frequencies (at ~1.70 GHz) with a relative improvement of ~69%. It is interesting to see that *519.llbm_r* has the best PxEE metric (~102% improvement) at 0.8 GHz. *519.lbm_r* is heavily vectorized and benefits from running at the lowest clock frequency.

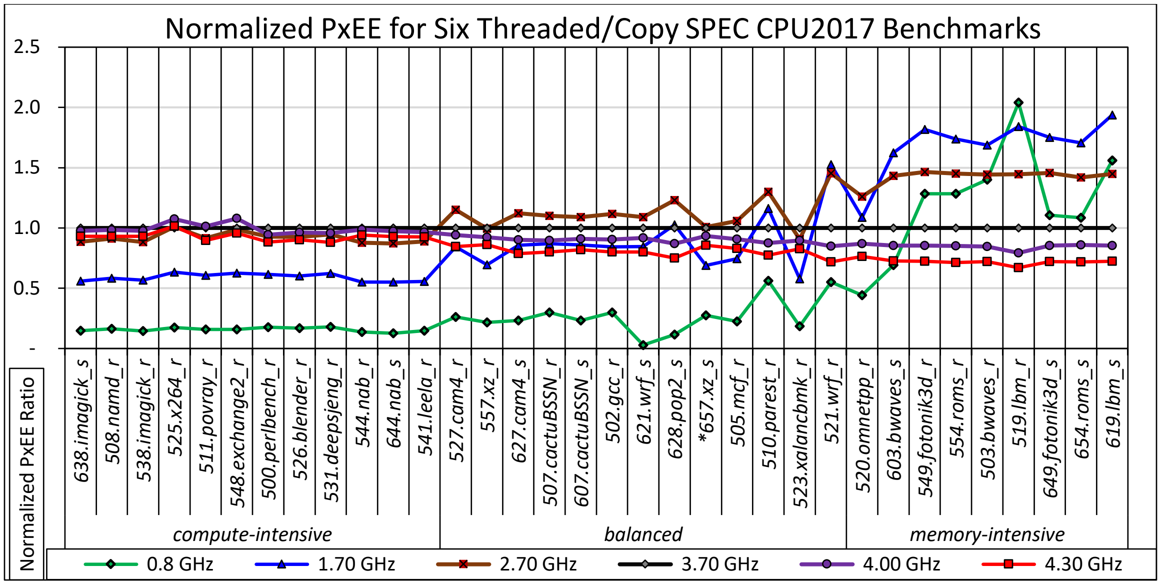


Figure 6.8 Normalized PxEE for SPEC CPU2017as a Function of Clock Frequency

The above results provide strong proof that energy efficiency can indeed be improved if the clock frequency is fixed to an operating point that is the best fit for a given benchmark. To illustrate this point further, Figure 6.9 shows the execution time on the x-axis and energy consumed on the y-axis measured on the test machine while running *649.fotonik3d_s* at 6 threads, while varying the clock frequency from 0.8 GHz to 4.3 GHz from above. Lowering clock frequency from 4.30 GHz to 2.70 GHz does not have a significant negative impact on execution time but saves energy almost 3 times. Further lowering clock frequency beyond 1.7 GHz starts increasing execution time and energy-consumed as well. Thus, from the shape of this energy-time curve, we can say that the most effective operating point for this benchmark should be in a range from 1.7 GHz to 2.3 GHz.

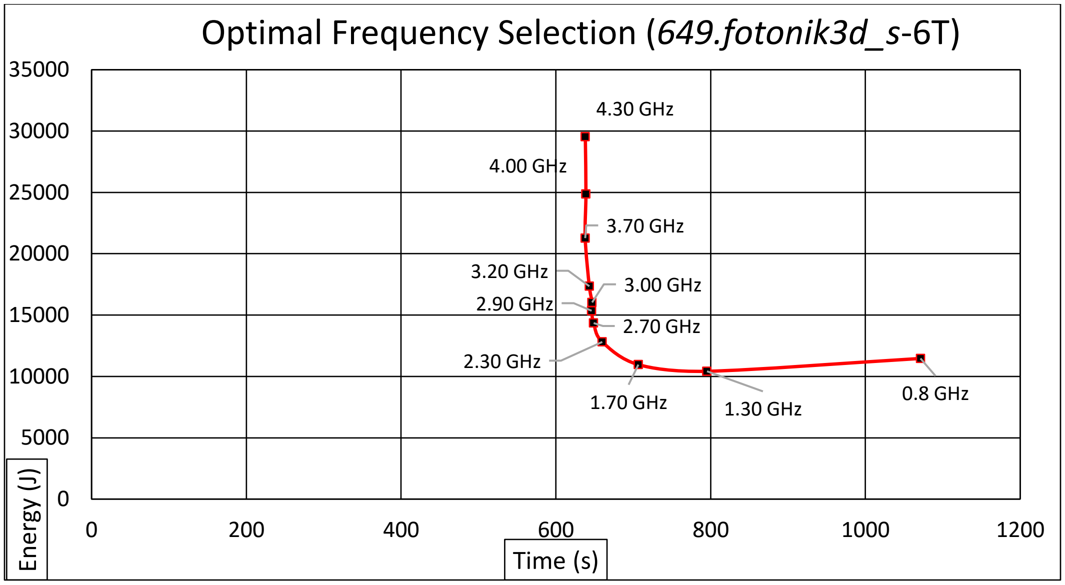


Figure 6.9 Optimal Frequency Selection of *649.fotonik3d_s*

Figure 6.10 quantifies the highest achievable PxEE gains from static frequency selection for each benchmark relative to PxEE measured when running under the *OS-ondemand* governor. Here, we find the operating point that produces the maximum PxEE for a given benchmark. Please note that different benchmarks will have different optimal operating points. This metric is then normalized to the PxEE measured when the corresponding benchmark is run under the *OS-ondemand* governor. The results show that PxEE improvements can be achieved for all benchmarks, though they are the largest for the memory-intensive benchmarks.

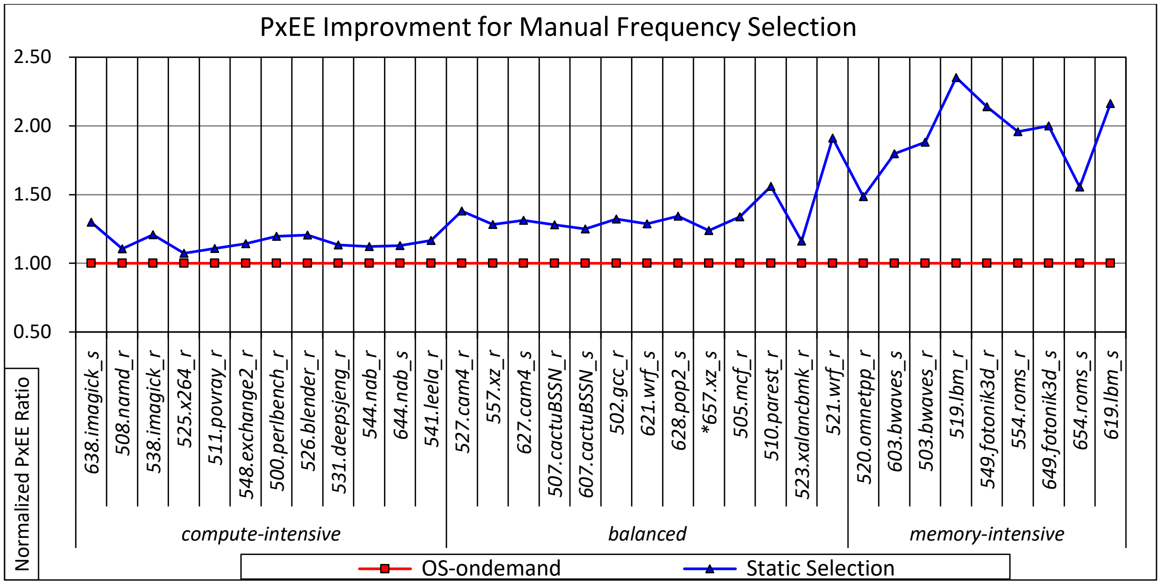


Figure 6.10 Highest Achievable PxEE Gains for Manual Frequency Selection

Whereas improvements demonstrated in Figure 6.10 are significant, finding a perfect operating point for a given benchmark on a given machine is not practical as it would require prior profiling which takes time and energy. In addition, programs go through different phases during their execution, and statically selected frequency throughout benchmark execution cannot provide the best possible results. Hence, better power management techniques have the potential to take advantage of DVFS and provide even better energy efficiency.

CHAPTER 7

RESULTS

In this section, we discuss the results of the experimental evaluation. The baseline performance and energy efficiency are measured on the test machine running the *powersave* governor with the *intel-pstate* driver. As discussed in (2.5), this governor corresponds to the default Linux *ondemand* governor. This governor selects the highest operating frequency (P0 state) during benchmark execution because the test machine is fully loaded. We will refer to this governor as *OS-ondemand*. We measure the performance and energy-efficiency of the proposed techniques *FS-PS*, *FS-TS*, *FS-MS*, and *FS-LLCM*, as well as the previously proposed *FS-CPI,* and then compute derived metrics P.S, EE.I, and PxEE.I. The experiments are conducted for all three workloads of interest: SPEC CPU20017 (Section 7.1), Parsec 3.0 (Section 7.2), and SPECpower_ssj2008 (Section 7.3). Section 7.1 provides an in-depth analysis of the results for our primary workload, discussing separately performance speedup (7.1.1), energy-efficiency improvement (7.1.2), and the product of the two (7.1.3).

Section 7.1.4 provides a comparison of the metrics. Section 7.1.5 discusses some limitations of FS-CPI and Section 7.1.6 puts all the SPEC CPU2017 results together.

7.1 SPEC CPU2017

7.1.1 Performance

Figure 7.1 shows the performance speedup (*P.S* defined in Eq. 5.1) for all considered techniques and benchmarks. The results show that all of the proposed techniques, including FS-CPI, expectedly degrade the performance relative to the *OS-ondemand* governor (red line). The degree of performance degradation varies across the individual techniques and benchmarks.

First, we discuss the results for benchmarks in the *compute-intensive* group. The performance losses of individual techniques are summarized by taking into account the execution times of all benchmarks within this group. FS-PS has the highest performance loss of ~23%. This result is somewhat expected as this governor uses the pipeline slot stall ratio in selecting the next P-state. Very few benchmarks can fully utilize all processor pipeline slots. FS-TS has a lower performance loss of ~14%. For FS-PS and FS-TS, the degree of performance degradation directly correlates to the pipeline slot stall ratio and the total stall cycle ratio, respectively. For example, the FS-PS performance loss for *638.imagick_s* is as lows as 4% because this benchmark has a very high pipeline slot utilization ratio in the entire suite. On the other side, the FS-PS performance loss reaches 35% for *644.nab_s*. FS-MS has an even smaller total loss of only ~9%. FS-LLCM has performance similar to the reference governor with a total performance loss of just ~1%. The maximum FS-LLCM performance loss observed in *526.blender_r* is below 3% and many benchmarks in this group do not see any performance degradation. Finally, FS-CPI has a performance loss of ~7%. The performance losses for this technique are similar to the ones observed in FS-MS.

For the *balanced* benchmarks, the proposed techniques result in even higher performance degradation compared to the *compute-intensive* benchmarks. FS-PS has the highest total performance loss of ~32%, ranging from 9% (*521.wrf_r*) to 51% (*523.xalanbcmk_r*). FS-TS has a total performance loss of 23%, exhibiting very similar trends as FS-PS, albeit with smaller losses. FS-MS has a total performance loss of ~19%. We see this behavior as a consequence of a higher percentage of stalls being caused by either the front-end or back-end. These stalls will lead to transitioning to lower clock frequencies. However, lowering clock frequencies often negatively affects performance in this type of benchmark. This is especially evident for *523.xalancbmk_r* which has a high degree of stalls caused by memory references that are resolved in the L3 cache. On the other side, FS-LLCM and FS-CPI have somewhat smaller performance degradation, the total losses are ~9% and ~10%, respectively. These two techniques exhibit similar behavior for this group of benchmarks as well.

In the case of the *memory-intensive* benchmarks, the total performance degradation is significantly smaller for all considered techniques. The *OS-ondemand* will place the processor in the highest operating frequency (P0), even though the majority of clock cycles are stalls caused by the memory subsystem. FS-PS, FS-TS, FS-MS, and FS-LLCM have performance degradation of ~15%, ~9%, ~8%, and 12%, respectively. FS-CPI has the smallest performance degradation of just ~3%. The trends in performance losses observed in FS-CPI and FS-LLCM deviate from each other in this group of benchmarks.

Considering all the benchmarks together, taking the total execution times of all the benchmarks, FS-PS has the total performance loss of ~23%, FS-TS ~16%, FS-MS ~12%, and FS-LLCM ~9%. Finally, FS-CPI has the smallest performance loss of just ~6%. FS-CPI followed by FS-LLCM provide the smallest performance degradation. In conclusion, if we are interested in performance only, the OS-ondemand governor gives the best results across all the benchmarks in SPEC CPU2017.

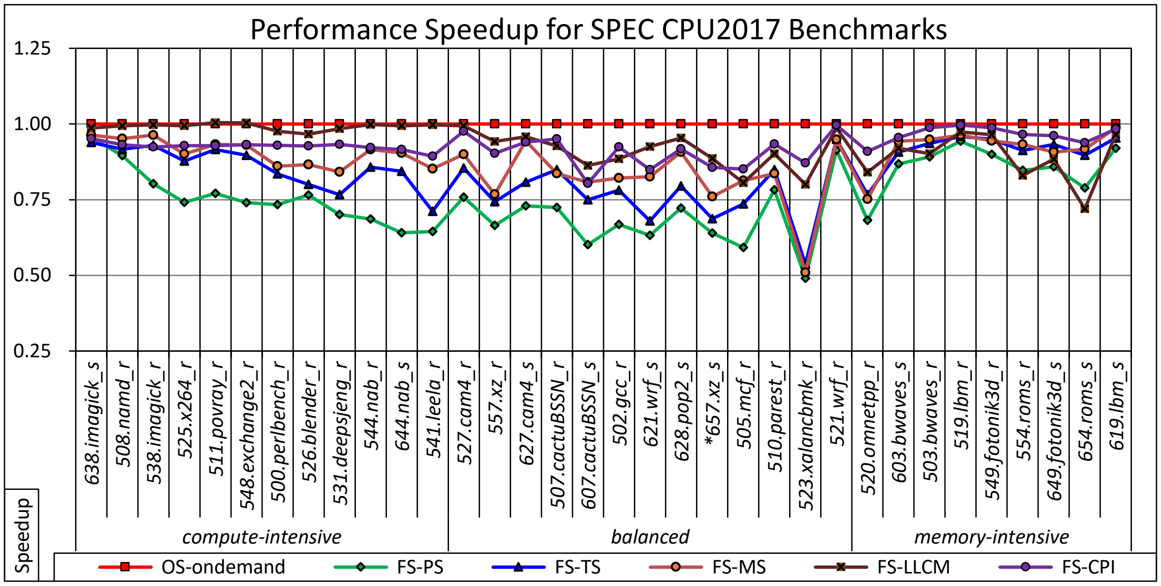


Figure 7.1 Performance Speedup for Individual SPEC CPU2017 Benchmarks

7.1.2 Energy Efficiency

Figure 7.2 shows the energy efficiency improvement as defined in Eq. 5.2. The reference *OS-ondemand* consumes the most energy and FS-PS the least across all benchmarks. In the case of the *compute-intensive* benchmarks, FS-PS and FS-TS provide energy efficiency improvements of ~34% (ranging from 5% to 63%) and ~24% (ranging from 9% to 51%), respectively. These energy savings significantly outweigh the corresponding performance losses. FS-MS has a modest total energy-efficiency improvement of ~15% and FS-LLCM ~1%. FS-CPI has a total energy-efficiency improvement of ~14%.

In the case of the *balanced* benchmarks, the energy efficiency improvements are significantly higher than those observed in the *compute-intensive* benchmarks. FS-PS and FS-TS improve energy efficiency by ~95% (from 72% to 132%) and ~77% (from 19% to 86%), respectively. FS-MS and FS-LLCM improve the total energy efficiency by ~58% and ~32%, respectively. The total EE.I for FS-CPI is ~34%, ranging from 20% to 45%.

The highest energy-efficiency improvements are observed for the *memory-intensive* benchmarks. FS-PS provides the highest total energy-efficiency improvements of 183% (from 126% to 225%), followed by FS-TS with 154% (from 113% to 194%). FS-MS and FS-LLCM improve energy efficiency by ~122% and 138%, respectively. Finally, FS-CPI also improves energy efficiency, but by only ~84%.

Considering all of the benchmarks together, summarizing the total energy consumed for all the benchmarks regardless of their group, FS-PS has the total energy-efficiency improvement of ~92%, FS-TS ~75%, FS-MS ~58%, and FS-LLCM ~44%. Finally, FS-CPI has the smallest gains of just ~41%. The gains in energy efficiency outweigh the performance losses in all considered techniques.

FS-PS and FS-TS both rely on microarchitecture events that fully capture the utilization of the pipeline, whereas FS-MS and FS-LLC rely on events that capture the effectiveness of the memory subsystem alone. The results indicate that the former have a higher potential to improve energy efficiency in SPEC CPU2017 benchmarks.

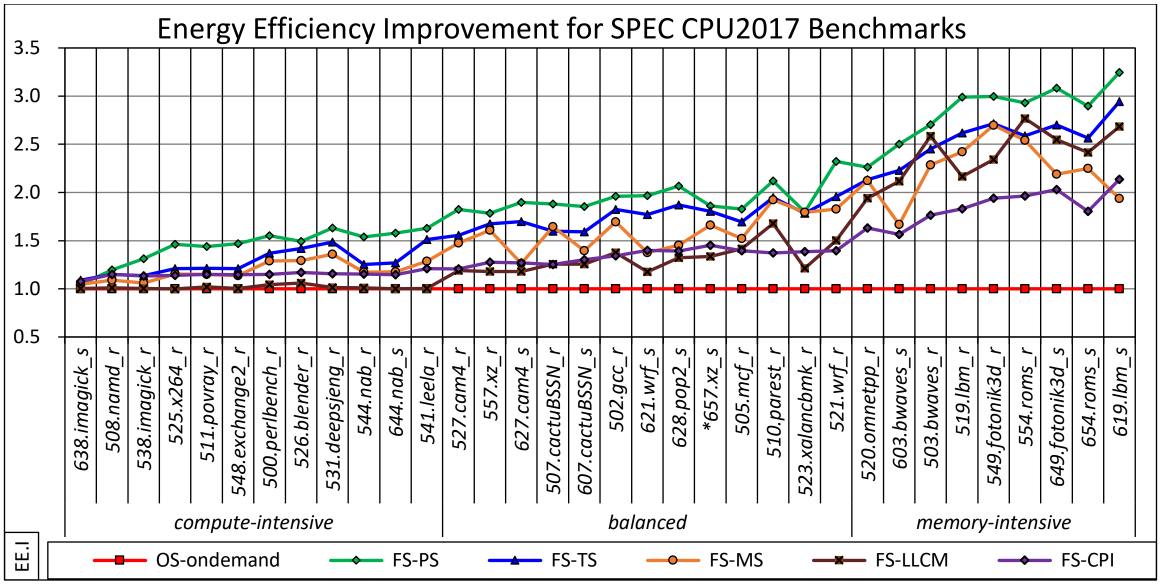


Figure 7.2 Energy-Efficiency Improvement for SPEC CPU2017 Benchmarks

7.1.3 PxEE

Finally, we evaluate the impact of the proposed techniques on the combined metric PxEE. Figure 7.3 shows PxEE.I, as defined in Eq. 5.3. All techniques provide improvements in PxEE relative to the reference governor. These improvements are as high as 6% (FS-TS) for the *compute-intensive* benchmarks. For the *balanced* benchmarks, FS-PS, FS-TS, and FS-MS PxEE improvements are ~32%, ~34%, and ~28%, respectively. The FS-LLCM and FS-CPI improvements are ~20% and ~21%, respectively. When considering the *memory-intensive* benchmarks, the proposed techniques improve PxEE significantly: FS_PS provides the highest gains at 141%. Next FS-TS has a gain of 132%. FS-MS and FS-LLCM also have respectable gains of ~100% and 110% respectively. Finally, FS-CPI has the smallest PxEE improvements of only ~78%. The relatively higher loss in performance observed for FS-PS and FS-TS is compensated by the gains in energy efficiency to provides positive PxEE gains. However, we should note that the proposed techniques underperform for

523.xalancbmk_r providing a PxEE loss of ~3%. This is because the benchmark has a significant number of stalls that are L3 bound. As the stalls are resolved on-chip, reducing operating frequency severely hurts performance.

Considering all of the benchmarks together i.e., the execution times and energies consumed are summarized across all benchmarks before they are used in equations (3)-(5), FS-PS and FS-TS perform the best in PxEE, both providing a total improvement of ~48% relative to the reference governor. FS-MS and FS-LLCM have PxEE improvements of ~39% and 31%, respectively. Finally, FS-CPI has an overall gain of ~32%.

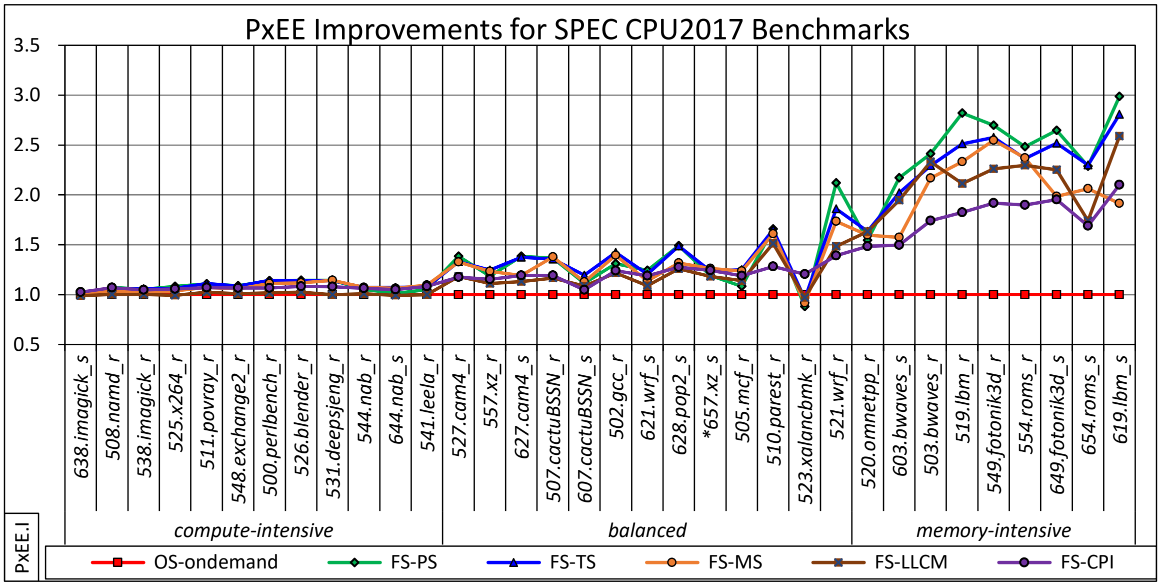


Figure 7.3 PxEE Improvement for SPEC CPU2017 Benchmarks

7.1.4 Discussion: On Effectiveness of Different Techniques

From the results in the previous section, we notice that the effectiveness of the proposed techniques generally follows a trend. For example, FS-PS and FS-TS show consistently higher gains than others across all benchmarks, and the relative difference between the two shows a trend. However, the other two techniques, FS-MC and FS-LLCM, show significant deviation for specific benchmarks. In this section, we investigate the irregularity in the results by taking a specific example of *549.fotonik3d_r*. Table 7.1 shows the P.S, EE.I and PxEE for all the proposed techniques for this benchmark. Considering PxEE, we see that FS-PS has the highest gains, Next, FS-TS and FS-MS have similar gains of over ~155%. However, LLCM has a lower PxEE gain at ~126%.

Table 7.1: P.S, EE.I, and PxEE for *549.fotonik3d_r*

	FS-PS	FS-TS	FS-MS	FS-LLCM
P.S	0.90	0.95	0.94	0.96
EE.I	3.00	2.71	2.70	2.34
PxEE	2.70	2.58	2.55	2.26

Figure 7.4 shows the run-time measurements of the input parameters that proposed techniques rely on in making frequency changes. The primary y-axis shows the pipeline stall ratio, the total stall ratio, and the memory stall ratio (used in FS-PS, FS-TS, and FS-MS, respectively) and the secondary y-axis shows the last level misses per kilo instructions during the execution of *549.fotononki3d_r*. Both y-axes represent the full range of the parameters that are mapped onto operating frequencies. As these parameters have different ranges, they are all normalized to the 0-1 scale. They are sampled periodically every 100 ms. The measurements are taken when the FS-TS governor is in charge of the clock.

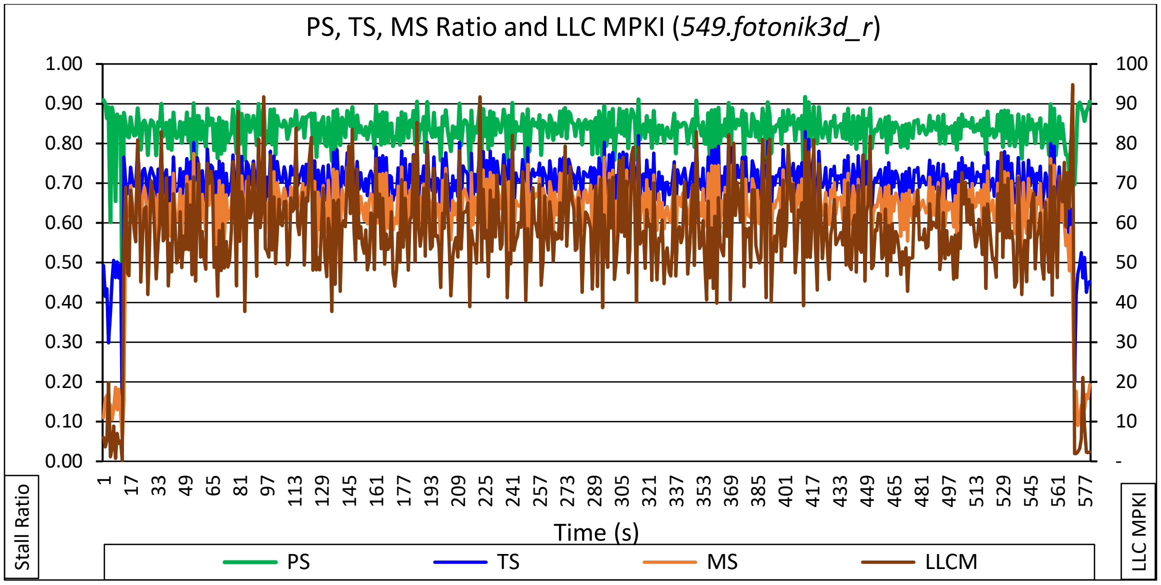


Figure 7.4 Runtime metrics measurements for *549.fotonik3d_r*

We observe that the values of the parameters remain fairly consistent throughout the execution of the benchmark. FS-PS has an average stall rate of ~90%. As a result, a lower operating frequency is selected. Next, FS-TS and FS-MS have similar average stall rates of ~70% and 65%, respectively. Thus, they will select a higher operating frequency than the FS-PS. Finally, the FS-LLCM parameter values overlap with the FS-MS parameter values, but with higher fluctuations. These fluctuations cause larger frequency changes, resulting in higher power consumption and lower PxEE gains.

Please note that the ratios vary significantly for different benchmarks and as a result, there are instances where some techniques perform better on some benchmarks and not on others. Though FS-LLCM provides the lowest gains, in this case, there are examples where FS-LLMC has similar PxEE as FS-PS and FS-TS (e.g, 603.bwaves_s).

7.1.5 Discussion: Limitations of CPI

To further investigate the pros and cons of the proposed techniques, we compare FS-TS and FS-CPI. We take *654.roms_s* as an example. Regarding performance, both FS-TS and FS-CPI have similar outcomes with performance losses of ~10% and 6%, respectively. However, FS-TS improves energy efficiency over the reference governor by ~159%, while FS-CPI only by ~96%. When considering PxEE, we have ~130% improvement by FS-TS and only ~69% from FS-CPI. FS-TS is the better choice as it provides similar performance, with significant gains in energy efficiency. Figure 7.3 shows the run-time measurements of the input parameters these two techniques rely on. The primary y-axis shows the total stall ratio (used in FS-TS) and the secondary y-axis shows the average CPI during the execution of *654.roms_s*. Both y-axes represent the full range of the metrics that are mapped onto operating frequencies. They are sampled periodically every 100 ms. The measurements are taken when the FS-CPI governor is in charge of the clock control.

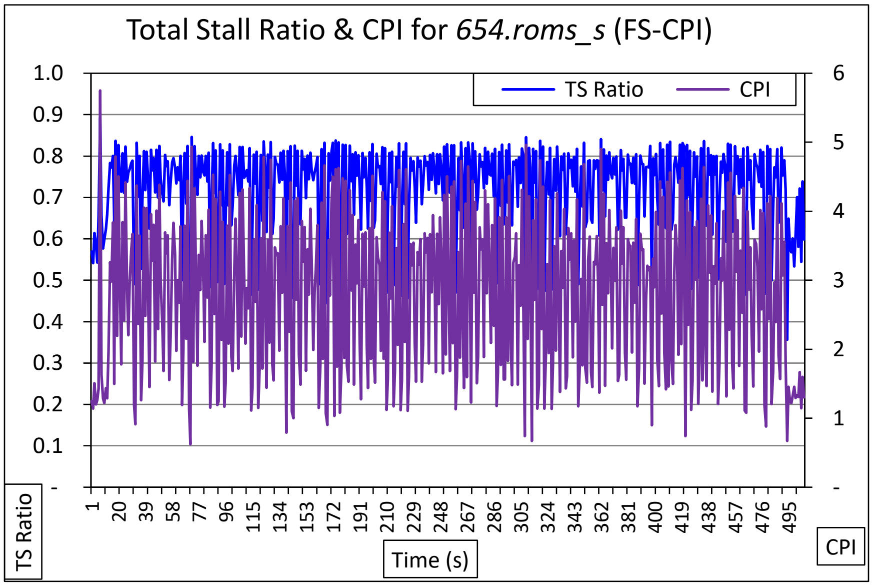


Figure 7.5 Runtime measurements of the total stall ratio and the average CPI for *654.rom_s*.

The total stall ratio and CPI are both sensitive to the operating frequency. However, we can observe that the amplitudes of the changes in CPI are significantly larger than the amplitudes of the changes in the total stall ratio. These fluctuations cause FS-CPI to change the operating frequency very often (sometimes at the end of each sample period), causing wide swings in the operating frequency, e.g., from 1.5 to 4.0 GHz on the test machine. Those unnecessary wide-swing transitions add to the overall energy consumption and are detrimental to performance. On the other side, the total stall ratio is more stable causing smaller frequency changes, providing overall better energy efficiency.

7.1.6 Summary: Putting it all Together

Figure 7.6 shows the summarized view of all three metrics for the evaluated techniques for a fully loaded machine running SPEC CPU2017. The performance speedup and energy-efficiency improvement metrics are calculated by considering all of the benchmarks together, i.e., the execution times and energies consumed are summarized across all benchmarks before they are used in equations (Eq. 5.1)-(Eq. 5.3). The results show that the proposed techniques indeed significantly improve energy efficiency (green line) relative to the *OS-ondemand* governor (black line). FS-PS improves energy efficiency by ~92% albeit at the cost of performance degradation of ~23%. This technique works best for energy-constrained systems, where energy efficiency is the primary focus.

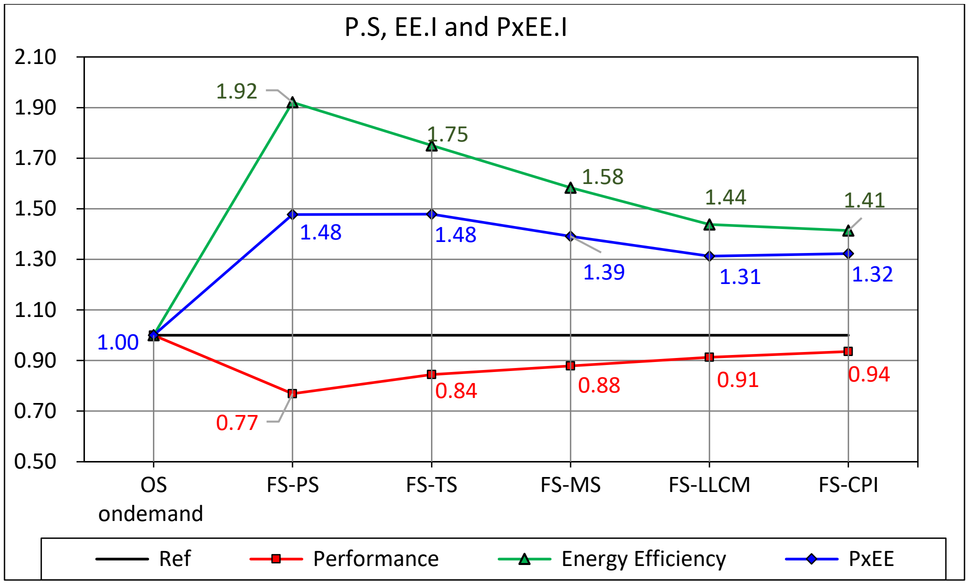


Figure 7.6 Summary of total performance, energy efficiency, and PxEE improvements for SPEC CPU2017 on a Fully Loaded CPU

FS-TS improves energy efficiency by 75% at the cost of performance degradation of 16%. Considering PxEE improvement both FS-PS and FS-TS have identical improvements of 48%. FS-MS further reduces performance loss (~8%) at the cost of reduced energy efficiency (~58%). Finally, FS-LLCM provides a performance-oriented approach where gains in energy efficiency come only from the *memory-intensive* class of benchmarks.

So far, we only looked at the results for a fully loaded system. Figure 7.7 shows all three metrics for a partially loaded system. The CPU2017 speed benchmarks are run with 4 threads and the rate benchmarks are run with 4 copies. We observe similar trends in P.S, EE.I, and PxEE.I, albeit the gains are somewhat smaller.

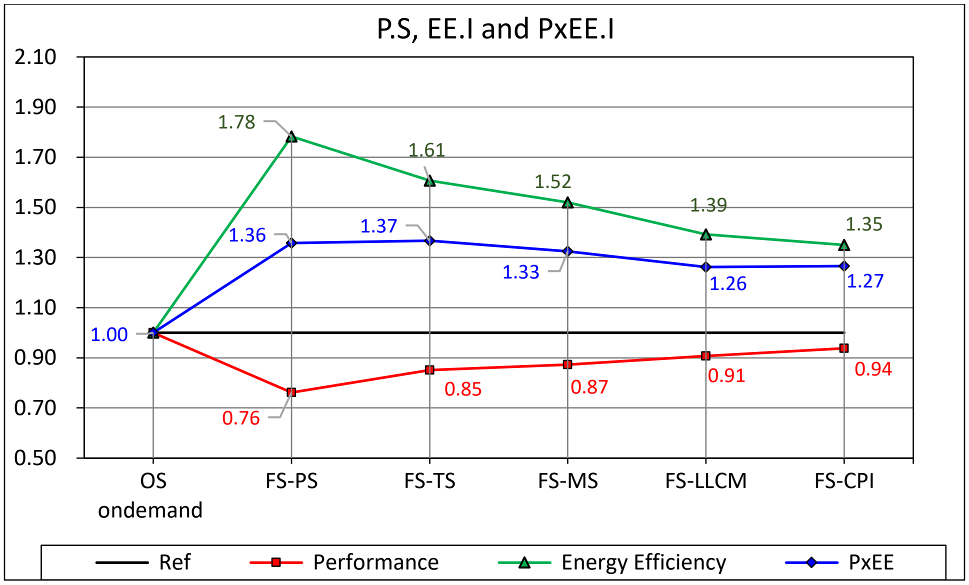


Figure 7.7 Summary of total performance, energy efficiency, and PxEE improvements for CPU2017 on a partially loaded machine.

For example, the PxEE.I of the best performing FS.PS and FS.TS techniques are 36% and 37%, respectively (relative to 48% and 48% for the fully-loaded machine). This is due to the active and passive power consumption from the idle cores that also operate in the same frequency without doing any work. Thus, we can expect that the benefits provided by the proposed techniques will decrease as the machine load decreases. However, these decreases may not be present in processors that can support individual cores to enter C-states while other cores are fully active.

7.2 Parsec-3.0

In this section, we validate the proposed techniques with an alternate multi-threaded workload, Parsec 3.0. The workload is older and lighter when compared to the CPU2017 suites. Figure 7.8 shows the summarized view of all three metrics for the evaluated techniques for a fully loaded machine. The performance speedup and energy-efficiency improvement metrics are calculated by considering all of the Parsec-3.0 benchmarks together, i.e., the execution times and energies consumed are summarized across all benchmarks before they are used in equations Eq. 5.1-Eq. 5.3.

In terms of performance (red line), as expected we see degradation from every proposed technique. FS-PS has the worst performance loss, whereas FS-LLCM and FS-CPI have the least. In terms of energy efficiency (green line), results indicate noticeable gains across all techniques. FS-PS has the highest gains in energy efficiency and FS-LLCM has the least. Finally, when we consider PxEE, the gains are modest ranging from ~5% (FS-PS) to ~18% (FS-MS).

The main factor contributing to the low gains observed here is that the benchmarks are compute-intensive with a small memory footprint. As the benchmarks have significantly aged, modern systems do not generate significant execution stalls for the Parsec suite.

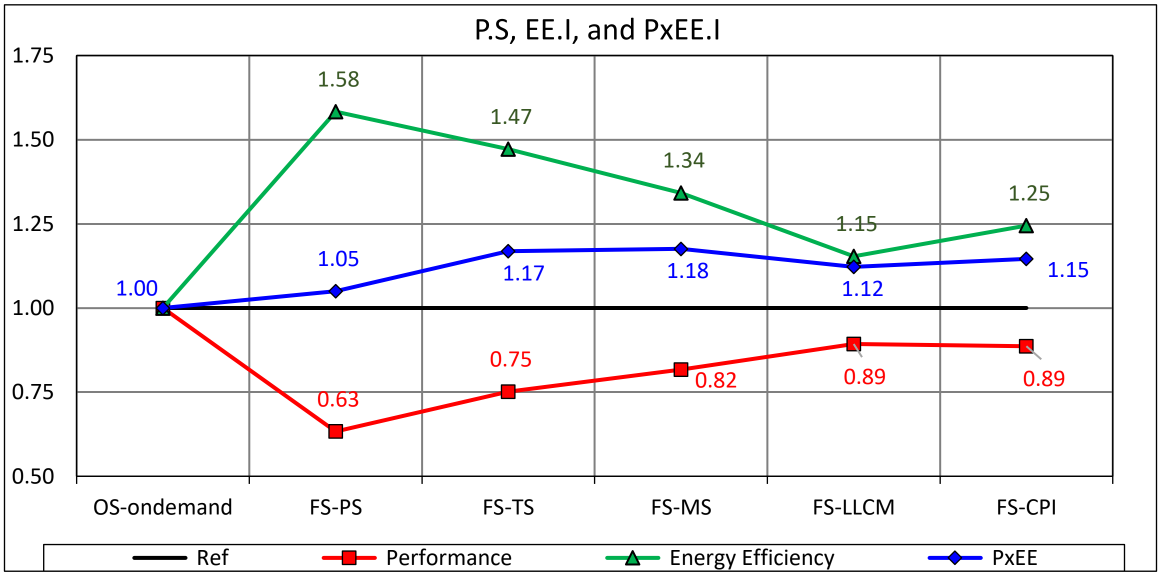


Figure 7.8 Summary of total performance, energy efficiency, and PxEE improvements for Parsec 3.0

7.3 SPECpower2008jbb

We now evaluate the proposed DVFS techniques on the SPECpower2008_jbb benchmark suite. Performance for this benchmark is reported in the number of ssj operations under different load levels. Power consumption and overall effectiveness expressed in the number of ssj operations per Watt are also reported. Figure 7.9 shows the raw performance achieved for each of the DVFS techniques while varying the transactional load from 0% to 100%. As expected, the *OS-ondemand* provides the best performance. Next, we observe a similar trend in terms of performance for all the techniques across all the load levels. As the benchmark runs a single workload with a different amount of delay is introduced in each load level, we take the example of three specific load levels of 100%, 60%, and 10% load level.

In all three cases, FS-PS has the highest performance loss of ~47%. This is due to the low utilization of the pipeline. The benchmark is unable to effectively issue enough micro-operation to populate all the available slots. Next, FS-TS and FS-MS provide similar losses of ~32% and ~30%. This indicates that a higher percentage of the stalls are memory-related. Finally, FS-LLCM and FS-CPI have a similar loss of ~8% and 6%. We observe three distinct pairings as discussed above across all load levels.

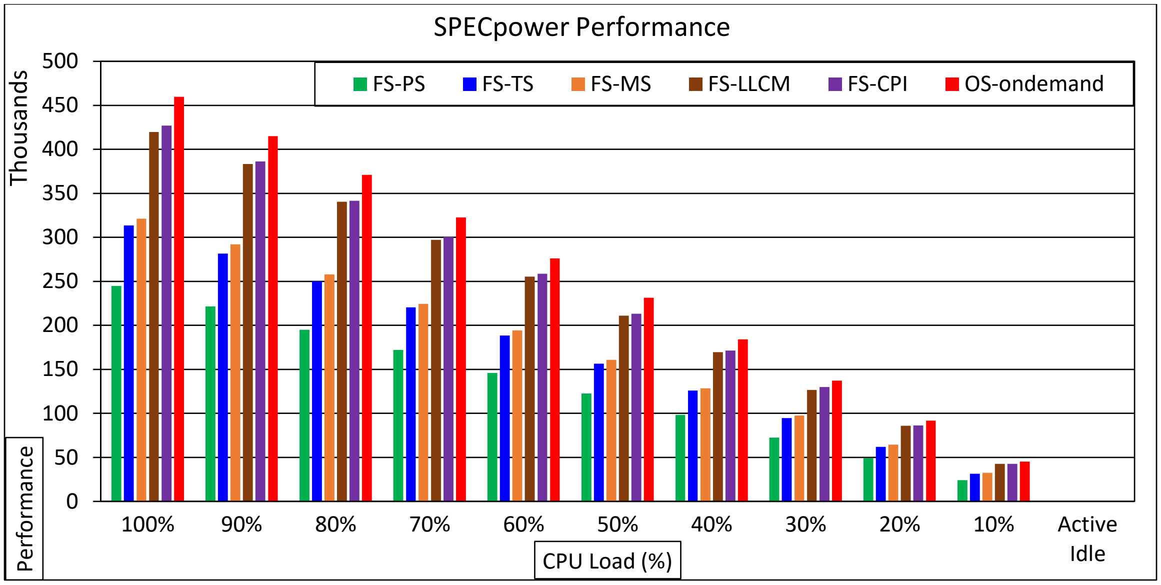


Figure 7.9 SPECpower Raw Performance (ssj ops)

Figure 7.10 shows the active processor power consumed during the execution of the benchmark while varying the machine load levels. The results show that the proposed techniques significantly reduce active power consumption. The OS-ondemand power consumption is significantly higher than the proposed techniques.

Similar trends observed for performance are observed for power consumption as well. We see reduced power consumption metrics for all the proposed techniques. We take the example of 100%, 60%, and 10% load. At 100% and 60%, we observe three distinct groupings. FS-PS has an improvement in power consumption of ~67% at both the load levels. Next, FS-TS and FS-MS have similar improvement at ~58%. Finally, FS-LLCM and FS-CPI have improvements of ~24%. However, while considering ~10% load we see deviation for the results. FS-PS improvements in power consumption are reduced to ~34%, FS-TS and FS-MS have ~29% and ~20% improvements respectively. Whereas FS-LLCM and FS-CPI have an increase in power consumption by ~14% and ~7% respectively. This is partly due to an increase in power consumption (~1 W) due to the technical implementation.

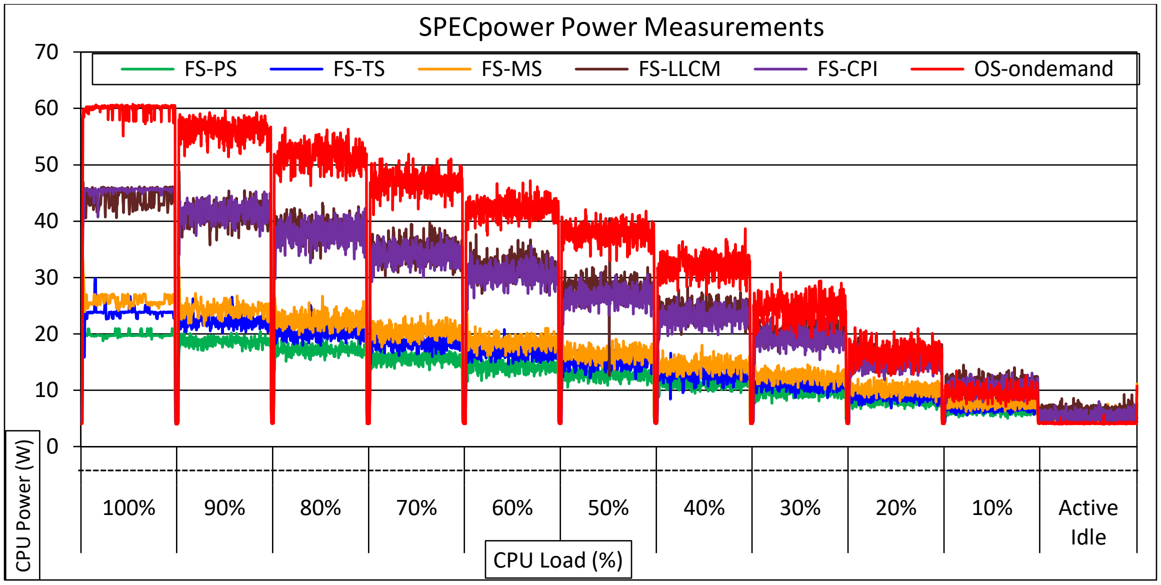


Figure 7.10 Runtime Power Measurements of SPEC power Benchmark

Finally, we calculate the performance/watt to evaluate the overall effectiveness of different power management techniques. Figure 7.11 shows the performance/watt of all the proposed DVFS techniques at each load level. We see that when our proposed techniques significantly outperform the state-of-the-art OS-demand by providing higher performance per unit watt consumed. Considering 100% and 60% load levels, FS-PS and FS-MS have a similar gain of 60%, whereas FS-TS has the best performance per watt metric of ~73%. Finally, FS-LLCM and FS-CPI have a gain of ~25%. Considering the load of ~10% the proposed techniques underperform due to low utilization and the added power consumption.

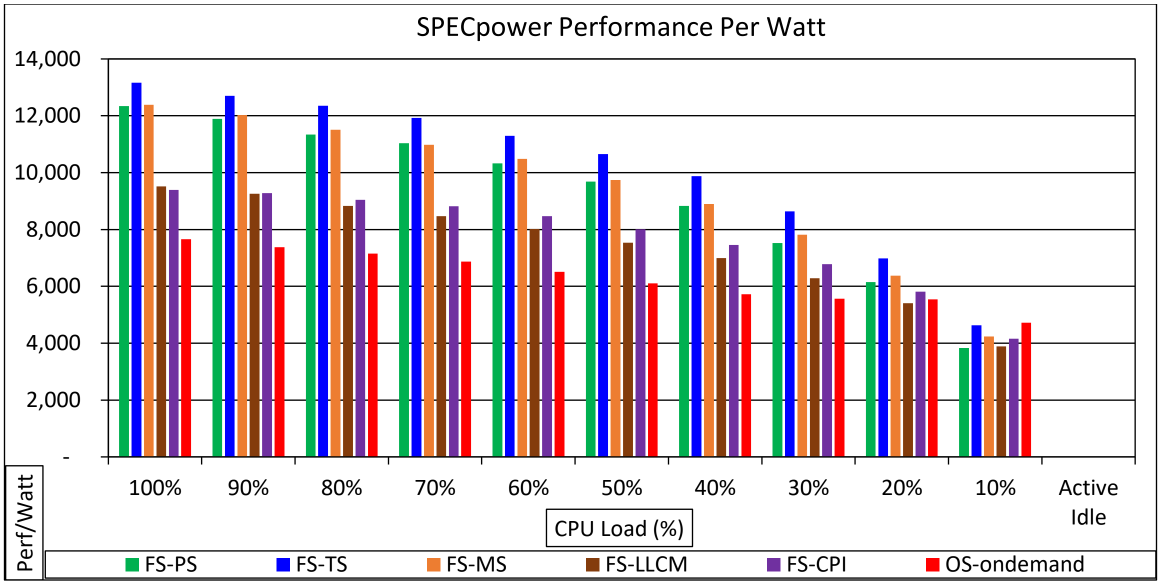


Figure 7.11 Performance Per Watt for all the Proposed Techniques

CHAPTER 8

RELATED WORK

Power management techniques have been an integral part of all modern computing systems from handheld services to large servers [17]. One of the most effective approaches to regulating processor power consumption is dynamic voltage and frequency scaling. The impact of DVFS on energy consumption is significant and a large body of prior work has explored various avenues to improve energy efficiency using DVFS. Researchers have explored analytical models, simulations, and experimental evaluations to propose and test ideas to improve performance and reduce power consumption through DVFS.

Multiple research studies focus on the development of analytical models for static and dynamic power consumption of various processor components in an effort to estimate the run-time power consumption of the entire processor. B. Goel *et al.* present a methodology for deriving analytical models for static and dynamic power consumption and use those models for uncore and cores [20]. The study also shows how to isolate and quantify the power consumption of different processor components. A study from Esmaeilzadeh *et al.* develops power models for multi-core processors. The models are used to predict the effects of semiconductor node and frequency scaling on the performance and power of future generations of multicore processors [18]. The transition and energy overhead associated with DVFS is modeled by S. Park *et al.* [47]; the researchers provide a detailed analysis of various components associated with the overhead. A study from T. Rauber *et al.* develops analytical models for power consumption of Intel Haswell and Skylake processors and uses them to determine a clock frequency that minimizes power consumption [49] [50]. They verified the accuracy of their model through experimental evaluations using the NAS benchmarks [57] and found that the optimal frequency found through their models provides a 7% gain in energy efficiency relative to the default configuration. Predictive models for multi-dimensional power-performance optimizations on many-core processors are investigated in a study by M. Curtis-Maury *et al.* [13]. They explore interactions between DVFS and dynamic concurrency throttling (DCT) and develop a library that supports fine-tuning of operating points of cores running different threads in an OpenMP application.

Y. Cho *et al.* present analytical solutions to the problem of determining energy-optimal voltage scale factors for each task, while allowing each task to be preempted and to have its energy cost function [12]. Their experimental study reports a 10% additional savings in the total system energy compared to the previous leakage-aware DVS schemes. A. Iyer *et al.* presents an online DVFS technique by utilizing interface queues to guide the DVFS control in multiple clock and voltage domain architectures [33].

Estimating processor power consumption for a given application is challenging due to the internal execution characteristics of applications that exploit hardware very differently. Various methods to estimate the power consumption of a processor have been studied. They can be classified into four categories: (a) cycle level estimation

[45]; (b) instruction-level power analysis (ILPA) [59]; (c) functional level power analysis [48]; and (d) system-level power estimation [56]. In cycle-level estimation, the power consumption of each processor unit (arithmetic units, registers, memory, etc.) is estimated at each clock cycle. This method is not feasible anymore as the complexity of modern processors makes this method too expensive in terms of computation. Instruction level power analysis involves estimating the power consumption of each instruction executing in the processor. The power consumption of a program can then be computed as the sum of the power consumed by each instruction of which is composed. The modeling complexity grows with the number of instructions that the processor can execute concurrently. The functional level power model by G. Qu *et al.* initially utilizes empirical data collection to identify the power consumption linked to different functional blocks of the processor [48]. The model utilizes the empirical data set to predict the power consumption of embedded software. The system-level power estimation model abstracts the low-level power estimation techniques by considering the entire system. The model encompasses the functional level power estimate to set up generic power models for various modules of the system. A simulation framework at the transactional level evaluates the activities of the functional units to determine system power [51].

Regarding the experimental evaluation of DVFS, several studies have shown that energy profiling to find the best operating point for each benchmark can be very beneficial for both performance and energy efficiency [5] [15] [24] [27] [36]. Here, a benchmark is run at the fixed operating point that is found to provide the highest energy efficiency. Results from such studies show that the energy efficiency of memory-bound applications can improve by over 150% with minimal loss in performance [15] [27]. However, this approach relies on previous profiling to find the best operating point and does not accommodate for runtime changes throughout a benchmark's execution. Recently, De Vogeleer *et al.* use measurements in a controlled environment on a mobile CPU to confirm a realistic power/energy equation for CPU power [14]. They show the existence of an energy/frequency convexity rule; that is, the existence of a unique optimum frequency for energy efficiency for a fixed workload.

Accurate power measurement techniques are vital for experimental evaluations. A study from C. Isci *et al.* proves the accuracy of the onboard power monitoring infrastructure during the run-time in several sub-modules of x86 Intel processors [32]. Studies have also evaluated the onboard energy-oriented features available on modern processors [22] [54]. The results of these studies give us confidence in the measurement infrastructure available in modern processors.

Meanwhile, finding an efficient method to select an optimal operating frequency during a program's run-time remains a challenging problem [43]. Several studies have proposed techniques for selecting the operating frequency that outperforms the current power governors. A study from M. Nanja *et al.* suggested using the performance counter to measure instructions per cycle (IPC) and memory references per cycle to make scaling [44]. Another such method proposes the use of the cycles-per-instruction (CPI) when selecting P-states [3] [34]. An experimental study from D. Molka *et al.* proposed the use of hardware counters to select a particular frequency of operation [55]. The study uses instructions per memory access to make frequency decisions. Hwisung Jung *et al.* presented a power management framework for dynamic continuous frequency adjustment which provides power-saving opportunities by dynamically and continuously adjusting a variable operating frequency on a functional level granularity [35]. Utilizing the basic premise of eliminating the power and delay costs incurred by the power state transitions which involve clock generators (e.g., PLL), the authors report a ~14% savings in energy consumption.

The use of turbo-mode has gained significant traction and is now a standard feature in processors [53] [76]. Each new generation has more aggressive use of the turbo mode to provide better performance. However, applications that were written several years ago (also referred to as aging applications) may see a significant negative impact on functionality and performance. A study from S. Matheus *et al.* explored the impact of turbo modes on the execution time of parallel programs and provided guidelines to developers to maximize performance and maintain functionality [38]. The performance impact of DVFS for realistic memory systems is explored in [39]. The experimental evaluation is done with the SPEC CPU 2006 benchmarks, which are based on a sequential workload.

In this study, we propose alternate dynamic voltage and frequency scaling techniques and extend our earlier research [24] that was based on workload-driven DVFS. Our study utilizes architectural evaluation to make an informed decision on dynamically selecting P-states that results in significant energy savings. Our study explores several avenues of dynamic voltage and frequency scaling and we also compare a previously proposed technique and show that our proposals provide significantly higher gains.

CHAPTER 9

FUTURE WORK

There are several possible avenues for future work concerning the proposed DVFS techniques. First, we start with the implementation of the techniques. The *OS-ondemand* and our proposed techniques use a linear mapping to map a given metric of interest to the available P-states on a system. This assumes that the relationship between the power and frequency is linear. However, in reality, we can observe through measurements that this is not the case.

Figure 9.1 shows the full load processor power consumption while varying the operating frequency. We observe that the relationship between power and frequency is near-linear till 3.14 GHz and we see a steep curve thereafter, especially in the turbo mode. An increase in frequency above the nominal frequency only provides modest results with significant energy consumption. We also note that a similar performance curve can be obtained with any modern processor supporting turbo mode.

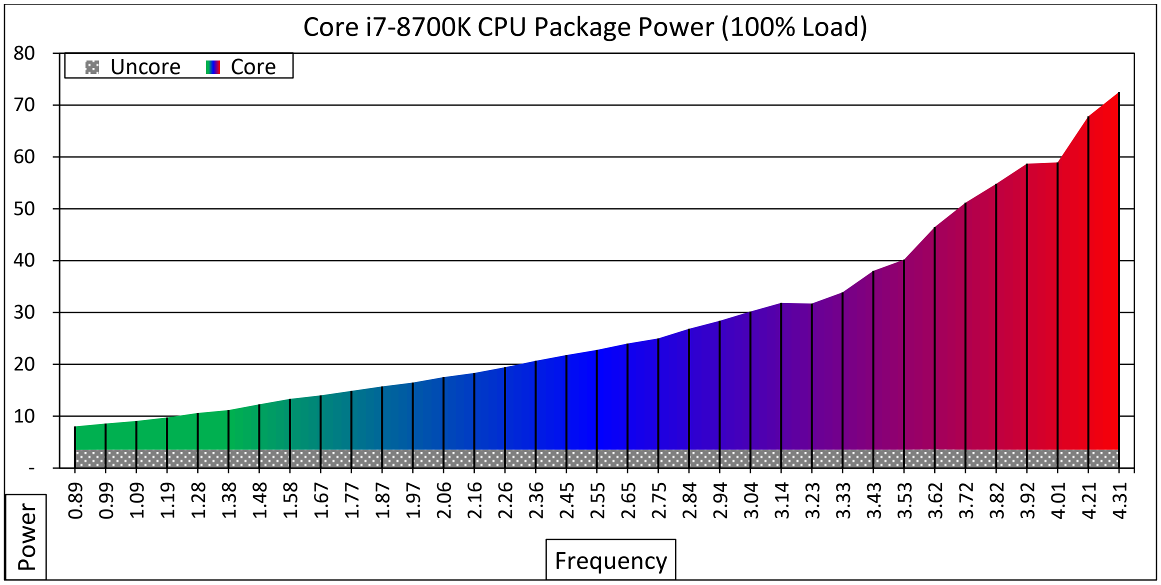


Figure 9.1 Full Load Processor Power Consumption at various Operating Frequency

Next, each of the proposed techniques uses just two events to determine the operating frequency. As seen in CHAPTER 7, a single technique does not provide the best results for all classes of benchmarks. One way to rectify this would be to use multiple events to determine the next operating point. Adding additional events would improve the robustness of the techniques and also help handle anomalies such as *523.xalancbmk_r*, which underperforms for stall-based techniques.

The current implementation shown in the study has a worst-case idle time of 13 ms for a sampling period of 10 ms. However, when the CPU is fully loaded the execution time of the implementation varies. As a result, the technique was invoked every 100 ms. A drawback here is that the sampled characteristics may not be of interest by the time the requested P-state transition is complete. The primary workload used in the study does not get affected by the invocation period due to the steady-state characteristic. However, for an application that has multiple execution phases, a higher invocation frequency is more desirable.

All our experiments were performed on a single socket system where the processor only supports socket level P-state management. It would be interesting to evaluate the impact of DVFS on a multi-socket machine that supports core-level P-state management. We also note that the experiments were performed on an Intel machine. However, similar infrastructure to access the PMU and change P-states exist in ARM and AMD processors, albeit with a smaller granularity or access.

Nowadays, Internet-of-Things (IoT) devices generate data at high speed and large volume. Often the data require real-time processing to support high system responsiveness which can be supported by localized Cloud and/or Fog computing paradigms. An interesting direction for future work is to evaluate the impact of the proposed DVFS techniques on IoT devices. As these devices are power sensitive, techniques to improve efficiency would greatly improve the longevity and affordability of these devices.

CHAPTER 10

CONCLUSIONS

Dynamic voltage and frequency scaling are one of the most important tools in regulating the power consumption of modern processors. The state-of-the-art demand-based implementations of DVFS governors in modern OSes favor performance over energy efficiency. As the operating costs of computing continue to increase, more power-oriented DVFS governors need to be implemented.

This dissertation presents the results of the measurement-based analysis of various dynamic voltage and frequency scaling techniques. We observe that the current implementations of DVFS in the OSes are not ideal for *memory-intensive* benchmarks. Based on our architectural evaluation, we propose, implement, and experimentally evaluate four new techniques that determine the P-state of the processor cores using metrics derived from the PMU events: (i) the ratio of pipeline slot stalls (FS-PS), (ii) the ratio of cycle stalls (FS-TS), (iii) the ratio of memory-related cycle stalls (FS-MS), and (iv) the number of last level cache misses per kilo instructions (FS-LLCM). We also investigate the effectiveness of the previously proposed CPI-based frequency selection and describe its shortcomings.

The study first quantitatively evaluates the effectiveness of the state-of-the-art power management technique in modern processors (the ondemand governor) and determines its shortcomings, especially in terms of its energy efficiency. It provides an in-depth analysis of the SPEC CPU2017 benchmarks using the Top-down microarchitectural analysis method and classifies the benchmarks into three groups based on their characteristics. Through experimental evaluation using three different types of work-loads, namely SPEC CPU2017, Parsec 3.0, and SPECpower_ssj2008 the effectiveness of the proposed techniques and the existing state-of-the-art are shown.

The results of the experimental evaluation show that the proposed techniques significantly improve EE and PxEE metrics relative to the existing approaches. PxEE improves from 31% to 48% when all benchmarks are considered together. Furthermore, we find that the proposed techniques are especially effective for a class of memory-intensive benchmarks with a PxEE improvement from 100% to 141%. The proposed techniques also outperform an earlier DVFS proposal that utilizes the cycles-per-instruction metric when changing processor states.

GLOSSARY

Notation	**Description**
ACPI	Advanced Configuration & Power Interface
AVX	Advanced Vector Instruction Set
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
CPB	Energy Performance Bias
CPU	Central Processing Unit
CSR	Control & Status Register
DVFS	Dynamic Voltage & Frequency Scaling
EET	Energy Efficiency Turbo
FIFO	First in First Out
FIVR	Fully Integrated Voltage Regulators
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HWPM	Hardware Power Management
IoT	Internet-of-Things
ILPA	Instruction Level Power Analysis

IPC	Instructions Per Cycle
IPS	Intel parallel Studio
LLC	Last Level Cache
MST	Model Specific Registers
OOO	Out-of-Order
OS	Operating System
PCU	Power Control Unit/ Package Control Unit
PLL	Phase Locked Loop
PMA	Power Management Agents
PMU	Performance Monitoring Unit
RAPL	Running Average Power Limit
SMT	Simultaneous Multi-Threading
SPEC	Standard Performance Evaluation Corporation
TBM3	Turbo Boost Max 3.0
TDP	Thermal Design Power
TMAM	Top-down Microarchitectural Analysis Method
UFS	Uncore Frequency Scaling

REFERENCES

[1] Alif Ahmed and Kevin Skadron. 2019. Hopscotch: a micro-benchmark suite for memory performance evaluation. In *Proceedings of the International Symposium on Memory Systems* (MEMSYS '19), Association for Computing Machinery, New York, NY, USA, 167–172. DOI:https://doi.org/10.1145/3357526.3357574

[2] A. R. Alameldeen and D. A. Wood. 2006. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro* 26, 4 (July 2006), 8–17. DOI:https://doi.org/10.1109/MM.2006.73

[3] Peter Altevogt, Hans Boettiger, Wesley M. Felter, Charles R. Lefurgy, Lutz Stiege, and Malcolm S. Ware. 2008. Method for Autonomous Dynamic Voltage and Frequency Scaling of Microprocessors. Retrieved April 22, 2021 from https://patents.google.com/patent/US20080098254/en

[4] Anders S. G. Andrae and Tomas Edler. 2015. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* 6, 1 (June 2015), 117–157. DOI:https://doi.org/10.3390/challe6010117

[5] Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. 2016. Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Trans. Archit. Code Optim.* 13, 4 (December 2016), 51:1-51:26. DOI:https://doi.org/10.1145/3011017

[6] Malini K. Bhandaru and Eric J. Dehaemer. 2013. Providing energy efficient turbo operation of a processor. Retrieved May 9, 2021 from https://patents.google.com/patent/WO2013137859A1/en

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (PACT '08), Association for Computing Machinery, New York, NY, USA, 72–81. DOI:https://doi.org/10.1145/1454115.1454128

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*, ACM Press, Berlin, Germany, 41–42. DOI:https://doi.org/10.1145/3185768.3185771

[9] Edward A. Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J. Lambert, Kaladhar Radhakrishnan, and Michael J. Hill. 2014. FIVR — Fully integrated voltage regulators on 4th generation Intel® Core™ SoCs. In *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014*, 432–439. DOI:https://doi.org/10.1109/APEC.2014.6803344

[10] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. 2009. Evaluation of the Intel® Core™ i7 Turbo Boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 188–197. DOI:https://doi.org/10.1109/IISWC.2009.5306782

[11] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 12, 4 (December 2015), 41:1-41:22. DOI:https://doi.org/10.1145/2829952

[12] Youngjin Cho, Naehyuck Chang, Chaitali Chakrabarti, and Sarma Vrudhula. 2006. High-level power management of embedded systems with application-specific energy cost functions. In *Proceedings of the 43rd annual Design Automation Conference* (DAC '06), Association for Computing Machinery, New York, NY, USA, 568–573. DOI:https://doi.org/10.1145/1146909.1147057

[13] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. 2008. Prediction models for multi-dimensional power-performance optimization on many cores. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 250–259.

[14] Karel De Vogeleer, Gerard Memmi, Pierre Jouvelot, and Fabien Coelho. 2014. The Energy/Frequency Convexity Rule: Modeling and Experimental Validation on Mobile Devices. In *Parallel Processing and Applied Mathematics* (Lecture Notes in Computer Science), Springer, Berlin, Heidelberg, 793–803. DOI:https://doi.org/10.1007/978-3-642-55224-3_74

[15] Armen Dzhagaryan and Aleksandar Milenković. 2014. Impact of thread and frequency scaling on performance and energy in modern multicores: a measurement-based study. In *Proceedings of the 2014 ACM Southeast Regional Conference* (ACM SE '14), Association for Computing Machinery, New York, NY, USA, 1–6. DOI:https://doi.org/10.1145/2638404.2638473

[16] Stéphane Eranian. 2008. What can performance counters do for memory subsystem analysis? In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)* (MSPC '08), Association for Computing Machinery, New York, NY, USA, 26–30. DOI:https://doi.org/10.1145/1353522.1353531

[17] H. Esmaeilzadeh, T. Cao, X. Yang, S. Blackburn, and K. McKinley. 2012. What is Happening to Power, Performance, and Software? *IEEE Micro* 32, 3 (May 2012), 110–121. DOI:https://doi.org/10.1109/MM.2012.20

[18] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2013. Power challenges may end the multicore era. *Commun. ACM* 56, 2 (February 2013), 93–102. DOI:https://doi.org/10.1145/2408776.2408797

[19] Lev Finkelstein, Efraim Rotem, Aviad Cohen, Ronny Ronen, and Doron Rajwan. 2013. Power management for multiple processor cores. Retrieved November 18, 2020 from https://patents.google.com/patent/US8402290B2/en

[20] Bhavishya Goel and Sally A McKee. 2016. A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 273–282. DOI:https://doi.org/10.1109/IPDPS.2016.118

[21] Corey Gough, Ian Steiner, and Winston Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress.

[22] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 896–904. DOI:https://doi.org/10.1109/IPDPSW.2015.70

[23] Ranjan Hebbar and Aleksandar Milenković. 2021. A Preliminary Scalability Analysis of SPEC CPU2017 Benchmarks. In *SoutheastCon 2021*, IEEE, Atlanta, GA, USA, 1–8. DOI:https://doi.org/10.1109/SoutheastCon45413.2021.9401917

[24] Ranjan Hebbar and Aleksandar Milenković. 2021. An Experimental Evaluation of Workload Driven DVFS. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ACM, Virtual Event France, 95–102. DOI:https://doi.org/10.1145/3447545.3451192

[25] Ranjan Hebbar S R. 2018. SPEC CPU2017: Performance, Energy and Event Characterization on Modern Processors. M.S.E. The University of Alabama in Huntsville, United States -- Alabama.

[26] Ranjan Hebbar S R and Aleksandar Milenković. 2019. SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (ICPE '19), ACM, New York, NY, USA, 111–118. DOI:https://doi.org/10.1145/3297663.3310314

[27] Ranjan Hebbar S R and Aleksandar Milenković. 2019. Impact of Thread and Frequency Scaling on Performance and Energy Efficiency: An Evaluation of Core i7-8700K Using SPEC CPU2017. In *2019 SoutheastCon*, IEEE, Huntsville, AL, USA, 1–7. DOI:https://doi.org/10.1109/SoutheastCon42311.2019.9020637

[28] Ranjan Hebbar S R, Mounika Ponugoti, and Aleksandar Milenković. 2019. Battle of Compilers: An Experimental Evaluation Using SPEC CPU2017. In *2019 SoutheastCon*, IEEE, Huntsville, AL, USA, 1–8. DOI:https://doi.org/10.1109/SoutheastCon42311.2019.9020474

[29] J. L. Henning. 2000. SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer* 33, 7 (July 2000), 28–35. DOI:https://doi.org/10.1109/2.869367

[30] J. L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (September 2006), 1–17. DOI:https://doi.org/10.1145/1186736.1186737

[31] Song Huang, Michael Lang, Scott Pakin, and Song Fu. 2015. Measurement and characterization of Haswell power and energy consumption. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing* (E2SC '15), Association for Computing Machinery, New York, NY, USA, 1–10. DOI:https://doi.org/10.1145/2834800.2834807

[32] C. Isci and M. Martonosi. 2003. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 93–104. DOI:https://doi.org/10.1109/MICRO.2003.1253186

[33] A. Iyer and D. Marculescu. 2002. Power efficiency of voltage scaling in multiple clock multiple voltage cores. In *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, 379–386. DOI:https://doi.org/10.1109/ICCAD.2002.1167562

[34] Darrin Paul Johnson, Eric Christopher Saxe, and Bart Smaalders. 2012. Frequency scaling of processing unit based on aggregate thread CPI metric. Retrieved April 22, 2021 from https://patents.google.com/patent/US8219993B2/en

[35] Hwisung Jung and Massoud Pedram. 2008. Continuous Frequency Adjustment Technique Based on Dynamic Workload Prediction. In *21st International Conference on VLSI Design (VLSID 2008)*, 249–254. DOI:https://doi.org/10.1109/VLSI.2008.98

[36] Michael A. Laurenzano, Mitesh Meswani, Laura Carrington, Allan Snavely, Mustafa M. Tikir, and Stephen Poole. 2011. Reducing Energy Usage with Memory and Computation-Aware Dynamic Frequency Scaling. In *Euro-Par 2011 Parallel Processing* (Lecture Notes in Computer Science), Springer Berlin Heidelberg, 79–90.

[37] Arindam Mallik, Bin Lin, Gokhan Memik, Peter Dinda, and Robert P Dick. 2006. User-Driven Frequency Scaling. *IEEE Computer Architecture Letters* 5, 2 (February 2006), 16–16. DOI:https://doi.org/10.1109/L-CA.2006.16

[38] Sandro Matheus V. N. Marques, Thiarles S. Medeiros, Fábio D. Rossi, Marcelo C. Luizelli, Alessandro G. Girardi, Antonio Carlos S. Beck, and Arthur F. Lorenzon. 2019. The Impact of Turbo Frequency on the Energy, Performance, and Aging of Parallel Applications. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, 149–154. DOI:https://doi.org/10.1109/VLSI-SoC.2019.8920389

[39] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. 2012. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 155–165. DOI:https://doi.org/10.1109/MICRO.2012.23

[40] Aleksandar Milenkovic, Vladimir Uzelac, Milena Milenkovic, and Martin Burtscher. 2011. Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems. *IEEE Trans. Comput.* 60, 7 (July 2011), 992–1005. DOI:https://doi.org/10.1109/TC.2010.146

[41] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. 2004. Microbenchmarks for determining branch predictor organization. *Software: Practice and Experience* 34, 5 (April 2004), 465–487. DOI:https://doi.org/10.1002/spe.572

[42] Trevor Mudge. 2001. Power: A First-Class Architectural Design Constraint. *Computer* 34, 4 (April 2001), 52–58. DOI:https://doi.org/10.1109/2.917539

[43] M. Najibi, M. Salehi, A. Afzali Kusha, M. Pedram, S. M. Fakhraie, and H. Pedram. 2006. Dynamic Voltage and Frequency Management Based on Variable Update Intervals for Frequency Setting. In *2006 IEEE/ACM International Conference on Computer Aided Design*, 755–760. DOI:https://doi.org/10.1109/ICCAD.2006.320116

[44] Murthi Nanja. 2010. Performance monitoring based dynamic voltage and frequency scaling. Retrieved August 20, 2019 from https://patents.google.com/patent/US7770034B2/en

[45] M. Nemani and F.N. Najm. 1999. High-level area and power estimation for VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 6 (June 1999), 697–713. DOI:https://doi.org/10.1109/43.766722

[46] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the Linux Symposium*, 223–238.

[47] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. 2013. Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (May 2013), 695–708. DOI:https://doi.org/10.1109/TCAD.2012.2235126

[48] Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. 2000. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Annual Design Automation Conference* (DAC ’00), Association for Computing Machinery, New York, NY, USA, 810–813. DOI:https://doi.org/10.1145/337292.337786

[49] Thomas Rauber, Gudula Rünger, and Matthias Stachowski. 2018. Performance and energy metrics for multi-threaded applications on DVFS processors. *Sustainable Computing: Informatics and Systems* 17, (March 2018), 55–68. DOI:https://doi.org/10.1016/j.suscom.2017.10.015

[50] Thomas Rauber, Gudula Rünger, and Matthias Stachowski. 2019. Model-based optimization of the energy efficiency of multi-threaded applications. *Sustainable*

Computing: Informatics and Systems 22, (June 2019), 44–61. DOI:https://doi.org/10.1016/j.suscom.2019.01.022

[51] Santhosh Kumar Rethinagiri, Rabie Ben Atitallah, and Jean-Luc Dekeyser. 2011. A system level power consumption estimation for MPSoC. In *2011 International Symposium on System on Chip (SoC)*, 56–61. DOI:https://doi.org/10.1109/ISSOC.2011.6089692

[52] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (March 2012), 20–27. DOI:https://doi.org/10.1109/MM.2012.12

[53] Sumit Kumar Saurav, Ganga Prasad G.L, and Manisha Chauhan. 2016. Adaptive Power Management for HPC applications. In *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, 1–7. DOI:https://doi.org/10.1109/ICGHPC.2016.7508065

[54] Rober Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. 2019. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, 399–406. DOI:https://doi.org/10.1109/HPCS48598.2019.9188239

[55] Robert Schöne and Daniel Hackenberg. 2011. On-line analysis of hardware performance events for workload characterization and processor frequency scaling decisions. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering - ICPE '11*, ACM Press, Karlsruhe, Germany, 481. DOI:https://doi.org/10.1145/1958746.1958819

[56] A. Sinha and A.P. Chandrakasan. 2001. JouleTrack-a Web based tool for software energy profiling. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 220–225. DOI:https://doi.org/10.1109/DAC.2001.156139

[57] Vaibhav Sundriyal and Masha Sosonkina. 2018. Modeling of the CPU frequency to minimize energy consumption in parallel applications. *Sustainable Computing: Informatics and Systems* 17, (March 2018), 1–8. DOI:https://doi.org/10.1016/j.suscom.2017.12.002

[58] Guy Therien and Michael Walz. 2006. Power management system that changes processor level if processor utilization crosses threshold over a period that is different for switching up or down. Retrieved November 18, 2020 from https://patents.google.com/patent/US7017060B2/en

[59] V. Tiwari, S. Malik, A. Wolfe, and M.T.-C. Lee. 1996. Instruction level power analysis and optimization of software. In *Proceedings of 9th International Conference on VLSI Design*, 326–328. DOI:https://doi.org/10.1109/ICVD.1996.489624

[60] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 39th International Conference on Parallel Processing Workshops*, 207–216. DOI:https://doi.org/10.1109/ICPPW.2010.38

[61] V. Uzelac and A. Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 207–217. DOI:https://doi.org/10.1109/ISPASS.2009.4919652

[62] Vladimir Uzelac, Aleksandar Milenković, Milena Milenković, and Martin Burtscher. 2014. Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing. *IEEE Transactions on Computers* 63, 4 (April 2014), 1008–1020. DOI:https://doi.org/10.1109/TC.2012.267

[63] Vincent M Weaver. 2013. Linux perf event Features and Overhead. (2013), 7.

[64] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring Energy and Power with PAPI. In

2012 41st International Conference on Parallel Processing Workshops, 262–268. DOI:https://doi.org/10.1109/ICPPW.2012.39

[65] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 35–44. DOI:https://doi.org/10.1109/ISPASS.2014.6844459

[66] Huazhe Zhang and Henry Hoffmann. 2015. A Quantitative Evaluation of the RAPL Power Control System. *Feedback Computing 2015* (2015), 6.

[67] 2016. Intel® 64 and IA-32 Architecture's Optimization Reference Manual. (June 2016), 672.

[68] Power Management States: P-States, C-States, and Package C-States. Retrieved August 21, 2020 from https://software.intel.com/content/www/us/en/develop/articles/power-management-states-p-states-c-states-and-package-c-states.html

[69] Advanced Configuration and Power Interface - an overview | ScienceDirect Topics. Retrieved January 20, 2021 from https://www.sciencedirect.com/topics/computer-science/advanced-configuration-and-power-interface

[70] Power Management with Lenovo Efficiency Mode. 13.

[71] US7840825B2 - Method for autonomous dynamic voltage and frequency scaling of microprocessors - Google Patents. Retrieved October 20, 2019 from https://patents.google.com/patent/US7840825B2/en

[72] Perf : Linux profiling with performance counters. *Perf Wiki*. Retrieved March 19, 2018 from https://perf.wiki.kernel.org/index.php/Main_Page

[73] Intel® VTune™ Amplifier 2018 User's Guide. *Intel Developer Zone*. Retrieved March 28, 2018 from https://software.intel.com/en-us/vtune-amplifier-help-introduction

[74] Intel Tick-Tock Model. *Intel*. Retrieved April 11, 2018 from https://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html

[75] What Is Intel® Turbo Boost Technology? *Intel*. Retrieved April 28, 2021 from https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html

[76] Intel® Turbo Boost Technology 2.0. *Intel*. Retrieved May 9, 2021 from https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[77] An Overview of the 6th Generation Intel® Core™ Processor (Code-named... *Intel*. Retrieved April 27, 2021 from https://www.intel.com/content/www/us/en/develop/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake.html

[78] Advanced Configuration and Power Interface - an overview | ScienceDirect Topics. Retrieved April 21, 2021 from https://www.sciencedirect.com/topics/computer-science/advanced-configuration-and-power-interface

[79] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. *Intel*. Retrieved June 3, 2020 from https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html

[80] Recognize and Measure Vectorization Performance. *Intel*. Retrieved January 30, 2021 from https://www.intel.com/content/www/us/en/develop/articles/recognizing-and-measuring-vectorization-performance.html

[81] Welcome to LaCASA. Retrieved May 12, 2021 from http://lacasa.uah.edu/portal/index.php

[82] Intel® Core™ i7-8700K Processor Product Specifications. *Intel® ARK (Product Specs)*. Retrieved March 23, 2018 from https://tinyurl.com/ybcw5vc8

[83] SPEC CPU® 2017. Retrieved March 19, 2018 from https://www.spec.org/cpu2017/

[84] The PARSEC Benchmark Suite. Retrieved May 12, 2021 from https://parsec.cs.princeton.edu/

[85] SPECpower_ssj® 2008. Retrieved May 12, 2021 from https://www.spec.org/power_ssj2008/