# Low-cost Techniques for Enhancing Energy Efficiency and Information Security in Next Generation Multi-core Server System Design

by Fan Yao

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of the George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

August 31, 2018

Dissertation directed by

Guru Prasadh Venkataramani
Associate Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Fan Yao has passed the Final Examination for the degree of Doctor of Philosophy as of June 7, 2018. This is the final and approved form of the dissertation.

**Low-cost Techniques for Enhancing Energy Efficiency and Information Security in Next Generation Multi-core Server System Design**

Fan Yao

Dissertation Research Committee:

Guru Prasadh Venkataramani, Associate Professor of Engineering and Applied Science, Dissertation Director

Tian Lan, Associate Professor of Engineering and Applied Science, Committee Member

Suresh Subramaniam, Professor of Engineering and Applied Science, Committee Member

Ahmed Louri, Professor of Engineering and Applied Science, Committee Member

Miloš Doroslovački, Associate Professor of Engineering and Applied Science, Committee Member

*Dedicated to my beloved parents*

## Acknowledgments

First, I would like to thank my advisor, Professor Guru Prasadh Venkataramani for his tremendous guidance, patience and support throughout my Ph.D. program. As an advisor, he is always there to inspire interesting research directions, provide useful feedbacks and offer thinking in unique perspectives. From him, I learned how to identify important problems, develop effective solutions and more importantly how to present ideas in a compelling manner. Professor Venkataramani is also an awesome mentor and friend. He gave me valuable advices that help me stay focused, think big and keep positive. I really appreciate what I gained from him in both research and life.

I would like to thank my dissertation committee members, who have truly guided me for my successes. I really feel fortunate to have collaborated with many of them. Especially, I have been working with Professor Subramaniam since the beginning of my Ph.D. program. He has not only given me constructive advices but also provided outstanding feedback from a different research background. Professor Lan is extremely supportive, and I was able to get great suggestions from him for almost any topic. Professor Doroslovački has been a constant source of knowledge and thoughts. He has offered many inspirations during my problem-solving process. I would like to also thank Professor Howie Huang, Professor Zhenyu Li and Professor David Nagel for their instructions and help on various occasions. I want to thank Marco Suarez and Jason Hurlburt from SEAS facility for their incredible support on the research facilities.

I also would like to thank my friends and colleagues in GWU, Ming Yao, Jie Chen, Hang Liu, Yongbo Li, Jingxin Wu, Hongfa Xue, Hongyu Fang, Sai Santosh Dayapule, Yurong Chen, Bingqian Lu, Juzi Zhao, Yu Xiang, Wenhui Zhang, Shijing Li, Maotong Xu, Ambaw Ambaw, Sultan Alamro, Hao Zheng, Yuede Ji, Pradeep Kumar and many others, who have filled my Ph.D. journey with lots of fun and joy.

Finally, I would like to thank my parents, Weibin Yao and Lianhua Wang for their relentless love and unconditional support. I would not get so far without them and I

am deeply grateful to them.

This dissertation is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557, CNS-1718133 and CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684.

**Abstract of Dissertation**

**Low-cost Techniques for Enhancing Energy Efficiency and Information Security in Next Generation Multi-core Server System Design**

**Thesis Statement: Energy efficiency and information security have become two critical design considerations for multi-core server systems. This dissertation provides in-depth understandings of these two important aspects, and proposes low-cost and effective techniques to improve energy efficiency and enhance information security for next generation multi-core server systems.**

The rapid advancements in software have necessitated the design of computer platforms with enormous computing capability. To enable modern software paradigms, multi-core server systems have become the mainstream computing infrastructures as they offer faster speed and higher parallelism. Traditionally, a huge amount of efforts are made to optimize the performance of server hardware. With the rapid growth of multi-core server systems and the surging trend of information management on these platforms, there is a growing need to consider two critical design aspects beyond performance: *energy efficiency* and *information security*.

Energy consumption contributes significantly to the operational costs as server systems nowadays are more and more energy-hungry. Unfortunately, existing multi-core server hardware is extremely energy inefficient, especially under low-utilizations. As a result, considerable wasteful energy is consumed when the system is idle. Meanwhile, many server workloads have stringent Quality of Service (QoS) constraints to be satisfied. To meet the QoS constraints, service providers typically over-provision hardware resources that are often unused under regular operation. This further exacerbates the energy inefficiency issue in large-scale server systems. Improving energy efficiency for multi-core server systems while maintaining various application QoS constraints is an important yet challenging task.

While higher energy efficiency is undoubtedly beneficial, it is equally important to sustain an information-secure computing environment. Today's end users are increasingly relying on server systems for storing and processing their personal data. Information leakage can lead to huge financial loss to both service users and providers. In recent years, it has been shown in several studies that the underlying multi-core processor architectures are vulnerable to information leakage attacks that can pose significant threats to information security. These attacks are extremely dangerous as they allow secret data to be exfiltrated between isolated domains at relatively high bit rates. To enhance information security in multi-core servers, it is essential to perform a comprehensive investigation of information leakage vulnerabilities and carefully design countermeasures that guard the server hardware against exploitations by malicious parties.

As energy efficiency and information security are becoming more critical, computer architects and system administrators are urgently looking for efficient solutions to address them. To enable fast adoption in practical settings, it is essential to design *cost-effective techniques* that judiciously leverage existing supports and keep the level of hardware changes as low as possible.

This dissertation aims to tackle the energy efficiency and information security challenges in multi-core systems with low costs. Three novel techniques that make smart use of existing hardware supports to offer optimized energy efficiency for QoS constrained applications are proposed. We build realistic QoS-aware system frameworks that improve energy savings under various workload characteristics. To enhance server information security, we systematically explore the information leakage vulnerabilities in modern multi-core architecture. Our work results in the findings of two new types of high-speed information leakage exploitations: *NUMA-based covert channels* and *cache coherence state-based covert channels*. We propose low-cost defense techniques that effectively quantify and thwart the newly discovered information leakage attacks. Our contributions open new directions for computer architects and system designers to build affordable solutions that will push the envelope of energy efficiency and address emerging information security issues in multi-core server systems.

## Table of Contents

## List of Figures

## List of Tables

# Chapter 1 Introduction

## 1.1 Multi-core Server Issues beyond Performance

The breakdown of Moore's law has significantly pushed the wide adoption of multi-core processor to the computing landscape during the past decade. With the growing computation demands from modern software paradigms, computing system providers are rapidly incorporating multi-core processors into their server systems. Conventionally, a tremendous amount of endeavors are put to optimize the performance of multi-core server systems. While there is no doubt that performance is critical, the evolution of information technology has been increasingly arousing the attention of energy efficiency and information security as the key design aspects beyond performance.

***The need for server energy efficiency.*** Today's multi-core servers are increasingly energy-hungry. It is reported that large-scale server systems such as data centers and server farms account for almost 3% of the US domestic energy consumption [85]. Increasing demand from users for personalized and contextual retrieval of large volumes of data and the associated computations have exacerbated the energy issues. Maximizing server energy efficiency is the key to saving energy and amortizing operational costs. Unfortunately, contemporary server systems are extremely energy inefficient due to two main factors: *First*, current server hardware from processors to system components suffer from lack of energy proportionality. Studies by Barroso et al. [47] have shown that servers with 30% utilization level typically consume about 60% of the peak power. Besides, a big portion of power is drawn from server platforms even when they are completely idle. *Second*, server-class latency-critical workloads typically have stringent service level agreements that are required to be met. To ensure QoS, service providers often over-provision servers to satisfy the peak load while leaving a large amount of the servers under-utilized or idle most of the time [107]. This further exacerbates the server energy efficiency issue.

To overcome the energy efficiency dilemma, we need a systematic approach that

understands power characteristics of server hardware (e.g., cores, processors, and peripheral infrastructure) as well as the interactions between software applications and these hardware components. Meanwhile, due to the considerable variabilities in workloads, there does not exist a single configuration for energy management that works well across various scenarios concerning different system utilizations, server configurations and performance constraints. Determining individual low power management policy in each scenario manually is cost-prohibitive and impractical. Therefore, it is essential to explore adaptive mechanisms that can automatically adapt to workload characteristics and improve energy efficiency in future server platforms.

***The growing concern of information security on hardware.*** While energy efficiency of server systems is clearly necessary, ensuring information security and reducing financial cost related to information leakage on server systems is of equal significance. During the past decade, the proliferation of personal and sensitive data has been increasingly incentivizing adversaries to penetrate the system confinement mechanisms for data theft. Security breaches leading to information leakage have introduced massive economic loss every year [63]. Conventionally, attackers target software-level vulnerabilities to carry out attacks. With rapid improvements in software confinement and isolation mechanisms, it is becoming increasingly difficult for information leakage attacks to exploit software vulnerabilities. To bypass software-level protections, attackers are turning to implement their exploitations on shared hardware resources. It has been shown in several recent studies that various performance optimizations (such as SMT and caching) in modern processor architecture allow sensitive bits to be leaked to malicious parties in the form of side or covert timing channels [30]. Information leakage attacks targeting hardware are especially worrisome since adversaries communicate simply by modulating the timing of resource accesses, and do not explicitly transmit or receive bits. Therefore, the attackers do not leave any physical trace during the process of communication, which makes them extremely hard to be detected. Since multi-core processors generally feature multitude of shared hardware resources, they unavoidably offer richer space for timing channel exploits. To protect multi-core servers, as the first step, it becomes valuable to

comprehensively evaluate multi-core processor hardware architecture and microarchitecture designs, and perform intensive investigations to unravel information leakage vulnerabilities. We should then devise effective strategies to neutralize their efforts in order to prevent such information leakage. We envision that information security would be regarded as the first order design constraint in computer architecture in the near future.

As multi-core server and software application continue to evolve, enhancing energy efficiency and information security becomes an indispensable mission. While approaches that leverage customized hardware or a complete redesign of processor architecture to address these two qualitative aspects of computing are possible, these solutions typically incur high design cost and thus cannot be deployed in most practical settings. Therefore, it is equally important to explore low-cost techniques by either using existing hardware supports or introducing minimal hardware changes when necessary to enable timely deployment. Our work highlights the criticality of understanding energy efficiency and information security for next generation multi-core server system design and builds stepping stones for further explorations in these directions.

## 1.2 Overview

This dissertation contains five chapters.

- Chapter 1 motivates the need for improving the two qualitative aspects of computing in multi-core server systems, namely energy efficiency and security (with respect to information leakage). It also identifies the significance of having improved energy and security using low-cost techniques for next-generation multi-core server systems.

- Chapter 2 provides the necessary background for this dissertation. Specifically, we introduce some hardware supports that can be used as the potential knobs for improving multi-core server energy efficiency. We explain why it is important to harness processor and server low-power states to achieve energy savings. We

then present the information security issues in multi-core server hardware and give background information on hardware information leakage in the forms of side/covert timing channels.

- Chapter 3 demonstrates three novel techniques that make use of processor and system low-power states to achieve enhanced energy efficiency while satisfying application's QoS constraints in multi-core server systems. Specifically, *Dual Delay Timer* makes intelligent use of system low-power states to achieve higher energy saving for latency-critical workloads. *WASP* performs *workload-adaptive* energy and latency optimization by jointly using processor and system low-power modes. Finally, *TS-Bat* applies efficient batching to enable processor's residency in the most energy-conserving state for optimized CPU energy savings. The evaluations on physical testbed show that the proposed techniques achieve substantial energy savings under various QoS constraints.

- In Chapter 4, we systematically investigate the information leakage vulnerabilities in multi-core architectures. We first illustrate a new covert timing channel that exploits cache access latency difference due to Non-uniform Memory Architectures, and propose a lightweight detection technique that uses statistical metrics to quantify the presence of such information leakage attacks. We further demonstrate cache coherence state-based covert channels where the adversaries transmit secrets by manipulating the coherence states for shared cache blocks. Characterizations are performed on the cache coherence state-based covert channels in terms of bit rates and bit accuracy. We finally propose slight modifications to the existing cache coherence protocol to annul the timing difference between different states and thus effectively thwart such covert timing channels.

- Chapter 5 presents the conclusions of this dissertation work and discusses future research directions.

## 1.3 Scope of this Dissertation

The main goal of this work is to provide understandings of the implications of processor/server hardware on energy efficiency and information security, and offer low-cost techniques that enhance them for the next generation multi-core server systems. We design effective techniques to tackle both issues by leveraging existing hardware supports and only introducing minimal architectural changes when necessary. We perform quantitative analysis of our proposed solutions mainly on real hardware and occasionally on simulators that model power and timing aspects of the hardware where necessary.

To improve multi-core server energy efficiency, we are interested in techniques that take advantage of processor and system low-power states, which are widely supported in commercial processors. We aim for system-level energy consumption that corresponds to both dynamic and static power usage of the server hardware. Our proposed low-cost techniques target multi-core multi-server systems that run latency-critical workloads with runtime QoS constraints. Such applications are extremely important in delivering many fundamental services to end users, including search, caching and web serving. Our works show promising results in improving server energy efficiency without violating user's performance constraints. For information security, we look into information leakage attacks in the form of covert timing channels. Our work investigates such information leakage vulnerabilities due to the *design of multi-core processor architecture*. We are interested in characterizing such vulnerabilities as well as constructing efficient mechanisms to detect and defend against adversaries that exploit these vulnerabilities. Our investigations reveal two new vulnerabilities, one associated with the NUMA configuration and another associated with the cache coherence fabric. Our proposed techniques utilize lightweight monitoring as well as slight modification to existing hardware that either identify or stop the covert timing channels. We envision that this dissertation will motivate researchers to contribute on designing efficient and practical solutions that improve the energy efficiency and information security to the next level in future multi-core server system design.

# Chapter 2 Background

In this chapter, we present the background for energy efficiency and information security problems in multi-core server systems. First, we introduce the challenges of energy efficient computing and motivate the use of server low-power states for energy efficiency optimization. Second, we clarify concepts of emerging information leakage attacks in multi-core processor architectures, and highlight the need to systematically understand information security vulnerabilities in multi-core hardware and carefully design solution techniques.

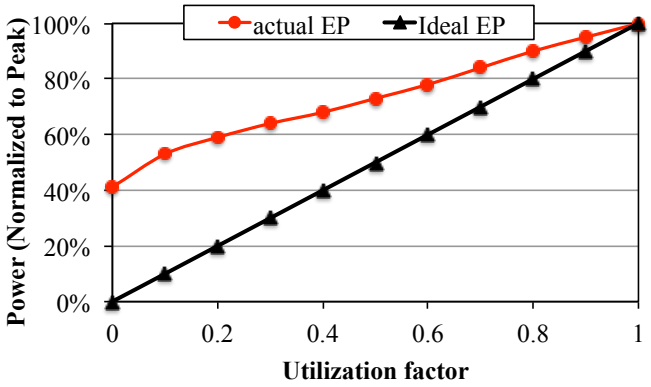



Figure 2.1: Actual server energy proportionality (*actual EP*) vs. ideal server energy proportionality (*ideal EP*)

## 2.1 Server Energy Efficient Computing

### 2.1.1 The Energy Inefficiency Problem

Demands for personalized and contextual retrieval of large volumes of data from the users have strongly driven the growth of server systems. The enormous server energy consumption has raised unprecedented concern globally. Such prohibitive energy cost largely results from lacks of sufficient energy efficiency in contemporary server infrastructure [101, 142]. Two major factors contribute to the energy efficiency issue. Firstly, energy proportionality over server utilization is extremely unsatisfying. Figure 2.1 shows the power usage for a typical multi-core server compared to a server with ideal energy proportionality. As shown in the figure, there exists a huge gap in

energy proportionality between a typical server (*actual EP*) and the ideal server (*ideal EP*). Notably, servers consume a large portion of the peak power at relatively low utilizations. Even when the server is completely idle, it still burns 40% of the peak power. This indicates that contemporary server hardware components are *inherently* energy inefficient. Server system energy inefficiency is further plagued by the second factor, *ineffective system-level resource management*. Many application workloads have QoS constraints that need to be satisfied. For latency-critical workloads, the QoS constraints are defined as the tail latencies [81] (e.g., $90^{th}$, $95^{th}$, $99^{th}$ percentile latency) which can be very sensitive to many runtime variabilities including server system configurations, job arrivals and workload characteristics. In order to guarantee QoS, most server systems are provisioned for the *peak demand*, and configured to operate at capacities much higher than necessary [47]. Therefore, servers are often underutilized and operated within the lowest energy efficiency region. Overall, a considerable amount of wasteful energy is consumed due to keeping server hardware (e.g., CPUs) in power-consuming mode and making an excessive number idle or only lightly used.

### 2.1.2 Improving Energy Efficiency with Server Low-power States

Prior studies on improving energy efficiency in server systems can be broadly classified into two categories: (i) cluster-level power management techniques that dynamically re-size server farms by dispatching workloads to a subset of servers and turning off the rest of the servers [55, 75, 129]; (ii) server-level dynamic power managements that leverage *Dynamic Voltage and Frequency Scaling* (DVFS) to minimize energy while not adversely affecting application performance [66, 82, 101, 102]. While cluster-level energy optimization strategies can potentially offer promising energy savings through cutting down platform power, they work at a coarser granularity and tend to be less useful for latency-critical workloads. DVFS is shown to be useful in saving processor *dynamic power*. However, DVFS-based mechanisms are less effective as *idle power* is becoming the primary contributor to the energy inefficiency in server systems. Additionally, with the trend of transistor scaling, the dynamic range

of voltage is largely shrinking, which further diminishes the usefulness of frequency and voltage scaling [108].

In recent years, hardware low-power modes are emerging as important features that are widely supported in server platforms. Processor low-power states reduce CPU power by clock-gating and power gating various core-level and uncore components such as function units, L1 caches and last level shared caches [73]. Meanwhile, servers nowadays support system sleep states that reduce the entire server power by putting server components (e.g., memory and disks) to various power conserving non-active states. Each low-power state is associated with a latency that is required to wake up the various system components from non-working state to the active state. Typically deeper low-power states have higher power savings but take longer timer to resume back to the working state [116]. Processor and system low-power states can significantly reduce idle power consumption (up to 90% [100]) compared to systems without enabling them. Therefore, low-power modes are *promising knobs* to improve server energy efficiency to *the next level*.

While taking advantage of situations where servers can enter low-power states can achieve considerable static and idle energy savings, several challenges exist in mechanisms that harness low-power states. *First*, improper use of low-power states can easily lead to application SLA violations. This is because jobs that arrive when servers are in low-power mode need to wait until they resume to active state. Long waiting/queuing time can severely deteriorate the job tail latency. Consequently, server low-power states need to be carefully controlled in order to improve energy efficiency while meeting QoS constraints. *Second*, due to the variability in job sizes, system utilizations and performance constraints, determining individual low power management policy in each scenario manually is time-consuming and cost-prohibitive. Therefore, exploring *autonomous mechanisms* that can automatically adapt to various workload characteristics and improve energy efficiency is essential. Finally, as the low-power states are numerous in server platforms (i.e., core-level, processor-level and system-level), selecting the most beneficial low-power states and effective coordinating software applications with them is also challenging. Our proposed tech-

niques [172, 174, 175] (as described in Chapter 3) address the aforementioned issues by (i) making novel use of low-power states for applications to achieve higher energy saving, (ii) designing an adaptive algorithm that optimizes server energy efficiency and autonomously adjusts its parameters based on the observed workload characteristics to meet QoS constraints, and (iii) developing a QoS-aware scheduling framework that judiciously integrates temporal batching and spatial batching to maximize processor's residency in the most power-conserving state for enhanced CPU energy savings.

## 2.2 Information Leakage in Multi-core Server Systems

The proliferation of critical and information-sensitive data has enticed adversaries to compromise server systems and carry out information leakage attacks. Data breaches have introduced tremendous economic loss every year [63]. Traditionally, there is a large body of works that study useful static/dynamic techniques to guard system against software-level exploitations [27, 58, 59, 76–78, 89, 90, 162, 169]. As software confinement mechanism continues to be more capable, adversaries begin to target shared hardware resources to carry out malicious behaviors, notably information leakage attacks.

Among the many forms of sensitive information leakage, timing channels are particularly notorious for their stealthy exfiltration of sensitive information by an insider process (termed as a *trojan* or *victim*) to a malicious spy at relatively high bit rates. Such attacks can manifest as either *side channels* where a benign victim *unknowingly* leaks sensitive data to a malicious spy, or as *covert channels* where a malicious insider trojan process *intentionally* colludes with a spy process to reveal secrets illicitly. Since the system security policy explicitly prohibits information transfer between the insider process with processes outside of the trusted domain, the adversaries cannot directly utilize software-level communication interfaces to perform information leakage. Therefore, to make the attack possible, they start to use timing channels. Timing channels rely on modulations of the access timings on shared hardware resources and do not leave any physical traces. Timing difference can be observed either due to contention on shared resources or through microarchitectural states that results in

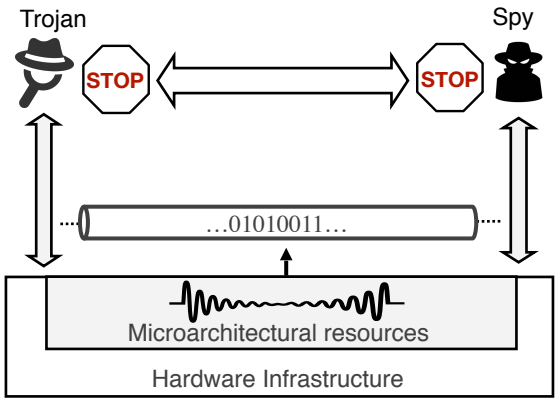



Figure 2.2: Information leakage attacks (covert channels) on shared hardware resources

distinct resource access latencies. These attacks have been demonstrated to successfully break security primitives in modern server systems such as the cloud [153].

Since both covert and side timing channels use timing modulation, we discuss covert channels without loss of generality. Figure 2.2 shows an illustration of an information leakage attack in the case of covert timing channel. Formally, Trusted Computer System Evaluation Criteria (TCSEC or Orange Book developed by US Department of Defense) [44] defines covert channel as any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy. Among the various types of covert channels, timing channels work by allowing a trojan process to signal information to a spy process by modulating its own use of system resources in such a way that the change in response time observed by the spy would provide information. TCSEC notes that covert channels with low bandwidths represent a lower threat than those with higher bandwidths. This is because, lower bandwidth channels become increasingly more expensive for the adversary with diminishing returns in terms of information gain (e.g., the adversary gets almost no useful or meaningful information on covert channels with bandwidth rate of 0.1 bits/sec or below). Based on measurements from several different computer systems, TCSEC classifies a high bandwidth covert channel to have a minimum rate of 100 bits/sec.

Several existing works have demonstrated the information leakage vulnerabili-

ties in various hardware components including private/shared caches, memory bus and branch predictors [46, 95, 115, 121]. The recent advancements in Spectre and Meltdown [84, 92] attacks have shown that microarchitectural timing channels are extremely dangerous and can be easily manifested in practical settings to exfiltrate sensitive information on mainstream hardware. The prevalence of multi-core processing and virtualization creates even larger attack surface by allowing many mutually distrusting parties to share the same resources.

Traditionally, a significant amount of attention has been directed to software security, and hardware security, especially information leakage in multi-core server architectures, is a new frontier yet to be thoroughly explored. Computer architects urgently need to understand information leakage vulnerabilities in hardware and how adversaries exploit them from an architecture perspective. To devise strategies to neutralize such efforts by the adversary, it is important for researchers to come up with techniques that are able to characterize, detect and thwart these attacks. More importantly, as processor vendors are usually cautious about incorporating new features into the existing design due to cost concerns, solution approaches that offer detection or prevention mechanisms for these information leakage attacks should not over-design the underlying architecture in order to make it affordable in realistic scenarios.

In our studies [168, 170, 171] (as described in Chapter 4), we investigate and present two new information leakage vulnerabilities in multi-core processor architectures. Two classes of information leakage attacks, namely the *NUMA-based (Non-Uniform Memory Access) covert channels* and the *coherence state-based covert channels*, are demonstrated on real multi-core server platforms. The first attack exploits the access timings across multiple levels of the cache hierarchy in NUMA-based architectures. While the second attack leverages the timings of load latencies corresponding to cache lines in difference coherence states (i.e., *Exclusive* and *Modified*). To protect systems against NUMA-based covert channels, we develop a lightweight statistical method that quantifies the presence of the attacks, and further discuss feasible mitigation techniques. To thwart cache coherence-state based attack, we pro-

pose slight modifications to cache coherence protocols that annul the read latencies between the two cache coherence states. Our proposed solutions can effectively guard multi-core servers from timing channel exploitations with low costs.

# Chapter 3 Low-cost Techniques for Improving Server Energy Efficiency

In this chapter, we show the detailed design of the three techniques that are used to improve the energy efficiency of server systems by leveraging low-power states. Specifically, we elaborate the details of processor- and system-level low-power states in Section 3.1. In Section 3.2, we demonstrate the design of a dual delay timer mechanism that orchestrates servers with system low-power states to achieve high server farm energy savings. In Section 3.3, we demonstrate WASP, a workload adaptive framework that leverages both processor and system low-power states to achieve optimized energy-latency tradeoff for latency-critical workloads. Section 3.4 demonstrates TS-Bat, a QoS-aware scheduling framework that performs temporal and spatial batching to achieve energy saving for multi-core processors by maximizing package-level low-power state residency. Finally, Section 3.5 and Section 3.6 present related work and summary of this chapter.

## 3.1 Understanding Server Low-power States

Emerging from embedded devices, low-power states are now an important feature targeted for power management in modern computer systems. The Advanced Configuration and Power Interface (ACPI) [65] provides a standardized specification for platform-independent power management. ACPI-defined interfaces have been adopted by several major operating system vendors [25, 141] and supported by various hardware vendors such as Intel and IBM [53, 71]. Hardware low-power states reduce the power envelope by clock gating and power gating various systems components such as processors cores, uncore resources, memories and peripheral devices.

Figure 3.1 illustrates the hierarchical power state management in ACPI. Specifically, ACPI uses global states, *Gx*, to represent states of the entire system that are visible to the user. Within each *Gx* state, there is one or more *system sleep states*, denoted as *Sx*. System sleep states define power status for various server components. For instance, *S0* is the working state that indicates the system is running and CPUs

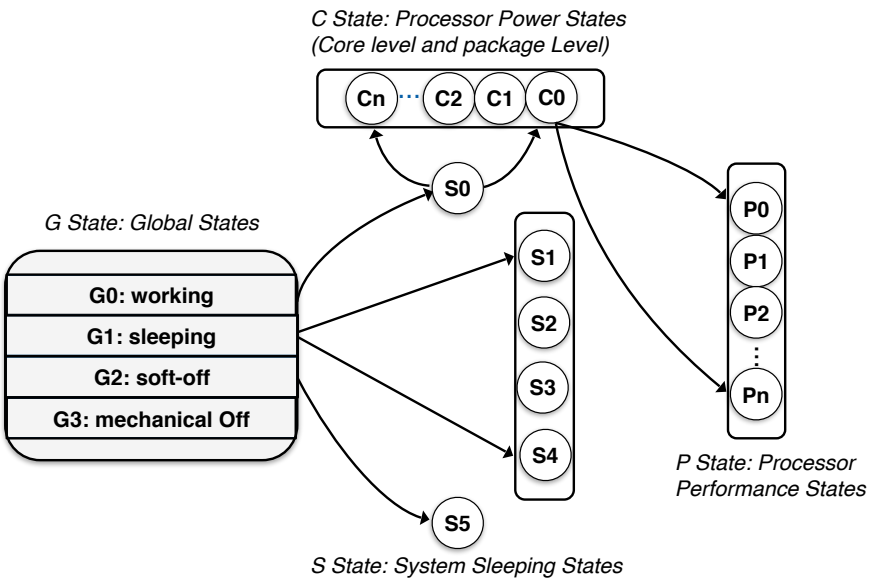



Figure 3.1: Hierarchy of server system low-power states and performance state.

are ready for executing instructions. Other $S$ states are non-active system states. *S3* is also known as the *Suspend to RAM* state where processors are turned off and all contexts are no longer maintained except memory subsystems. When the system sleep state is *S0*, the processor is allowed to reside in a set of $C$ states such as *C0*, *C1* and *C2*. C state enables fine-grained core and uncore components (e.g., private caches, shared caches and coherence fabric) low-power modes to achieve various level of power savings. Finally, when the processor core is active (C0), performance states can be configured to determine the speed of instruction execution at runtime (i.e., DVFS). In this work, we mainly target on server low-power states for energy efficiency optimizations for the reasons discussed in Chapter 2.

Modern processors generally provide high parallelism by integrating multiple cores within one processor package. Low-power C states are supported at both *core level* and *package level*. Core C state choices and residencies are determined by the Operating System (e.g., the *menu* CPU-idle governor in Linux [116]) based on applications' runtime activities. The package $C$ state is automatically resolved to the shallowest C state among all the cores. Waking up from package C state takes longer time than the same level of core C state since the un-core components have to be activated and warmed up before resuming the core execution contexts.

Low-power states can considerably reduce server system energy consumptions when the processor and system hardware are in the standby mode or underutilized. However, it is worth noting that the use of low-power states can introduce performance penalties due to the existence of delay (wakeup latency) for resuming servers in low-power states back to active. Typically, deeper low-power states indicate more aggressive energy savings but also corresponds to longer wake-up latency. For example, waking up processor from core C states takes microseconds while restoring servers from deep system sleep states (S states) can take upto a few minutes [81]. Therefore, judicious use of the above-discussed low-power states is necessary in order to trade off application performance (e.g., job latencies) for energy efficiency without violating user service level agreements.

## 3.2 A Dual Delay Timer Approach Through Leveraging System Sleep

Prior works have explored the use of processor idle and system deep sleep together with a single delay timer that controls the server's transition from idle state to system deep sleep to save energy for idle servers [55]. In our first proposed technique, we take advantage of processor idle and deep system sleep states combined with *dual delay timers* to orchestrate the entry and exit from system low-power states and maximize system energy savings for several workloads.

### 3.2.1 Server, Job, and Workload Model

We model the server farm as a multi-server system of homogeneous servers. Each server is equipped with an Intel Xeon processor that can process multiple jobs at a time (corresponding to the level of parallelism in multi-core processors). We use four synthetic workloads with average workload execution times shown in brackets next to them: Google search (4.2 ms), Apache (75 ms), Mail (92 ms) and DNS query (194 ms) jobs [108], and one real system trace, Wikipedia, with an average workload execution time of 3.5 ms [130]. The term utilization factor $\rho$ is defined as the fraction of time the server is expected to be busy executing jobs.

A system-wide load balancer is configured to dispatch jobs to servers. The *job latency* is defined as the time elapsed from when a job arrives to when the job completes its execution and departs the server farm. We monitor the average job latency (normalized with respect to the average expected execution time) and ensure that the worst case job latency is within bounds to meet QoS constraints.

### 3.2.2 Motivation

Sleep states have been implemented in most modern processors to reduce energy consumption, especially when the processor utilization levels are low. For example, when the processor operates at 10% of its peak capacity, the processor should ideally stay active for 10% of the time. For the remaining 90% of the time, the system should enter one of the low-power states or sleep states (that consumes significantly less power compared to the active mode), and ultimately approach energy proportionality. While sleep states are designed to lower the system energy consumption, one has to use them wisely in order to take advantage of their energy-saving benefits. To illustrate this, we perform motivational experiments that study the energy consumption of server farms under different workloads. The experiment is performed on a simulator that models a server farm with 50 four-core servers (See Section 3.2.5 for more details). Three different server power configurations are implemented as described below.

1. A-I is a power configuration where the server alternates between active ($C0S0_a$) and idle states ($C0S0_i$). A server is active when at least one of the four cores within the server has a job to process. The server enters idle state if none of its cores has a job to process.

2. A-I-S ($\tau = 0$) is a power configuration where the server transitions between three states – active ($C0S0_a$), idle ($C0S0_i$), and system sleep ($G1S3$). $\tau$ denotes the *delay timer* for transition from processor idle to system sleep state (i.e., S state). The server is active when at least one of the four cores within the server has a job to process. The server enters the idle state when none of the cores within the server have jobs to process, and then, it immediately goes to deep sleep since the delay timer $\tau$ is set to zero. In other words, the server goes to deep sleep immediately whenever there

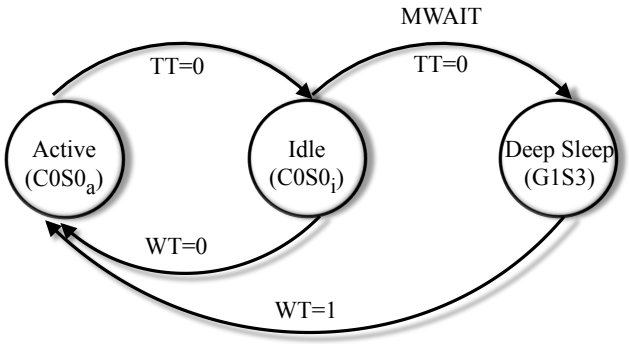
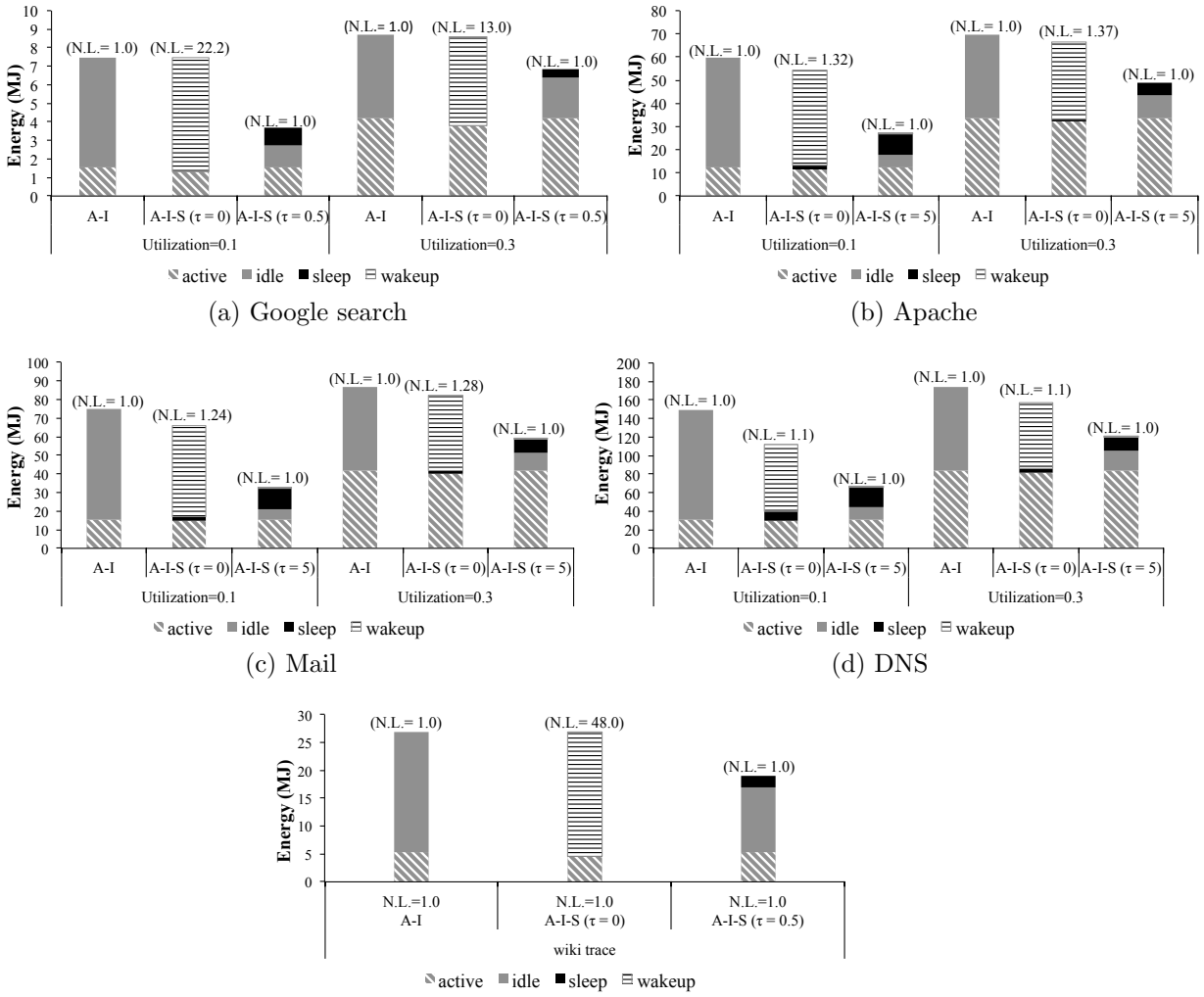



Figure 3.2: Server power states and their corresponding Transition (TT) and Wakeup (WT) times (in seconds).

are no jobs to be processed by the server. Figure 3.2 illustrates the transitions among the three states along with the corresponding transition times and wakeup times.

3. A-I-S ($\tau = c$) is a power configuration that is similar to the above, except that the server goes to deep sleep from idle after a delay timer expires. In other words, the server waits for $\tau = c$ seconds before transitioning to deep sleep state after entering the idle state. If a job arrives before the delay timer reaches zero, the server gets back to active state. The optimal $\tau$ value is chosen based on an exponential sampling of $\tau$ values and the associated energy savings. We note that prior studies [57] and [56] have also considered timer-based strategies to conserve energy. In [56], the authors used delay timers to transition between hypothetical sleep states, and in [57], the servers are completely switched off to conserve energy. As we show later (Section 3.2.6), our new dual timer strategy significantly improves the energy savings over the single delay timer approach.

Figure 3.3 shows the results of our experiment for various workloads. In each workload, we study the A-I, A-I-S ($\tau = 0$) and A-I-S ($\tau = c$) configurations for two different processor utilization levels of 0.1 (low server utilization in server farm) and 0.3 (average server utilization in server farm observed by Barroso et al. [47]). The corresponding energy consumption (in millions of Joules) are shown on the y-axis with breakdown between processor active, idle, sleep, and wakeup cycles. Average normalized job latency is shown above each power configuration at a certain server utilization level. We note that in a majority of cases, A-I-S ($\tau = 0$) configuration does




(a) Google search

(b) Apache

(c) Mail

(d) DNS

(e) Wikipedia

Figure 3.3: Energy breakdown for various workloads using three different server power configurations. Average Normalized Latency (N.L.) is shown above the bars normalized to the workload execution times.

not significantly improve energy, and in fact, the idle energy spent in A-I configuration is simply translated into wakeup energy for the processor to transition from sleep to active state in A-I-S ($\tau = 0$). This phenomenon actually diminishes the effectiveness of sleep state to save energy. Worse still, the latency impact of using A-I-S ($\tau = 0$) is extremely high, especially for workloads with short execution times such as Google search where we observe 22× performance slowdown at a utilization level of 0.1, 13× performance slowdown at a utilization level of 0.3, and Wikipedia with a 48× performance slowdown.

For A-I-S ($\tau = c$) configuration where $c$ is the optimal $\tau$ value, we find the following optimal values for $\tau$: $\tau = 0.5$ seconds for short latency jobs such as Google search and Wikipedia, and $\tau = 5.0$ seconds for long latency jobs including Apache, Mail and DNS. In general, we note that A-I-S ($\tau = c$) configuration significantly reduces the overall energy consumption while keeping the average normalized job latency to be almost the same as that for A-I configuration. Through our experiments, we measured the normalized job latency to be 1.0 in A-I-S ($\tau = c$). The following are our key observations:

1. At average server utilization levels of 0.1, we observe as much as 55.4% energy reduction for Apache and up to 55.7% energy reduction in Mail workloads compared to their corresponding A-I configuration. We note that such significant energy savings are possible because a majority of servers now enter deep sleep state (consuming 88.4% less power than active mode). Most sleeping servers are rarely woken up and remain in deep sleep mode due to the incoming jobs being serviced by the servers in the idle/standby mode with a delay timer $\tau = c$.

2. At average server utilization levels of 0.3, we observe less energy savings compared to utilization level of 0.1 due to a higher rate of incoming jobs. We measure about 32.4% energy reduction in Mail and 30.1% energy reduction in DNS workloads compared to the corresponding A-I configuration. Beyond active energy (that is spent on actually servicing the jobs), we note that servers remain idle most of the time, thus leading to overall energy reduction.

3. In Wikipedia workload trace, we observe 29.2% reduction in A-I-S ($\tau = 0.5$) compared to A-I configuration.

### 3.2.3 Dual Delay Timers Design Overview

In Section 3.2.2, we observed that A-I-S ($\tau = c$) is able to achieve significant energy savings over A-I while having similar performance in terms of job latency. However, when the system utilization is low (say 0.1), short latency workloads such as Google search still consume around 30% of the peak energy. This is due to many servers still remaining in the idle mode after they are done servicing their jobs.

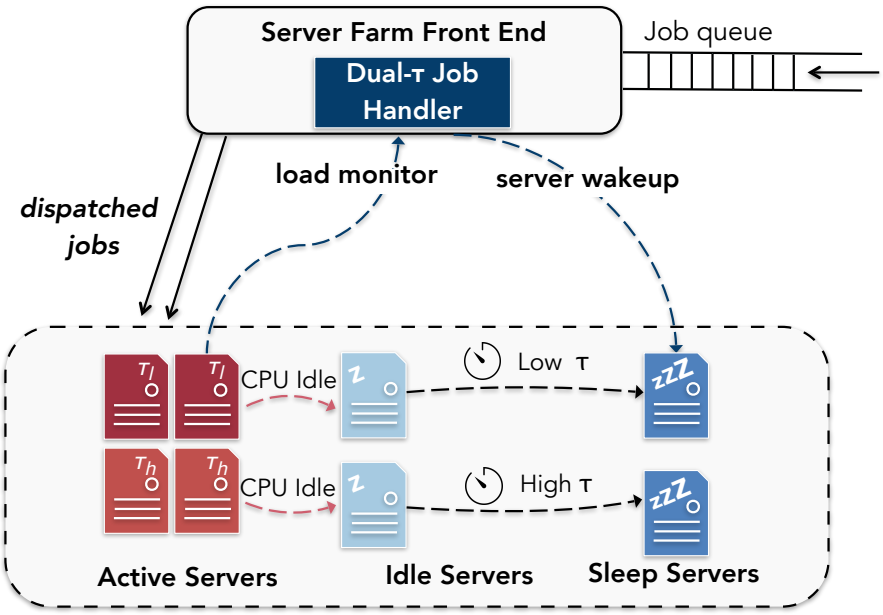



Figure 3.4: Dual Delay Timer Framework Overview

| Symbol | Description |
|---|---|
| $V_{ai}$ | set of servers in active or idle state |
| $V_s$ | set of servers in deep sleep |
| $\tau_h$ | high $\tau$ value |
| $\tau_l$ | low $\tau$ value |
| $t_w$ | threshold for waking up a sleeping server |

Table 3.1: Notations in the Dual Delay Timer Algorithm

To explore the possibility of further savings in system energy consumption, we devise *Dual Delay Timers* or *Dual-$\tau$*. Dual-$\tau$ is based on the following intuition: instead of keeping $\tau$ values to be the same for all the servers, further energy reduction could potentially be had by grouping the servers into two pools: one pool of servers with relatively high $\tau$ ($\tau_h$) that offers to be the standby machines for the incoming jobs, and the second pool of servers with low $\tau$ values ($\tau_l$) which can quickly go to deep sleep state. The best case scenario is when a small number of servers with high $\tau$ continue to stay in the standby mode to maximize the chances of accepting all of the incoming jobs, and the rest of servers with low $\tau$ quickly go to sleep resulting in optimizing the overall energy consumption. Figure 3.4 demonstrates the overview of the Dual Delay Timer design.

**Algorithm 1:** Dual Delay Timer Algorithm

```
   Input: t_w, n (total number of servers)
1  Initialization: V_ai = {s_1, s_2, ..., s_n};
   /* By default, all servers are placed in V_ai and set into idle state
      */
2  for i in [1, n] do
3  |  power state of s_i ← idle
4  end
   /* arrived jobs are first placed in the queue                        */
5  while there are unfinished jobs in queue do
6  |  pick a new job j from the head of the queue;
7  |  if at least one server with a free core exists in V_ai then
8  |  |  find set of servers S in V_ai with highest utilization;
9  |  |  if multiple servers exist in S then
10 |  |  |  give preference to a server with τ_h;
11 |  |  |  randomly pick a server, s_sch in S for scheduling;
12 |  |  end
13 |  |  schedule job j on a free core from s_sch;
14 |  |  continue;
15 |  end
16 |  else
17 |  |  compute the number of pending jobs in queue, p;
18 |  |  if  p > t_w then
19 |  |  |  pick a server s_sch from V_s;
20 |  |  |  give preference to a server with τ_h;
21 |  |  |  add s_sch to V_ai;
22 |  |  |  set s_sch to active state;
23 |  |  end
24 |  end
25 end
```

### 3.2.4 Dual Delay Timer Algorithm

The power management policy for Dual Delay Timer augments the A-I-S configuration by designating a small fraction of the server pool with high $\tau$ values, while others have low $\tau$ values.

Algorithm 1 presents our Dual Delay Timer approach that accepts the incoming jobs and assigns them to servers. The corresponding notations are listed in Table 3.1. Our Dual Delay Timer approach designates a small set of servers with high $\tau$ and the rest of the servers with a low $\tau$. The candidate servers in the two pools are rearranged periodically based on load balancing and fairness among all of the servers.

When a new job arrives, the job handler will first try to assign this job to one of the schedulable servers, which could either be an active server with available cores or a server in idle state, denoted as $V_{ai}$. The algorithm will choose a server from $V_{ai}$ with the highest utilization. This is done to favor the less utilized servers to enter idle state rapidly. If there are multiple servers with high utilization, a server with $\tau_h$ is prioritized in order to favor the servers with $\tau_l$ to enter idle state state rapidly. If none of the servers in $V_{ai}$ has an idle core, the incoming job is queued and a server in sleep state is woken up to enter active state. However, waking up a sleeping server every time when a job is queued can be suboptimal since the transition penalty for an asleep server is high. To address this issue, we add a simple threshold $t_w$. Only when the current number of pending jobs in queue is greater than $t_w$ would a server in deep sleep be woken up. The $t_w$ threshold is a tunable parameter that guides how conservatively the job handler wakes up a server. Throughout our experiments, $t_w$ is set to be the product of the number of cores per server and the number of $\tau_h$ servers; This effectively avoids unnecessary server wake-ups. Note that our algorithm assumes that the servers are capable of processing one job per core at a time without loss of generality. If the core is capable of running multiple jobs concurrently due to techniques like Simultaneous Multi-Threading, we could assign multiple jobs per core until it is fully occupied.

### 3.2.5 Experimental Setup

#### *3.2.5.1 Power Model*

In this work, we use the low-power states from the Intel Xeon E-5 processor [71]. Table 3.2 shows the power model illustrating the power consumption by various units in a given low-power mode. Note that the $C$ states mentioned in our power model refer to processor level $C$ states.

| **Components** | Active $C0S0_a$ | Idle $C0S0_i$ | Deep Sleep $G1S3$ |
|---|---|---|---|
| Cores [71, 100] | 130 W | 75 W | 16 W |
| Chipset [100] | 7.8 W | 7.8 W | 7.8 W |
| RAM [100] | 23.1W | 10.4 W | 3.0 W |
| HDD [100] | 6.2 W | 4.6 W | 0.8 W |
| NIC [100] | 2.9 W | 1.7 W | 0.5 W |
| PSU [100] | 70 W | 35 W | 1 W |
| Cooling [100] | 10 W | 1 W | 0 W |
| Total Power | 250 W | 135.5 W | 29.1 W |

Table 3.2: Power breakdown for an Intel Xeon-E5 based server.

#### *3.2.5.2 Simulation of Server Farms*

We have built an in-house event-driven simulator to analyze the energy savings of our approach. Even though several well-known data center and cloud simulators exist (such as [26, 109]), they do not fit our needs: we need a simulator that provides fine-grained modeling of sleep states and control of sleep state transitions as well as a basic framework to manipulate server power states in a centralized manner. Moreover, to analyze a wide range of workloads and applications, the simulator also needs to accept both synthetic workloads and realistic workloads from system traces. Our simulator has three major components: *workload generator*, *server power state/performance manager*, and *server farm job handler/load balancer*. The workload generator injects jobs into the system based on either a stochastic process (for synthetic workloads such as Google search, Apache, Mail, and DNS) or system job arrivals/service time traces collected from realistic data centers (for Wikipedia workload). The *power state manager* models various sleep states and is responsible for coordinating the servers to enter or wake up from a specific sleep state. For example, the *power state manager* can request a server to enter deep sleep (*G1S3* state) immediately or after a certain delay timer value. Our simulator reports detailed statistics including the breakdown of server energy and performance measures such as the job latency characteristics.

#### *3.2.5.3 Simulation Configurations*

To exploit the use of sleep states for energy optimization, we simulate a server farm with 50 four-core servers. In the rest of the Section (for Dual-$\tau$), unless otherwise noted, we assume this to be our default configuration. We configure the duration of our simulations to be long enough for the type of workload – for example, the simulation length for DNS workload (with largest job size) to be 20,000 seconds, which means roughly 40,000 jobs would be processed per server on average at server utilization of 0.1 and an even higher number of jobs at higher utilization levels. We then scale the execution time for other workloads based on their job sizes. As a result, Mail, Apache, and Google search would have simulation times of 10000, 8000, and 1000 seconds, respectively. These simulation lengths are also configured to ensure that enough number of bursty and non-bursty phases would be observed in our experiments where we model non-uniform job arrivals (Section 3.2.6.2). Also, in all of our experimental results, we report the steady state statistics by disregarding the warm-up time period during the first 10000 jobs arrivals.

#### *3.2.5.4 Workload Generation*

We use three types of workload arrival models. By default, we use Poisson Process for job arrivals, which is widely used in prior works to model data center workloads [55, 108]. To model bursty patterns in job arrivals that are also typically seen in data center environments, we use Markov Modulated Poisson Process, a well studied model to simulate workload burstiness [24, 28, 120]. Aside from such analytical models, we also use Wikipedia workload, a realistic system trace available for public use [130]. For the first two cases, we simulate three different levels of system utilizations (0.1 for low utilization, 0.3 for average utilization [47], 0.6 for high utilization).

*Poisson-based job arrivals*: The job service times are modeled as a uniform distribution with a mean service time, $1/\mu$, where $\mu$ is the service rate of a server. Uniform distribution, rather than the usual exponential distribution, is assumed to prevent the workload generator from producing very short jobs that can severely deteriorate

job latencies at the tail end. In a multi-core based server farm, the relation between system utilization $\rho$ and job arrival rate $\lambda$ is: $\rho = \frac{\lambda}{\mu * nServers * nCores}$, where $nServers$ is the number of servers and $nCores$ is number of cores per server.

*MMPP-based bursty job arrivals*: *MMPP* uses a continuous-time Markov chain to model different stages or states of the workload. Each state $x$ corresponds to a Poisson Process with job arrival rate $\lambda_x$. By orchestrating the transitions among various states with high and low $\lambda s$, *MMPP* is able to model workload *burstiness* at a finer-grain level. In our experiments, we use a 2-state *MMPP* model, in which one state has a high job arrival rate $\lambda_h$ representing periods of bursty arrivals, and the other state has a low arrival rate ($\lambda_l$) and models non-bursty periods of operation. There are two approaches to tune the levels of burstiness – increasing the ratio of job arrival rates between bursty and non-bursty state, $R_a = \lambda_h/\lambda_l$, or decreasing the proportion of time the process stays in bursty state. Detailed exploration of workload burstiness modeling is a rich area of study [24]. In our experiments to characterize burstiness, we define the ratio between $\lambda_h$ and $\lambda_l$, as well as the ratio between process durations. The job arrival rates are then translated to different utilization factors. $\lambda s$ in both states are computed and set so that the bursty workloads generated would have an average utilization factor of 0.1, 0.3, and 0.6, respectively. This is done to compare our results to Poisson-based workloads with the same system utilization factors. Table 3.3 illustrates the high $\rho$ and low $\rho$ (corresponding to the two states of the *MMPP* model) associated with different average utilization levels.

#### *3.2.5.5 Job Handler*

Our job handler uses the following algorithm: First, we check if an active server has an idle core. If so, the job handler schedules the job on a server with least number of idle cores. If multiple such servers exist, the handler picks one of them randomly and schedules the job on one of its cores. Second, if none of the active servers have any available cores to accommodate an incoming job, we check if an idle server exists. If so, we randomly wake up one of the idle servers and schedule the job on one of its cores. Third, if there are no active or idle servers to accommodate an incoming job,

| Utilization levels | High $\rho$ | Low $\rho$ | Window length |
|---|---|---|---|
| *0.1* | 0.4 | 0.025 | 30 seconds |
| *0.3* | 0.7 | 0.2 | 30 seconds |
| *0.6* | 0.8 | 0.4 | 30 seconds |

Table 3.3: *MMPP* $\rho$ values for bursty and non-bursty periods to achieve a certain *overall* system utilization level.

we check if a sleeping server exists. If so, we randomly wake up one of the sleeping servers and schedule the job on one of its cores. Fourth, if there are no available servers, the job is buffered until a server becomes available.

### 3.2.6 Evaluation

#### *3.2.6.1 Exploration of Dual Timers for Poisson Job Arrivals*

The parameter exploration space for Dual-$\tau$ is large due to combinational exploration of two $\tau$ values and the partitioning of servers into two $\tau$ categories. To reach the solution faster, we first conduct a uniform sampling of the three major parameters: high $\tau$, low $\tau$, and the number of servers with high $\tau$. Due to space constraints, we are unable to present all of our results. We summarize a few important observations from our experiments below:

1. The number of high $\tau$ servers that maximizes energy savings is approximately $\rho$ * total number of servers. This finding confirms our intuition that when using dual $\tau$, the system is able to utilize a minimal number of active/standby servers while putting the majority of servers to deep sleep.

2. Having relatively large high $\tau$ values and setting low $\tau$ to zero (the server immediately goes to sleep state when idle) maximizes energy savings at all utilization levels relative to A-I and A-I-S ($\tau = c$) configurations.

Table 3.4 summarizes the results of our experiments. We show the energy reduction with Dual Delay Timers compared to A-I and A-I-S along with the normalized percentile job latencies. We note that dual $\tau$ can achieve further energy savings of up to 16.7% beyond the A-I-S (optimal $\tau$) especially at server utilization levels of 0.1.

| | | Dual-$\tau$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| WD | Util. | Saving over A-I | Saving over A-I-S (opt $\tau$) | 50%-ile N.L. | 90%-ile N.L. | 95%-ile N.L. | High $\tau$ | Low $\tau$ | Servers# with High $\tau$ |
| Google | 0.1 | +61.05% | +21.49% | 1.00 | 1.12 | 1.23 | 0.50 | 0 | 6 |
| Apache | | +63.90% | +16.71% | 1.00 | 1.18 | 1.30 | 1.0 | 0 | 6 |
| Mail | | +63.90% | +15.35% | 1.00 | 1.18 | 1.29 | 1.0 | 0 | 6 |
| DNS | | +63.90% | +15.37% | 1.00 | 1.18 | 1.29 | 10.0 | 0 | 6 |
| Google | 0.3 | +39.86% | +23.74% | 1.00 | 1.00 | 1.03 | 5.0 | 0 | 18 |
| Apache | | +41.48% | +14.40% | 1.00 | 1.07 | 1.13 | 5.0 | 0 | 17 |
| Mail | | +41.49% | +12.89% | 1.00 | 1.07 | 1.13 | 5.0 | 0 | 17 |
| DNS | | +39.86% | +10.05% | 1.00 | 1.03 | 1.13 | 5.0 | 0 | 18 |
| Google | 0.6 | +11.15% | +10.08% | 1.00 | 1.10 | 1.16 | 5.0 | 0 | 32 |
| Apache | | +18.81% | +11.75% | 1.00 | 1.10 | 1.16 | 5.0 | 0 | 32 |
| Mail | | +18.95% | +11.11% | 1.00 | 1.10 | 1.16 | 5.0 | 0 | 32 |
| DNS | | +17.89% | +9.82% | 1.02 | 1.29 | 1.40 | 5.0 | 0 | 32 |

Table 3.4: Energy reduction for Dual-$\tau$ in various workloads and dual delay timer values compared with A-I and A-I-S (opt $\tau$: lowest energy). Job arrivals are modeled as Poisson Process, and Normalized Latencies (N.L.) are calculated with workload execution times as baselines.

#### *3.2.6.2 Exploration of Dual Timers for MMPP Job Arrivals*

We conduct experiments for *MMPP*-based workloads and utilize a Markov chain-based predictor for burstiness detection.

We perform parameter space exploration for $\tau$ values and number of servers with high $\tau$. Table 3.5 summarizes the results of our experiments demonstrating the savings in energy over A-I and A-I-S (optimal $\tau$: lowest energy) and the corresponding $\tau$ values and the number of servers with high $\tau$. From the table, we can see that our Dual Delay Timer is able to save nearly 25% energy over optimal A-I-S configuration with single $\tau$. Also, the number of active servers (with high $\tau s$) are now much smaller than the expected $\rho$ * number of servers. This is because of short periods of high server utilization levels after which even the active servers can go to deep sleep without affecting performance. Unlike Poisson-based job arrivals, *MMPP* workloads have a small, non-zero low $\tau$ value to service the jobs during bursty phases.

#### *3.2.6.3 Exploration of Dual Timers for Wikipedia Trace*

We performed exploration of dual delay timer values $\tau$ for real world workload traces from Wikipedia. Table 3.6 shows that setting low $\tau = 0.002$ seconds and high $\tau = 5$ seconds reduces system energy consumption by 31% over A-I-S (optimal $\tau$) configuration. Note that the Wikipedia trace is a slowly varying workload with ultra

| | | Dual-$\tau$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| WD | Util. | Saving over A-I | Saving over A-I-S (opt $\tau$) | 50%-ile N.L. | 90%-ile N.L. | 95%-ile N.L. | High $\tau$ | Low $\tau$ | Servers# with High $\tau$ |
| Google | 0.1 | +62.33% | +23.95% | 1.00 | 1.05 | 1.10 | 5.00 | 0.05000 | 2.00 |
| Apache | | +62.26% | +17.30% | 1.00 | 1.14 | 1.24 | 5.00 | 0.20 | 2.00 |
| Mail | | +62.17% | +17.74% | 1.00 | 1.18 | 1.31 | 5.00 | 0.20 | 2.00 |
| DNS | | +61.96% | +15.64% | 1.00 | 1.11 | 1.19 | 5.00 | 0.50 | 2.00 |
| Google | 0.3 | +38.92% | +21.94% | 1.00 | 1.03 | 1.07 | 5.00 | 0.05 | 8.00 |
| Apache | | +40.18% | +13.00% | 1.00 | 1.20 | 1.36 | 5.00 | 0.20 | 7.00 |
| Mail | | +39.84% | +13.78% | 1.00 | 1.29 | 1.57 | 5.00 | 0.20 | 7.00 |
| DNS | | +40.36% | +12.64% | 1.00 | 1.15 | 1.27 | 5.00 | 0.50 | 8.00 |
| Google | 0.6 | +15.95% | +15.54% | 1.00 | 1.01 | 1.05 | 5.00 | 0.05 | 18.00 |
| Apache | | +18.32% | +9.05% | 1.00 | 1.12 | 1.18 | 5.00 | 0.20 | 18.00 |
| Mail | | +18.23% | +9.08% | 1.00 | 1.15 | 1.24 | 5.00 | 0.20 | 18.00 |
| DNS | | +18.19% | +8.46% | 1.00 | 1.18 | 1.28 | 5.00 | 0.50 | 18.00 |

Table 3.5: Energy reduction for Dual-$\tau$ in various workloads and dual delay timer values compared with A-I and A-I-S (opt $\tau$: lowest energy). Job arrivals are modeled as *MMPP*, and Normalized Latencies (N.L.) are calculated with workload execution times as baselines.

| | | Dual-$\tau$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| WD | Util. | Saving over A-I | Saving over A-I-S (opt $\tau$) | 50%-ile N.L. | 90%-ile N.L. | 95%-ile N.L. | High $\tau$ | Low $\tau$ | Servers# with High $\tau$ |
| Wikipedia | N/A | +71.20% | +31.36% | 1.00 | 1.37 | 1.53 | 5 | 0.002 | 2 |

Table 3.6: Energy reduction for Dual-$\tau$ in Wikipedia trace for dual delay timer values compared with A-I and A-I-S (opt $\tau$: lowest energy). Normalized Latency (N.L.) is calculated with workload execution time as baseline.

low utilization (ranging from 5% to 10%). Detailed statistics in our experiments showed that under such scenario Dual Delay Timer is able to maintain the exact number of servers to be active without waking up even a single server prematurely after a short warm-up period. The results again show that Dual Delay Timer is especially effective under low system utilization levels.

### 3.2.7 Scalability with Number of Servers

In this section, we study the energy reduction benefits of Dual Delay Timer strategy by varying the number of servers. We adopt the same procedure for parameter exploration as we did in Section 3.2.3 while exploring the optimal dual $\tau$ values that minimize energy consumption. We repeat the experiments for three different numbers of servers: 20, 50, and 100. Figure 3.5 shows energy savings of our Dual Delay Timer strategy compared with the corresponding A-I configuration. For each server farm size, the four synthetic workloads are simulated under three utilization levels of 0.1,

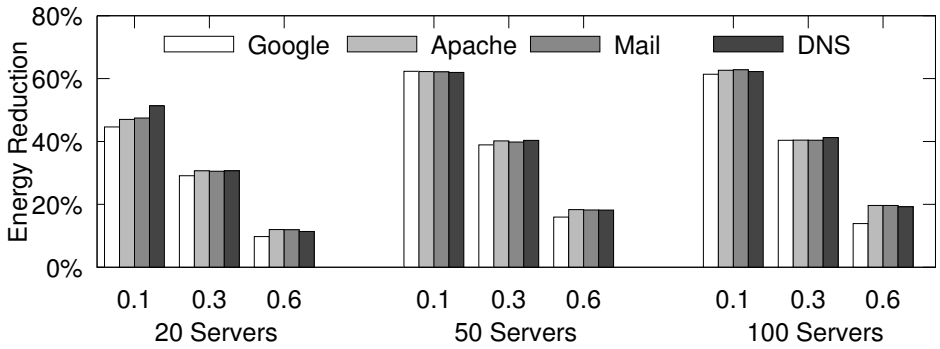



Figure 3.5: Energy reduction of Dual-$\tau$ compared to A-I in various workloads and server utilization levels for different numbers of servers – 20, 50, and 100.

0.3, and 0.6. From the scalability trend, we can see that when the server farm size increases from 20 to 50, the relative energy savings for all workloads increase at the scale of 10% to 20%. With 100 servers, the energy saving benefits are slightly better in comparison to 50 servers, and the benefits are significantly higher against the corresponding A-I configuration. For example, at server utilization of 0.1, energy saving benefits of dual delay timer is about 50%, and at server utilization level of 0.3, the corresponding energy savings are well over 30% for all of the workloads. In summary, about 45% to 63% energy is saved at average server utilization 0.1 depending on the size of the server farm, 28% to 40% energy is saved at server utilization of 0.3, and 10% to 20% energy is saved at server utilization of 0.6. Our mechanism shows good scalability and energy saving potential as server farm size increases.

## 3.3 WASP: Workload Adaptive Energy-Latency Optimization in Server Farms using Server Low-power States

In this section, we present WASP, a system framework that adaptively adjusts configuration parameters to achieve optimized energy-latency tradeoffs.

### 3.3.1 Motivation

The Dual-$\tau$ technique as discussed in Section 3.2 has demonstrated that the use of server low-power modes is promising in improving server system energy efficiency while still meeting the QoS constraints. However, we note that it is possible to build

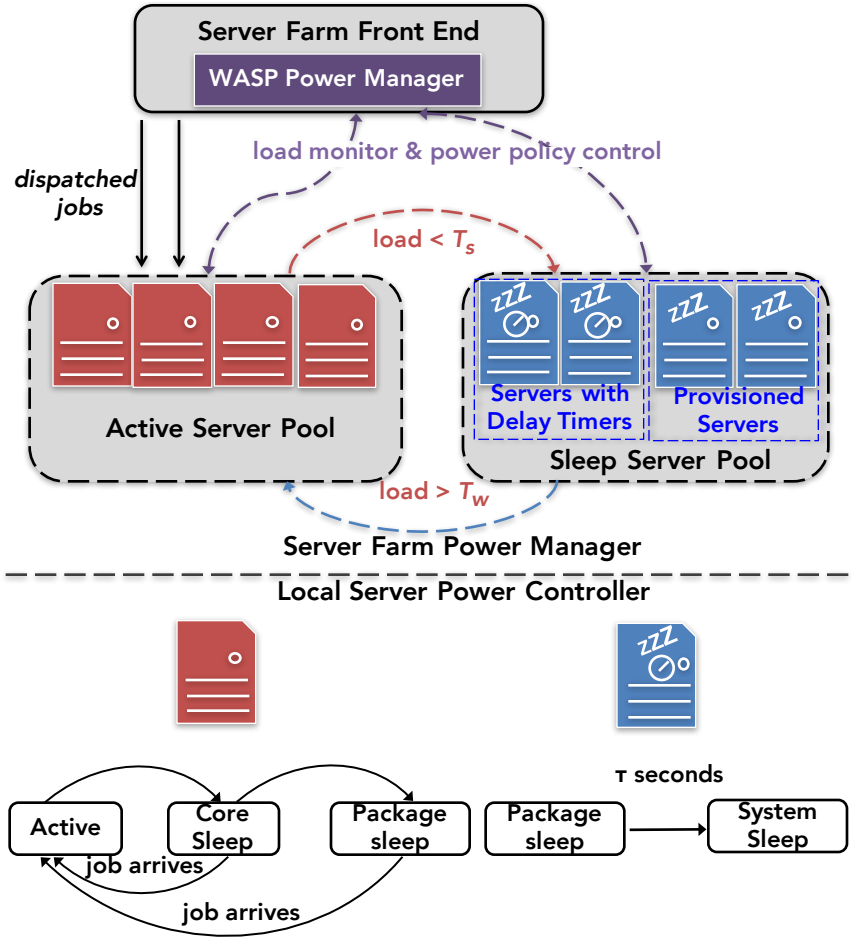



Figure 3.6: WASP Power Management Framework

even more effective energy optimization techniques by considering the following two capabilities : (i) adjusting system configurations to adapt to different workloads as well as distinct QoS constraints. (ii) using multiple processor and system low-power states simultaneously to achieve better energy and latency tradeoff. We therefore developed WASP, an energy optimization framework for server farms that is able to adapt itself to meet the performance demands while minimizing the system energy through *adaptively* adjusting its parameters based on the observed application characteristics such as job size, arrival pattern and system utilization levels.

### 3.3.2 WASP Design

To incorporate workload-awareness, we explore a two-level adaptive strategy that controls the active and low-power state transitions using a local server power con-

| Symbol | Description |
|---|---|
| $V_{act}$ | servers in active/package sleep mode |
| $V_s$ | servers in system sleep mode |
| $T_s$ | workload threshold to reduce active servers |
| $T_w$ | workload threshold to increase active servers |
| $N_p$ | number of provisioned shallow sleep servers |
| $\tau$ | delay time before entering system sleep |

Table 3.7: Notations in WASP power management algorithm

troller and a global server farm power manager. We now present the design for our WASP framework. As shown in Figure 3.6, the server farm power manager in the front end monitors the current load (number of pending jobs per server) and sends control commands to the local power controller. The server farm power manager puts the servers in either active or sleep modes. The bottom part of Figure 3.6 presents state transitions coordinated by the local server power controllers. Algorithm 2 and 3 illustrate the control algorithms for WASP power manager and the local power controller, respectively.

#### *3.3.2.1 Workload-adaptive Algorithm*

WASP automatically activates servers when the pending load becomes too high (that could lead to higher average job latency), and then places servers in low-power sleep mode to conserve energy when the workload becomes light. We achieve our goal of balancing energy consumption and latency by estimating the current load and placing servers in different power modes. There are two important parameters that govern transitioning between active and sleep modes: 1. $T_s$, workload threshold per active server below which WASP will put an active server to sleep, and 2. $T_w$, workload threshold per active server above which WASP will wake up an inactive server.

**Global Server Farm Power Manager**: All servers are initially in the shallow low-power state, and arriving jobs are placed in the server's local job queue. As jobs arrive, the *load per active mode server* is computed dynamically by the power

**Algorithm 2:** Global Server Farm Power Manager

```
Input: T_s, T_w, N_p, τ, n (total number of servers)
Initialization: V_act = {s_1, s_2, ..., s_n}; V_s = {};
while there are unfinished jobs do
    if a new job j arrives at time t_a then
        compute load_per_active_server;
        if load_per_active_server > T_w and |V_s| > 0 then
            retrieve a server s from V_s;
            V_act.add(s);
            create a trans_to_active_mode request tta_r;
            send tta_r to server s's power controller;
    if a job j finishes at time t_d then
        compute load_per_active_server;
        if load_per_active_server < T_s and |V_act| > 0 then
            retrieve a server s in V_act;
            V_s.add(s) ;
            create a trans_to_sleep_mode request tts_r;
            if count of shallow sleep servers> N_p then
                tts_r.enableDelayTimer(τ);
            else
                tts_r.enableDelayTimer(infinity);
            send tts_r to server s's power controller;
```

manager in the front end based on the number of jobs sent to individual servers and the completed jobs. The global server farm power manager maintains lists of servers in active and sleep modes. When new jobs arrive, it first checks if current load per active server is above $T_w$. If so, it selects a server in sleep server pool (if available) and sends a power mode transition request to the active state for that server. When the load per active server falls below $T_s$, the power manager selects an active server and sends a power mode transition request to enter sleep mode. Algorithm 2 describes the WASP power manager with its notations shown in Table 3.7.

**Local Server Power Controller:** The processor transitions to package sleep state when it becomes idle, and stays in that state until it receives the request to wakeup from the global power manager. If the server receives a request for transition to sleep mode, it will first finish up all the pending jobs in the local queue and then enters package sleep after which a delay timer is started. The server enters system

**Algorithm 3:** Local Server Power Controller

```
Input: τ (number of pending jobs in local buffer)
Initialization: local server power state ← package sleep;
current power mode pmod ← active mode;
delay time dt ← infinite ;
next state when idle ns ← package sleep;
while true do
    if mode transition request rq is received then
        if rq.requested_power_mode == pmod then
            continue;
        if rq.requested_power_mode == active then
            if timer set then
                unset_timer();
            dt ← infinite;
            transit to package sleep;
            ns ← package sleep;
        if rq.requeted_power_mode == sleep then
            dt ← (rq.τ);
        if timer expires then
            transit to ns;
        if all cores idle and pmod == sleep then
            set_timer(dt);
            ns ← system sleep;
```

sleep upon delay timer expiration. However, if the scheduler chooses to wake up the server before timer expiration (e.g., due to sudden load increase), the timer is reset and the server goes back to active mode.

For large server farms, we can apply one of two possible solutions: 1. Adopt a distributed power management approach where energy is optimized within individual domains of servers with their own power managers. 2. Adopt a hierarchical solution with multiple levels of global power managers. We note that a distributed power management approach may be more scalable with lower implementation complexity compared to a hierarchical approach that may involve longer latencies for decision making and higher bookkeeping overheads for the servers.

#### *3.3.2.2 Adaptive Server Provisioning*

Job arrival pattern may have local spikes (bursty), during which the service latency may suffer, especially when the servers are in low-power modes. To mitigate this problem, we provision a subset of servers in shallow sleep states dynamically by setting their delay timer values to infinity. WASP determines the number of provisioned servers *dynamically* through measuring the current standard deviation in the job arrival rate observed over a period of 2 minutes. Specifically, the server provision module samples the number of arrivals and calculates the utilization for each sample period (one second in our current setting). It then uses the sampling window to determine the standard deviation in the level of system utilization. The module will provision $\alpha \times stdev \times number\_servers$ dynamically in shallow sleep state. $\alpha$ is a tunable parameter. By default we set it to 3.0, since it typically covers a vast majority of the population (e.g., more than 99% of the population in Gaussian distribution).

### 3.3.3 Experimental Setup

We perform two sets of experiments: 1. simulations to explore the Pareto-optimal energy-latency tradeoff as well as corresponding $T_s$, $T_w$ and $\tau$ settings, and 2. prototype implementation on a testbed with web server deployment. We then elaborate on the experimental setup for both approaches.

#### *3.3.3.1 Server Power Profile and Low-power State Configuration*

We profile the power consumption of the Intel Xeon E-5 processor [71] using Intel's *Running Average Power Limit* (RAPL) interface. We build a customized cpuidle governor that allows specified low-power state transitions. The processor is programmed to transition between active state (C0) and low-power state $Cx$ (e.g., C1, C6). We observe that C6 state save significantly more power than other shallower states with minimal additional wakeup overhead. As a result, WASP maps *core sleep* and *package sleep* to core C6 and package C5 respectively. Similar to the configuration in Dual $\tau$ (Section 3.2, S3 state is chosen as the system sleep state since it yields

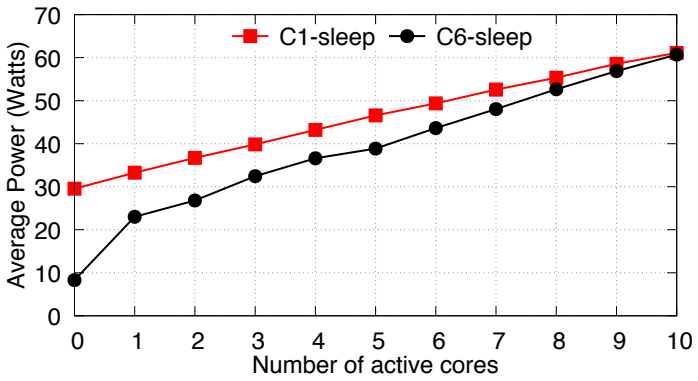



Figure 3.7: Power profile of a 10-core Xeon E5 processor with C0-C1 and C0-C6 transition settings whenever the server is idle.[1]

| **Component** | Core sleep C1* | Core sleep C6 † | Pkg. sleep C6 | System sleep |
|---|---|---|---|---|
| CPU | $33.0+3.1 \times (n_a - 1)$ | $23.0+3.8 \times (n_a - 1)$ | 8.3 | 8.3 |
| RAM [71] | 10.8 | 10.8 | 4.9 | 1.4 |
| Platform [107] | 45.5 | 45.5 | 23.6 | 4.8 |
| Total Power | $89.3+3.1 \times (n_a - 1)$ | $79.3+3.8 \times (n_a - 1)$ | 36.8 | 14.5 |

$*$ processor is active and the rest of the idle cores are in $C1$ state.
† processor is active and the rest of the idle cores are in $C6$ state.

Table 3.8: Power (W) breakdown for a system with $n_a$ active cores

substantial power saving with tolerable restoration time.

Figure 3.7 shows the measured power consumption of the processor for two configurations: *C0-C1*, *C0-C6* and at utilization levels from 0% to 100%. Using linear regression, a power model is built for the processor based on the sleep state selection and the number of active cores at full utilization. Table 3.8 shows the power consumption when a certain state is chosen for sleep mode. Table 3.9 shows the wakeup latencies for various low-power states. Note that the processor sleep state transition latencies are reported by the Linux cpuidle driver [116].

#### *3.3.3.2 Simulation Platform*

WASP uses an event-driven simulator based on Bighouse [109] that models server farm workloads and multi-server activity. We simulate a server farm with 100 ten-

[1]We use a microbenchmark that can calibrate itself and occupy the core based on required utilization settings. To occupy multiple cores, we run multiple copies of this microbenchmark each pinned to a core.

| Low-power State | Wake-up latency |
|---|---|
| Core sleep C1 | 10 $\mu$s |
| Core sleep C6 | 82 $\mu$s |
| Package sleep C6 | 1 ms |
| System sleep | 5 s |

Table 3.9: Processor/System low-power states and wakeup latencies

core servers (by default). In all of our experimental results, we report the steady state statistics by disregarding the warm-up period of the first 10,000 jobs. In the simulation, we use the benchmarks as discussed in Section 3.2.1 as representatives. For each of the representative workloads, we generate synthetic job arrivals with different utilization levels (0.1 for low, 0.3 for average [47], and 0.6 for high). Random job arrivals are modeled by Poisson process [108]. Besides synthetic workloads, we also perform simulation based on Wikipedia traces.

#### *3.3.3.3 Real System Experiments on Testbed*

We deploy a testbed with a cluster of 10 application servers together with one load-generating server and one load-balancing server; all servers support Intelligent Power Management Interface (IPMI) interface [42] for system-level power monitoring. Each application server is configured with the apache web service. The load generator keeps sending web requests to the system according to real system traces (See Section 3.3.5 for further details).

#### *3.3.3.4 Baseline Strategies*

We come up with two baseline strategies that are similar to those described in Section 3.3.1. Note that instead of C0 active and idle states, we leverage deeper core/processor C states that are more power-conserving.

1. *Active-Idle*, denoted as A-I, is a power configuration where the server alternates between active and shallow package sleep state, C1. A server is active when at least one of the cores in the processor within the server has a job to process. The server enters C1 if none of its cores are actively running jobs.

2. *Delay-Doze*, denoted as A-I($\tau$)-S, is a power configuration where the server transitions between three states – active, package sleep C6 (I), and system sleep (S3). When all cores are idle, the server immediately enters $I$, then goes to $S$ after a delay of $\tau$ seconds. If a new job arrives before the delay timer expires in $I$, the server would transition back to the active state. Note that the C state and system sleep state choices are consistent with the ones used by WASP (Section 3.3.2.1) for fair comparison.

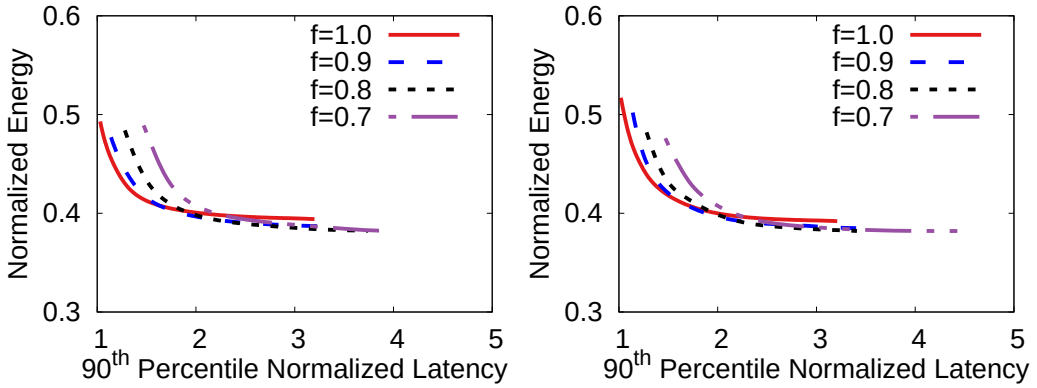
### 3.3.4 Energy-Latency Tradeoff Exploration

We conduct parameter exploration with thousands of simulator runs for every workload at a given utilization level to find optimal ranges of $T_s$, $T_w$ and $\tau$ values under various QoS constraints. We study using four synthetic workloads at three different system utilization levels, and using real system traces from Wikipedia. In each run, we measure energy consumption, average job latency, and $90^{th}$ percentile normalized job latency. The energy-latency frontier curves are then generated by eliminating the less-desired pairs which do not lead to Pareto-optimality. We note that further opportunity may exist if we combine sleep states with DVFS, that reduces dynamic power. Therefore, we conduct exploratory experiments to study the benefits in combining WASP with DVFS to optimize the energy spent during job execution.

#### *3.3.4.1 Frontier Curves on Random Arrivals*

Figure 3.8 and Figure 3.9 present the relationship between energy and job latency under various frequency ($f$) settings for the synthesis workloads. We summarize our findings as the following:

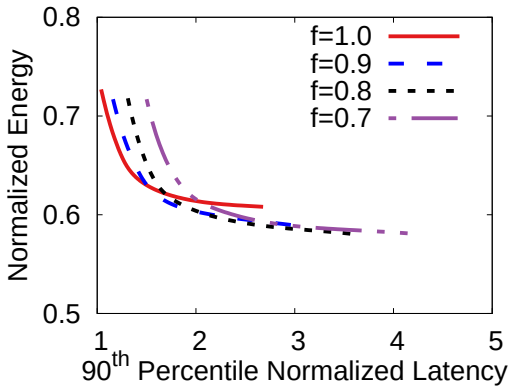
**WASP shows good energy-latency tradeoffs across workloads and utilization levels**. In short-latency workloads (Google and Apache) at utilization level of 10%, without any frequency scaling ($f$=1.0 is depicted as solid red curves), WASP demonstrates 57% reduction in energy compared to a naive Active-Idle (A-I) power management policy (See Section 3.3.3.4). Note that the $90^{th}$ percentile normalized latency is within 2.0. At utilization level of 30%, WASP achieves 39% energy reduction



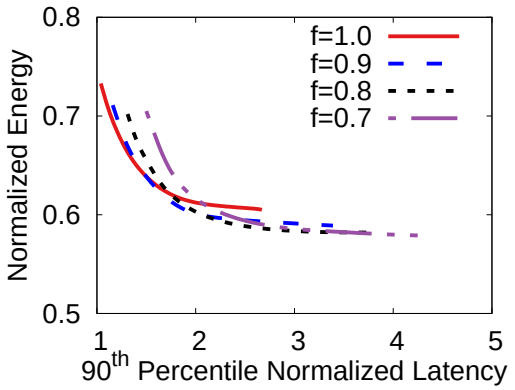

(a) Google, Util=0.1



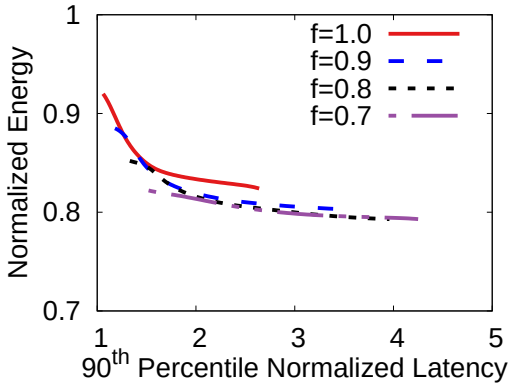

(b) Apache, Util=0.1



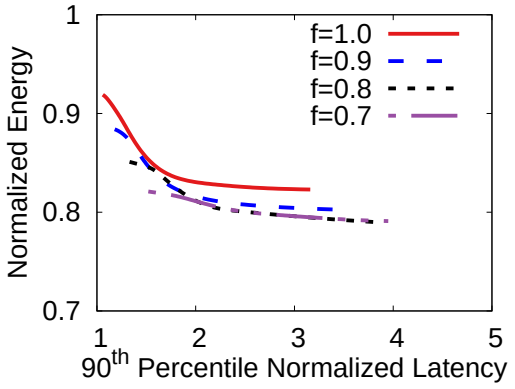

(c) Google, Util=0.3




(d) Apache, Util=0.3

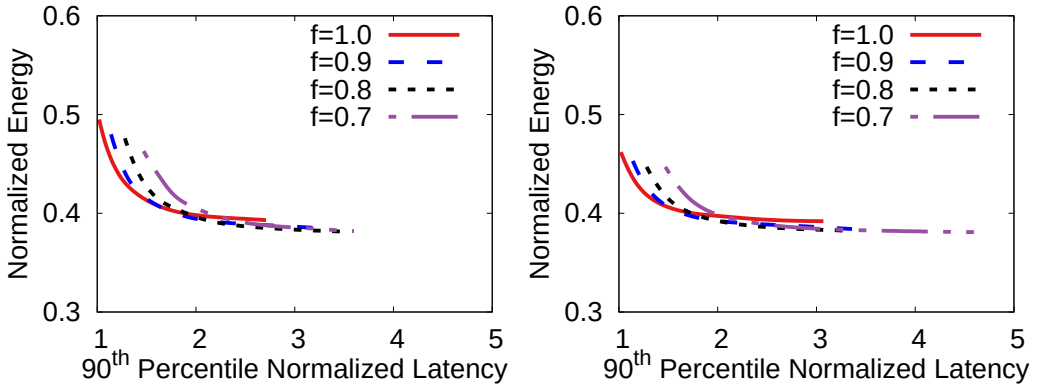



(e) Google, Util=0.6




(f) Apache, Util=0.6

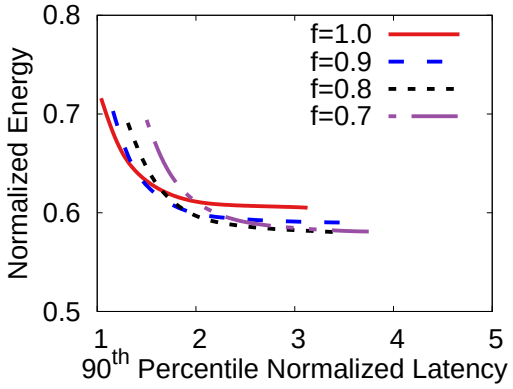
Figure 3.8: Pareto-efficiency frontier curves for energy vs. latency for **Google** and **Apache** benchmarks. Job arrivals are modeled as a Poisson Process, and energy, latency and frequency values are normalized with respect to the configuration using Active-Idle power management policy.



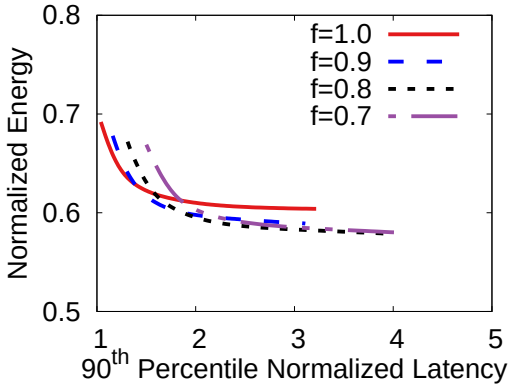

(a) Mail, Util=0.1



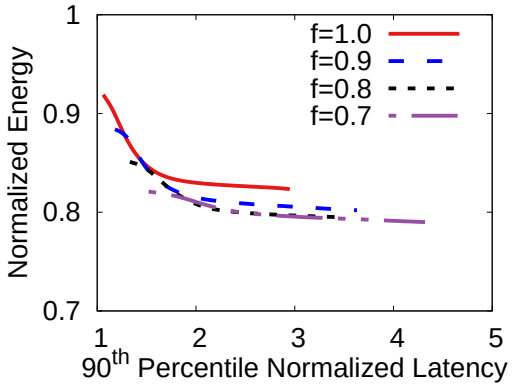

(b) DNS Service, Util=0.1



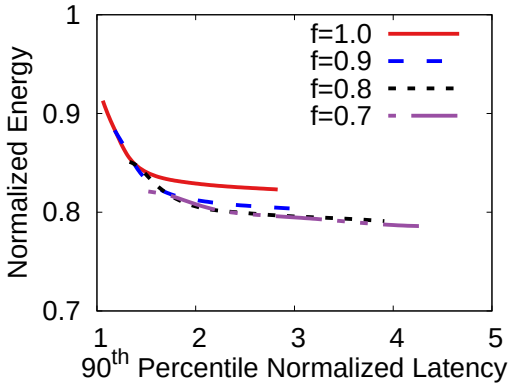

(c) Mail, Util=0.3




(d) DNS Service, Util=0.3




(e) Mail, Util=0.6




(f) DNS Service, Util=0.6

Figure 3.9: Pareto-efficiency frontier curves for energy vs. latency for **Mail** and **DNS** benchmarks. Job arrivals are modeled as a Poisson Process, and energy, latency and frequency values are normalized with respect to the configuration using Active-Idle power management policy.

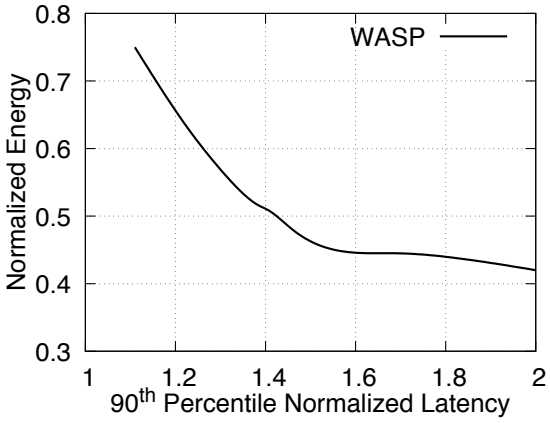



Figure 3.10: Pareto-efficiency frontier curve for Energy-Latency tradeoffs on real-world Wikipedia traces.

for Google workload. We note that WASP shows a similar trend in other workloads as well.

**Using DVFS in conjunction with WASP improves Pareto optimality to a limited extent with increased tail latency**. Our results show that, by lowering $f$ to some extent, higher energy reduction can be achieved without adversely affecting the latency values. For example, for DNS Service at the high utilization of 0.6 with $f = 1.0$, the energy reduction can be up to 17% when the $90^{th}$ percentile normalized latency is 2.0. When $f = 0.7$, energy decreases by another 6% with tail latency degrading to 3.0. However, if $f$ is lowered too much, there is a significant deterioration of performance without a proportional increase in energy savings. Consequently, to achieve higher energy savings in the active mode using DVFS, appropriate selection of $f$ is needed. While better energy saving may be possible by incorporating more intelligent DVFS control algorithms, we note that the room for improvement is limited due to the fact that WASP already takes advantage of system idleness for energy improvement.

#### *3.3.4.2 Frontier Curves on Non-bursty Traces*

We obtain publicly available Wikipedia website traces [131] that include arrival timestamps for each web request along with the URL. Studies by van Baaren et al. [131] have characterized the average job service time as 3.5 ms. For our study,

we obtain the arrival timestamps from the traces, and adopt the job service time distribution from [131]. We simulate a one-hour Wikipedia trace on a 10-machine configuration. Figure 3.10 shows the Pareto-optimal curves for energy and latency. WASP is able to achieve 58% energy saving over Active-Idle with $90^{th}$ percentile normalized latency below 2.0, which is similar to the energy reduction observed in the synthetic Web Service workload in Section 3.3.4.1.

#### *3.3.4.3 WASP Parameter Selections*

A natural question that arises during energy optimization is: what set of parameter values in the WASP algorithm help achieve energy-latency Pareto-optimality? Understanding the characteristics of these parameter values is essential for users to dynamically configure the system under various workloads and latency constraints. For each workload, we collect Pareto-optimal values of $T_s$, $T_w$, and $\tau$ for different utilization levels. Due to space limitations, we are unable to show all of our results. We list our key observations below:

1. *The values of $\tau$ that lead to different normalized tail latencies are fairly independent of utilization levels, but is job-size dependent.* The Pareto-optimal $\tau$ for Google is 0.5s and $\tau$ increases with job size, e.g., 10s for DNS Service.

2. *The values of $T_w$ are also independent of utilization levels.* Intuitively, $T_w$ controls how fast a server in sleep mode would transition to active mode. As $T_w$ increases, servers are woken up less often, which saves energy at the cost of increased tail latency. This gives hints on why $T_w$ also scales linearly with latency values.

3. *$T_s$ values are independent of job execution latencies and utilization levels.* Moreover, when upper bound of the $90^{th}$ percentile normalized latency is set to stringent values such as 2.0, the values of $T_w$ and $T_s$ are also quite close across benchmarks.

The characterization of the WASP parameters helps to optimally select $T_s$, $T_w$ and $\tau$ parameters according to the workload, utilization levels and tail latency requirements in an *automated* manner. Our experiments show that our regression model can accurately predict the three parameters fairly quickly. For verification, we use the regression-derived parameters (for a specific QoS) and compare the tail latency and

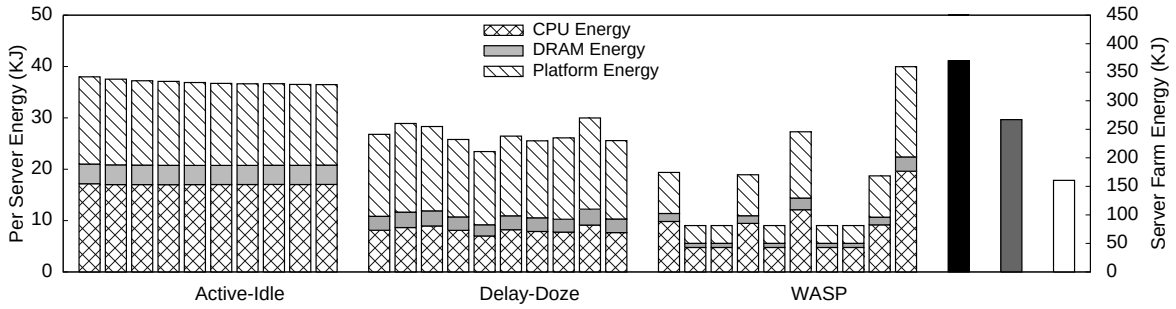



Figure 3.11: Energy measured on a server farm with 10 servers with different energy management policies. The first three groups of bars represent energy breakdown in each server when Active-Idle, Delay-Doze and WASP are applied, respectively. The rightmost three bars illustrate the total server farm energy consumption for Active-Idle (black bar), Delay-Doze (gray bar) and WASP respectively (white bar).

energy consumption with the ones we got from the frontier curves, and our results showed less than 5% absolute error.

### 3.3.5 Evaluation on Real System

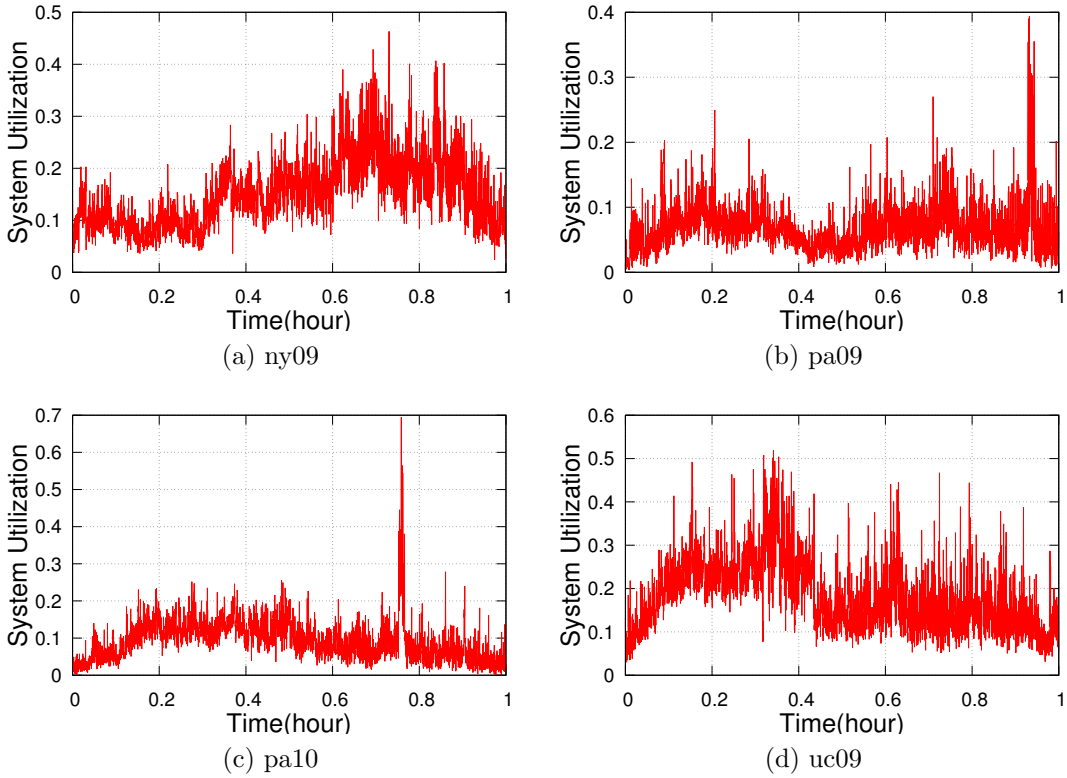
We evaluate WASP on a testbed with 10 Dell Poweredge servers equipped with Inten Xeon-based processors with all of the servers deployed on a dedicated rack. We installed a modified version of the Apache HTTP server for our Local Power Controller. We extended the Local Power Controller to also include a Delay-Doze timer. The Global Server Farm Power Manager is added to an additional apache server with the *mod_proxy_balancer* module used for load balancing. Specifically, the load balancer performs operating mode transitions in servers. This is done by sending special HTTP requests (*/hostname/trans-to-active-mode/*, */hostname/trans-to-lp-mode*) to the application server. It also monitors the power state for each server, and manages the server wakeups (from system sleep) using IPMI interface supported by Dell systems. The special requests are handled by the local power controller that would determine the server low-power transitions accordingly. We set up a custom cpuidle governor which allows direct processor C-state transitions from userspace (e.g., C0-C6). For power measurement, we leverage two techniques: the RAPL interface for fine-grained component power, and the IPMI's system power management interface for coarse-grained server power. We evaluate the effectiveness of WASP by providing

two sets of workloads to the system: the non-bursty Wikipedia workload, that does not require server provisioning, and four bursty NLANR workloads [4], that require server provisioning to handle bursty workloads (See Section 3.3.2).

#### *3.3.5.1 Non-bursty Traces*

We first performed real system energy measurements by deploying Wikipedia software stack, namely Wikipedia application (Mediawiki), database system (Mysql) on servers. We compare WASP against Active-Idle and Delay-Doze approaches described in Section 3.3.1. To capture detailed energy breakdown, we leverage RAPL interface for fine-grained power measurement. The RAPL utility records the CPU and RAM power values periodically. We configure WASP with the $T_s$, $T_w$ and $\tau$ parameters that achieve energy-latency Pareto-optimality with tail latency constraint set to 2.0. Similarly, for Delay-Doze, we explore various values of the delay-timer and choose the setting that achieves the best power with the same tail latency constraint. From our experiments, we get actual CPU and RAM energy consumption for each server. To get the overall server energy, we also factored in the platform energy shown in Table 3.2.

Figure 3.11 shows the per-server energy breakdown in terms of CPU, DRAM, and platform energy. With Active-Idle power management, all the 10 servers have similar energy consumption. With Delay-Doze, some of the servers are able to stay in system sleep state for longer periods of time, thus saving energy. With WASP, we can clearly see that most of the servers drastically reduce energy consumption, and only a minimal subset of servers (server#6 and #10) are used for servicing jobs. Note that the energy consumption of server#10 is slightly higher than that of Active-Idle power management since the server is at a higher utilization level while other servers remained inactive. Overall, WASP gains 39% reduction in energy saving compared to Delay-Doze, and 56% energy savings compared to Active-Idle.




(a) ny09 (b) pa09

(c) pa10 (d) uc09

Figure 3.12: System utilization for four bursty traces.

### *3.3.5.2 Bursty Traces*

As the raw NLANR network traces [4] present job arrivals that are too infrequent for the server farm system with 10 application servers (less than 2%), we speed up the trace by scaling the 24-hour trace to one hour. We choose four traces, namely *ny09*, *pa09*, *pa10* and *uc09*. Figure 3.12 shows the utilization levels (scaled) for the four traces over one hour. All traces exhibit bursty traffic patterns. For example, the *ny09* trace has highly fluctuating utilizations ranging from 4% to 45% with a large number of spikes. To run the traces, we set up a software stack similar to the one in Section 3.3.5.1. Each request in the trace is serviced by a PHP script that accesses a pre-defined set of pages randomly, and we note that the average service time is about the same as Wikipedia web requests.

To enable server provisioning, the Server Farm Power Manager additionally samples the server farm utilization levels based on the job arrival rates. Utilization is

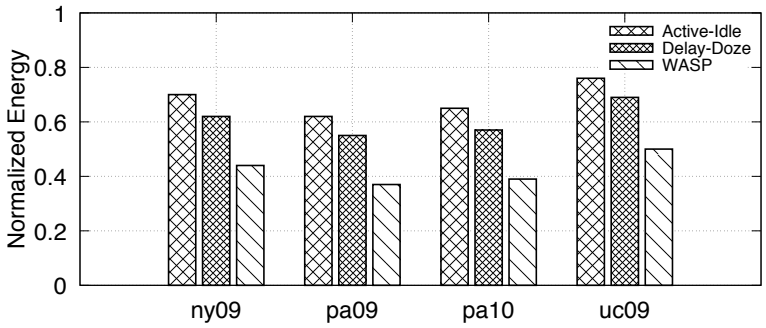



Figure 3.13: Normalized energy consumption relative to peak energy on a 10-server cluster.

calculated as the product of job arrival rate and average job execution time. Standard deviation on the samples for utilization levels is calculated every 120 seconds. The number of provisioned servers is calculated dynamically (See Section 3.3.2.2 for details). Note that, in our comparative studies, the delay-timer values are re-evaluated for each trace such that best possible energy savings are had while meeting the QoS constraints. Figure 3.13 shows the energy consumption for the four bursty workloads using Active-Idle, Delay-Doze and WASP. The energy is normalized to the peak energy which is $PeakPower * Time$. The energy reduction for WASP ranges from 34% to 40% compared to Active-Idle. Even with the best delay timer settings, Delay-Doze only achieves 9% to 12% energy reduction in bursty workloads. We observe that due to the job arrival rate spikes (especially for *uc09*), in order for Delay-Doze to meet the tail latency constraint of 2.0×, the delay timer has to be set to larger values, and in turn the servers have limited chance to enter system sleep state.

## 3.4 TS-Bat: Multi-core Processor Power-aware Scheduling with Temporal-spatial Batching

As shown in Section 3.2 and Section 3.3, server low-power states can save a considerable amount of energy. However, for multi-core processors, we observe that in order to achieve sufficient residency in energy-saving low-power states, it becomes necessary to *intentionally* generate processor package-level idleness. In this section, we demonstrate TS-Bat, a QoS-aware scheduling framework that performs temporal and

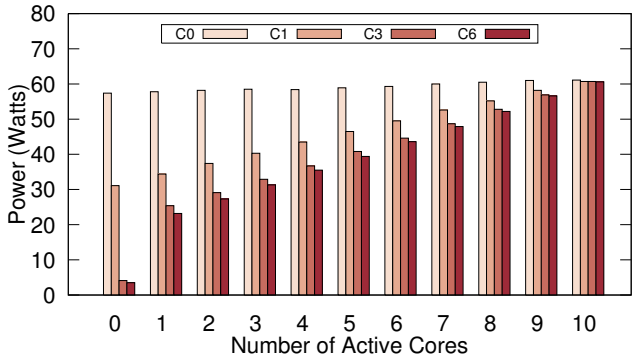



Figure 3.14: Power range of a 10-core Xeon processor with different numbers of active cores and various C states configurations.[2]

spatial batching to improve processor energy efficiency by optimizing package-level low-power state residency.

### 3.4.1 Multi-core Processor Power Characteristics

In order to effectively leverage processor low-power states, it is important to understand the power characteristics of multi-core processors under various C state configurations. Figure 3.14 shows the power profile for a 10-core Xeon E5-2680 processor when varying the number of active cores (idle cores are put to certain C state). The processor power is read using Intel's Running Average Power Limit (RAPL) interface [72]. We observe that the power proportionality increases as deeper level C states are chosen for idle cores.

For deep sleep states, such as C3 and C6, we observe a significant power drop for the processor from having one core active to all cores idle (the first two groups of bars). This is due to the fact that the processor package has to be in C0 (active) state whenever any of the cores is active. When all the cores are in low-power state $C_x$, the entire package can enter $C_x$ state, which further saves power through power-gating the shared resources such as the last level cache.

Figure 3.15 shows the processor power efficiency with different numbers of active

[2]We use a microbenchmark that can occupy a fixed number of cores with *taskset*. The rest of the idle cores are allowed to enter a controlled C state. Each power measurement is made using *RAPL* for a 5-minute run. Intel's Turbo Boost is disabled and the *performance* frequency governor is used to eliminate noise effect due to processor frequency fluctuations.

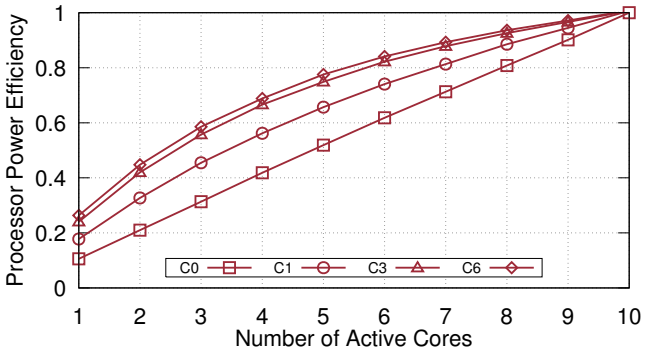



Figure 3.15: Power efficiency of a 10-core Xeon processor with different level of C states configurations.

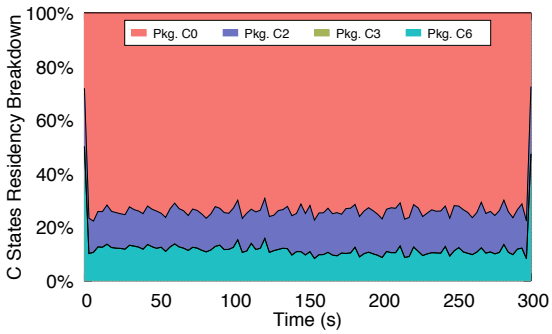
cores. We define power efficiency for a multi-core processor as: $\frac{(P_{all-cores-active}/N)}{(P_{n-cores-active}/n)}$, where $N$ and $n$ represent the total number of cores and the actual number of active cores respectively. We can see that the power efficiency increases as the processor is more utilized. This indicates that, to save energy, two strategies need to be considered together: (i) increasing the utilization of cores in the multi-core processor so that it is operating in the most energy-efficient mode; (ii) keeping all the cores idle so that a considerable amount of power could be saved from the package-level, deep C state.

### 3.4.2 Motivation for Job Batching

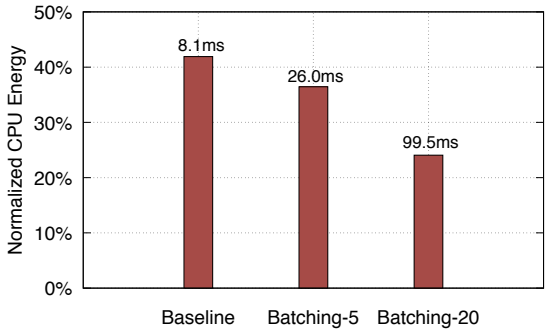
As discussed Section 3.4.1, multi-core processors consume a considerable amount of base power to keep the processor package active. Therefore, keeping the processor in package sleep state for a longer period of time is a straightforward strategy for saving processor energy, especially during periods when servers are underutilized. In order to reside in package-level low-power mode, all of the cores within the same processor need to be idle and enter the core C state first. However, due to the increasing core count in modern multi-core processors, the busy and idle activities for individual cores could hardly synchronize without additional control at the processor level. As a result, the processor package can rarely go to sleep state naturally.

To study the package C state residencies, we set up Apache web server on a Xeon-

[3]The C state residency is reported using *turbostat*. Due to limitation of the RAPL implementation on our platform, *Package C0* represents the combined residence for package C0 and C1.




(a) C state residency breakdown-baseline[3]




(b) Normalized energy for baseline and a simple batching mechanism

Figure 3.16: (a) Package C state residency breakdown for the processor running a web server with an average 10% utilization; (b) energy consumption for baseline (no batching), Batching-5, and Batching-20 that accumulate 5 and 20 jobs, respectively (normalized to energy consumption with C state disabled).

based server. The web application has an average request service time of $5ms$. We use $95^{th}$ *percentile latency* for QoS constraint, and assume that the QoS constraint for the web application is $50ms$. Also, we consider a baseline algorithm that performs load-balancing evenly across different cores. Figure 3.16a illustrates the fractions of time spent in various package C states over time for the baseline algorithm with under 10% system utilization. The plot shows that even at the low utilization level when the cores are supposed to be mostly idle, the processor spends a very small proportion of time in the C6 state.

We develop a simple batching algorithm that batches a fixed number of web requests in the front-end before dispatching them to the server. Figure 3.16b shows the normalized energy consumption for the baseline and two batching configurations that batches 5 and 20 jobs, respectively. The $95^{th}$ percentile latency is shown on top of each bar. Our results show that even naive batching can save considerable energy. *Batching-5* achieves around 16% energy reduction compared to the baseline and *Batching-20* yields almost 43% energy savings. Clearly, we observe that batching should be judiciously used: *conservative batching policies* leave considerable *latency slack* from the target (seen in Batching-5); *aggressive batching policies*, though capable of saving substantial amount of energy, may significantly violate QoS constraints due

to job queuing (seen in Batching-20).

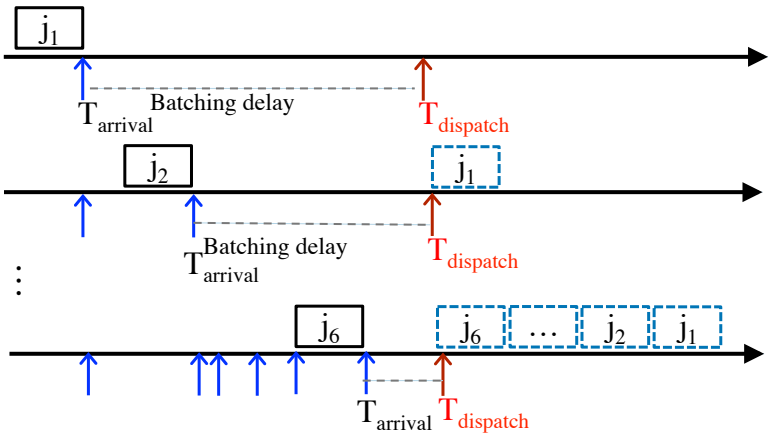
### 3.4.3 TS-Bat Design

In this section, we present the system design of TS-Bat. TS-Bat first performs temporal batching in the front end. Specifically, instead of dispatching job requests immediately to the individual servers, the temporal batching engine accumulates a certain amount of jobs and distributes the entire batch to a back end server. Essentially, this creates opportunities for the multi-core processors to use all of the cores at the same time (when the job batch arrives), thus improving the energy efficiency. As discussed in 3.4.2, batching job requests aggressively can adversely impact the job response time. To maintain the Quality-of-Service for the jobs, TS-Bat integrates a two-stage queuing model that determines the maximum number of jobs to batch without violating the target latency constraints.

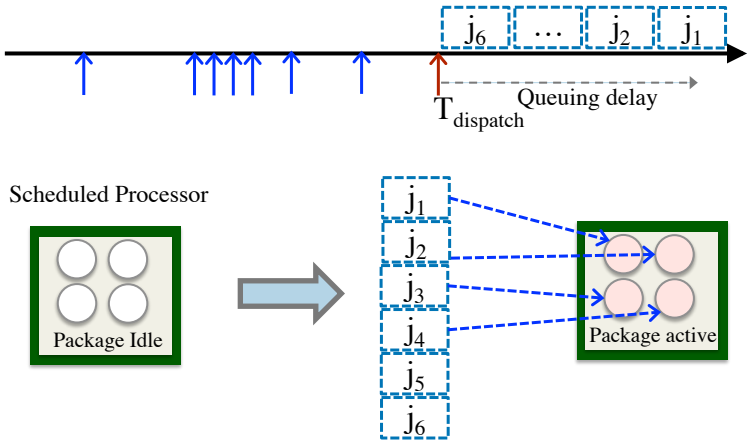
To further save energy, TS-Bat incorporates a spatial batching engine that maintains estimated *to-be* idle time for each of the servers. TS-Bat then schedules the job batch (from the temporal batching engine) to the first available server in a specific search order. Through spatial batching, jobs are concentrated on a small subset of servers such that the processors from the rest of the servers could stay in deep package sleep state without being unnecessarily woken up. The combined temporal and spatial batching make sure that significant processor energy could be saved while still maintaining the job QoS constraints.

#### *3.4.3.1 Design of Temporal Batching*

Data center service providers for latency-critical applications generally specify a target tail latency (e.g., $95^{th}$ percentile response time) for QoS guarantee. Typically, there is a latency slack between an application's actual job service time and the target tail latency. We can take advantage of this latency slack by accumulating jobs (temporal batching) before they are sent to the back-end servers. We note that, as long as the tail latency constraints are satisfied, it is acceptable to delay executing the jobs. In our work, we assume a multi-server infrastructure where each server has




(a) Temporal Batching in the Front-End




(b) Batched Jobs Processing on Application Server

Figure 3.17: An illustration of temporal job batching procedure assuming that the server is equipped with a 4-core processor. (a) shows how the jobs are batched together before they are dispatched; (b) illustrates how the batched jobs are serviced at the local server. Note that the first 4 jobs are processed simultaneously while the other jobs are queued.

parallelism due to the existence of multiple cores. We assume a FIFO job dispatching model where job requests arrive and get assigned in a first-in first-out order. Note that such queuing has been shown to be optimal for tail latency [87].

The challenging task of temporal batching is to determine the right number of jobs (denoted as $K$) to batch so that the QoS will not be violated. In order to derive this parameter, we need to understand various delays in the critical path of job batching and processing. Figure 3.17 illustrates an example scenario. Specifically, Figure 3.17a shows the job batching at the front-end. In this example, 6 jobs are batched before

they are scheduled to a server. Each job experiences a *batching delay* which starts from the time it arrives $T_{arrival}$ to the time when the entire batch gets scheduled on a server. Figure 3.17b illustrates the procedure for job processing at the local server. Note that server farm latency-critical workloads (e.g., web server) utilize multi-threading mechanism in multi-core processors to improve overall throughput. Typically a local load balancer designates queued requests to multiple threads running in parallel. Since we have a 4-core processor, the first 4 jobs will be serviced simultaneously while the remaining two jobs will experience a *queuing delay*. To formalize the problem, let $K$ be the maximum number of jobs that can be batched temporally, and $j_i$ ($1 \leq i \leq K$) is the $i^{th}$ arrived job in the system. The total delay $D_i$ for job $j_i$ can be represented as:

$$D_i = B_i + U_i \tag{3.1}$$

where $B_i$ and $U_i$ are the expected batching delay and queuing delay for $j_i$, respectively. Assuming that $S$ is the job service time distribution and $\lambda$ is the job arrival rate for the system, we have the following:

$$B_i = (K - i)/\lambda + \sigma \tag{3.2}$$

$$U_i = S^{95} * (i - 1)/C + W \tag{3.3}$$

where $\sigma$ is a constant that represents the overhead of batching, $C$ is the number of cores per server, $S^{95}$ is the $95^{th}$ percentile service time based on distribution $S$, and $W$ is the wakeup latency for the processor in package sleep state. Based on these equations, $K$ can be derived as the maximum value that satisfies the following inequality for all $i$:

$$D_i + S^{95} \leq Q \tag{3.4}$$

where $Q$ is the target tail latency.

The distribution $S$ can be generated by monitoring the service times at runtime. We assume that the service time distribution does not change much over time, which

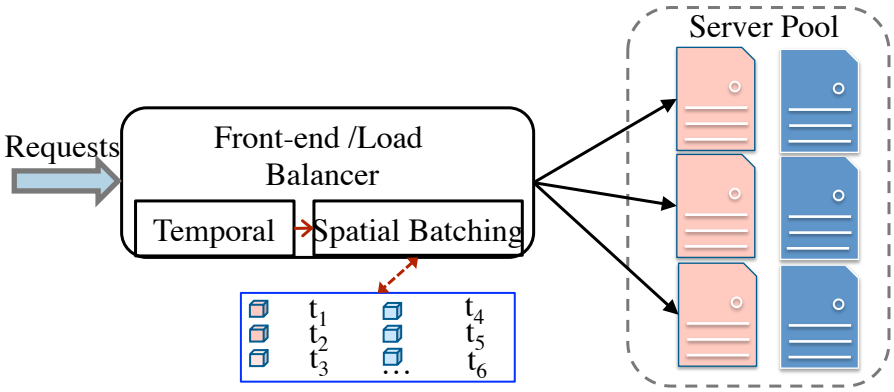



Figure 3.18: Overview of overall TS-Bat scheme. $t_i$ is the estimated processor idle time for server $i$. $t_1$, $t_2$ and $t_3 \geq t_{cur}$, which means these servers are currently busy processing the batched jobs; $t_4$, $t_5$ and $t_6 \leq t_{cur}$ indicating these three servers are idle.

is reasonable as data center operators typically do not mix latency-critical workloads with others [101]. As a result, $S$ only needs to be profiled once (e.g., in the warm-up period of every workload). The value $K$ can then be derived by repetitively incrementing $K$ until inequality (3.4) is no longer satisfied. Since $K$ is dependent on job arrival rate $\lambda$, the temporal batching engine periodically samples the job arrivals and updates the value of $K$. We note that once the distribution $S$ is determined, the value $K$ under different arrival rates can be pre-computed. Particularly, it is possible to compute various values of K for different QoS targets as a *lookup table* which can be looked up by the TS-Bat runtime to avoid repetitive calculation. We also note that, when $\sigma$ is sufficiently less than the job service time, the value of K can be independent of average job service time. Such observation can further be leveraged to reduce runtime computation overhead. Additionally, to avoid unnecessary delays due to having to batch $K$ jobs (e.g., a sudden drop in arrival rate), an optional timer can be set to trigger dispatching of the currently accumulated jobs such that the earliest job will meet its deadline.

#### *3.4.3.2 Design of Spatial Batching*

When a batch of jobs is accumulated by the temporal batching engine, the front-end needs to find a server to process it. One possible way is to evenly distribute the loads to all of the application servers. However, this approach is not energy-

efficient because randomly dispatching job batches can create frequent active periods for all servers. Since the operating system makes sleep-state decisions based on server activities, evenly distributing workload may leverage only shallow sleep states in the absence of sufficiently long idle periods. To avoid this, we propose a spatial batching engine that determines the server to schedule the job batch. To do this, the spatial batching engine maintains a list that provides estimated times when the servers would become idle: $t_{current} \geq t_i$ where $t_i$ is the estimated time when server $i$ resumes idle and $t_{current}$ is the current dispatching time. It then updates the server's estimated idle time as $t_{current} + T_b$, where $T_b$ is the estimated job batching time, which can be estimated as $\lceil \frac{K}{C} \rceil * S^{95}$. Figure 3.18 shows the overview of our TS-Bat approach.

### 3.4.4 Implementation

We implement a proof-of-concept prototype system including a load generator using httperf [112], a TS-Bat module, and apache HTTP servers in the back-end. httperf is modified so that it is able to generate loads to multiple apache servers. In the back-end, the apache server is configured in such a way that it always maintains exactly the same number of *httpd* processes as the number of cores. This makes sure that incoming batched jobs are processed based on the queuing model described in Section 3.4.3.

TS-Bat is implemented as a separate module integrated into httperf. Once initialized, the temporal batching engine samples the service times and job arrivals to determine $S^{95}$ and $\lambda$. After the two parameters are determined, it derives $K$ according to the methodology discussed in Section 3.4.3.1. The temporal batching engine then starts to perform job batching. It sets up a timer upon receiving the first job in each batch. The batching is complete either when $K$ jobs are accumulated or when the timer expires, whichever is first. The batching buffer is set to 200 empirically, which is sufficient to hold requests for our workload with the smallest service time. The job arrival rate $\lambda$ is sampled periodically every $t$ seconds, where $t$ is a tunable parameter that controls *TS-Bat*'s reactivity to load burstiness. By default, $t$ is set to 5 seconds. The spatial batching engine chooses back-end servers based on its estimated

idle period, and this information is stored in a linked-list. Note that, to eliminate the potential of resource wear-out, the spatial batching engine would shuffle the order of the servers in the list every $T seconds$ so that all of them are exercised equally in the long run. We set $t$ to 1 second and $T$ to 60 seconds in our experiments.

### 3.4.5 Experimental Setup

**Server platform**. We deployed a testbed with a cluster of 17 servers, including 2 standalone Xeon E5603-based servers and 15 Xeon E5650-based servers from the Dell Poweredge M1000e Blade system. The two Xeon E5603 servers are used as load generators and the blade servers are configured to run apache web service. All apache servers are interconnected with a Netgear 24-port Gigabit switch (star topology). Since our blade servers do not support RAPL interface, we utilize the IPMI interface for system-level power reading. Each Xeon E5650-based server is configured with the Apache HTTP server. The server power is queried and saved every 1 second. We conservatively set the wakeup latency from package sleep state to 1 ms (the actual transition time is usually shorter than 1 ms)

**Benchmarks selection and load generation**. To run various workloads, we developed CGI scripts for the Apache servers. We select a subset of the PARSEC [22] benchmarks to be executed by the CGI scripts. These PARSEC benchmarks represent emerging class of workloads from recognition, mining and synthesis domains of applications that can frequently benefit from running on the cloud. We generate workloads from five selected applications (with their average execution times shown in brackets next to them): Bodytrack (108ms), Raytrace (79ms), Vips (42ms), Fluidanimate (33ms) and Ferret (21ms). Each workload is configured to run for 30 minutes. httperf is set to generate job arrivals based on exponential distribution. We configure httperf to generate three different levels of utilizations: 10%, 20%, and 30%.

**TS-Bat parameter configuration**. The batching parameter K is determined based on the algorithm shown in Section 3.4.3.1. In our experiment, we observe that the $K$ value derived from the analytical model may violate the target QoS on a small number of occasions. One potential reason is that there exists resource contention

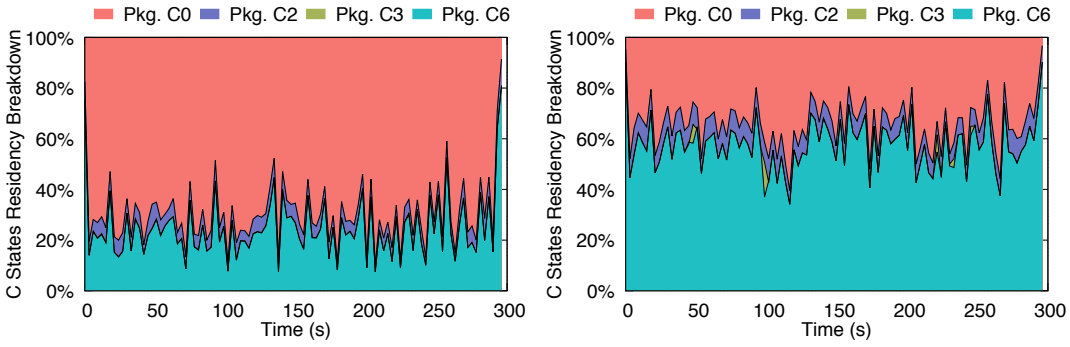
between concurrent jobs. To sustain the QoS target, we set the actual value to $K - \epsilon$. We observe that $\epsilon = 2$ works practically well for all of our cases. Additionally, since the target QoS for different applications may differ, we define QoS as the $95^{th}$ percentile latency normalized to the job's average service time. Note that the target tail latency (QoS) should be provided to the temporal batching engine. We set two different QoS targets for each workload: 5x $95^{th}$ percentile latency (QoS-tight) and 10x $95^{th}$ percentile latency (relaxed QoS).

### 3.4.6 Evaluation

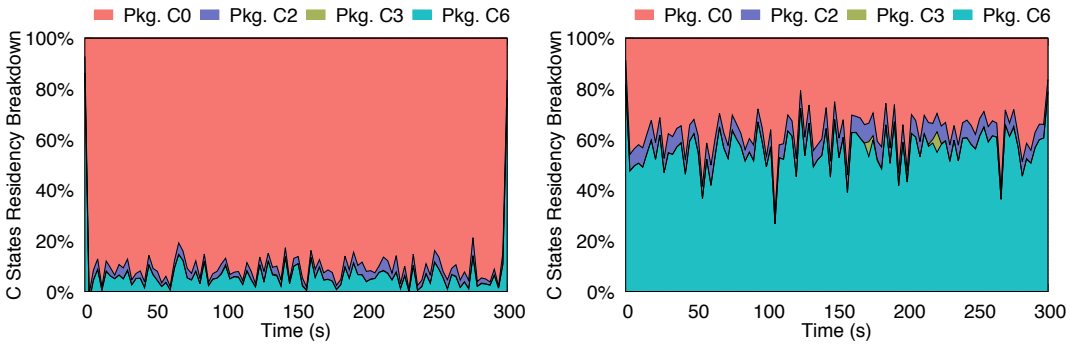
We evaluate TS-Bat in two steps. Specifically, we first demonstrate the energy savings and job performance using just temporal batching on the Intel Xeon E5-2680 server. Then we enable both temporal and spatial batching engines on the blade system and illustrate the potential energy savings. We evaluate TS-Bat in two steps. Specifically, we first show the energy savings and job performance using just temporal batching. Then we enable both temporal and spatial batching engines, and illustrate the overall energy savings that can be obtained from their combined deployment.

#### *3.4.6.1 Results of Temporal Batching*

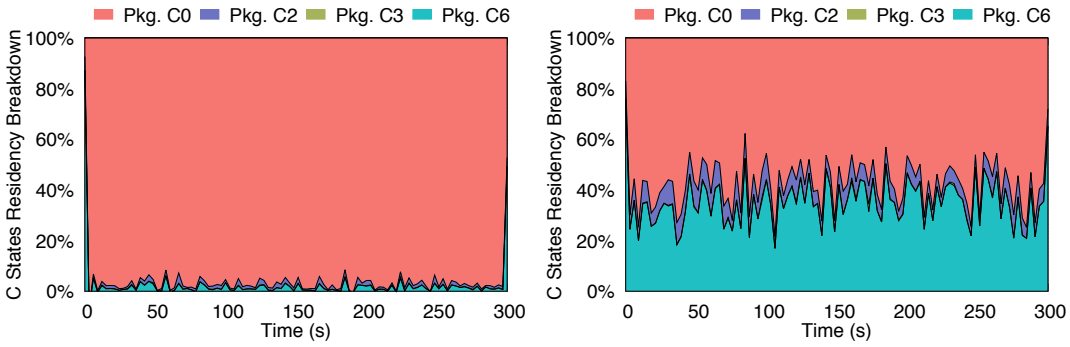
To evaluate the effect of temporal batching, we use a single Apache HTTP server. For this experiment, we use a single benchmark, Bodytrack. We observe that other benchmarks also exhibit similar result trends based on our experiments. For Bodytrack, the two target latencies are 540ms (QoS-5) and 1080ms (QoS-10). Figure 3.19 shows the package C state residency for Bodytrack with and without Temporal Batching. We can see that without batching, the processor spent less than 20% in the package C6 sleep state under 10% utilization (Figure 3.19a), which is significantly lower than the ideal residency of 90% under ideal energy proportionality. Server residency in the power-saving states almost diminishes as the load increases to 20% and 30% (Figure 3.19c and Figure 3.19e). This clearly indicates the inefficiency of low power management in under default OS settings. On the other hand, with temporal batching with TS-Bat, the *Package C6* residency is significantly improved compared to the




(a) Bodytrack without T.B. (Util. 10%) (b) Tmp. Bat. with QoS-tight (Util. 10%)



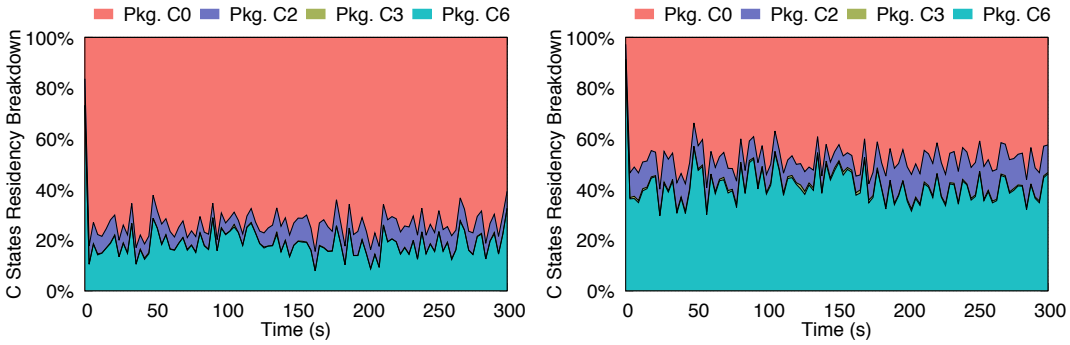

(c) Bodytrack without T.B. (Util. 20%) (d) Tmp. Bat. with QoS-tight (Util. 20%)




(e) Bodytrack without T.B. (Util. 30%) (f) Tmp. Bat. with QoS-tight (Util. 30%)

Figure 3.19: Package C state residency breakdown for **Bodytrack** benchmark. Figure (a), (b) and (c) correspond to the residency breakdown with baseline configuration (no batching) under 10%, 20% and 30% system utilization respectively. Figure (d) (e) and (f) are for the same plots under Temporal Batching with tight QoS (5x).

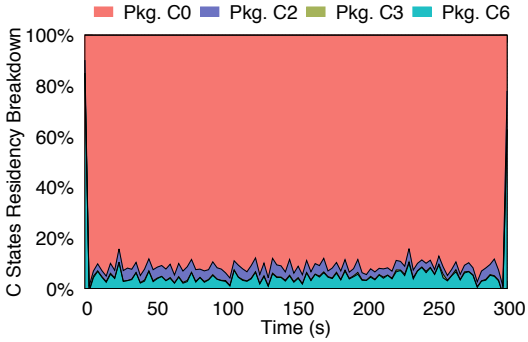
baseline without batching. For instance, the processor spent 41% more time in package C6 state under 10% utilization (Figure 3.19b), and spent 29% more time at 30% system utilization (Figure 3.19f). Finally, the low-power state residency decreases much slower as the utilization level increases, compared to the baseline. Figure 3.20 illustrates the C state residency for the Vips benchmark. Similarly, the percentage



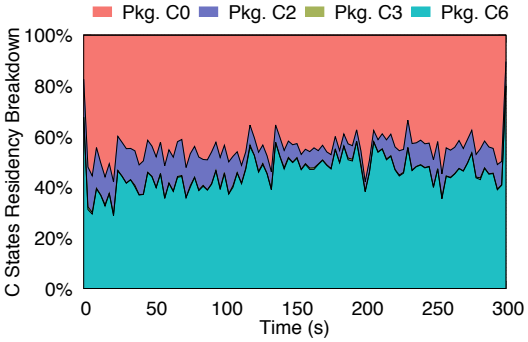

(a) No Tmp. Bat. (Util. 10%)



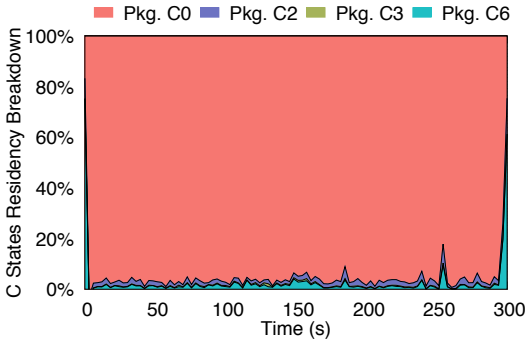

(b) Tmp. Bat. with QoS-tight (Util. 10%)



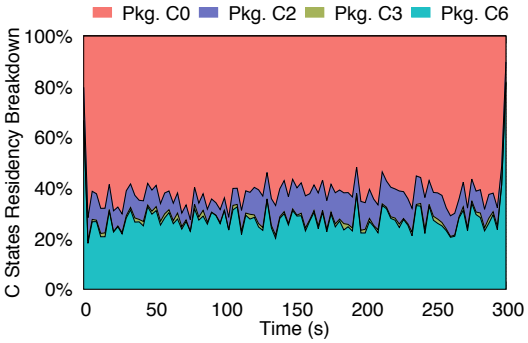

(c) No Tmp. Bat.(Util. 20%)




(d) Tmp. Bat. with QoS-tight (Util. 20%)



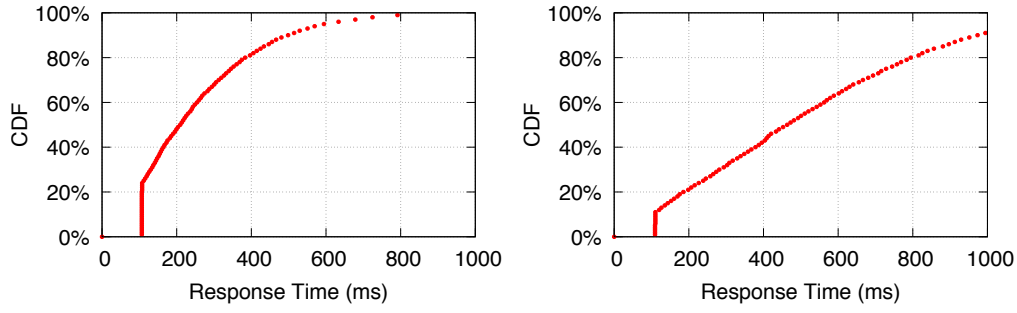

(e) No Tmp. Bat.(Util. 30%)




(f) Tmp. Bat. with QoS-tight (Util. 30%)

Figure 3.20: Package C state residency breakdown for **Vips** benchmark. Figure (a), (b) and (c) correspond to the residency breakdown with baseline configuration (no batching) under 10%, 20% and 30% system utilization respectively. Figure (d) (e) and (f) are for the same plots under Temporal Batching with tight QoS (10x).

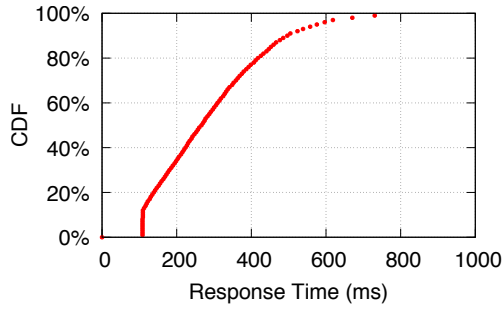
of package C6 state residency is greatly increased under different system utilization levels. Moreover, we can see that, compared to Bodytrack, the C6 state residency is slightly less. We note that the job execution time of Vips is much shorter than that of Bodytrack. Therefore, under the same utilization, the inter-arrival times for *job batches* are relatively longer for Bodytrack, which favors entering of deep sleep



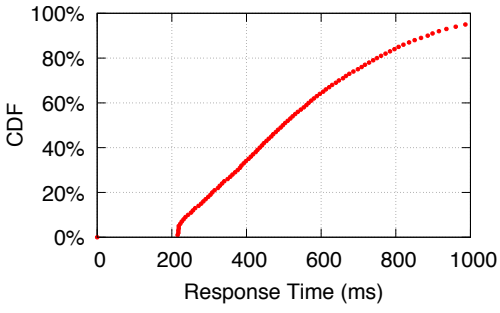

(a) Utilization 10% (QoS-tight)



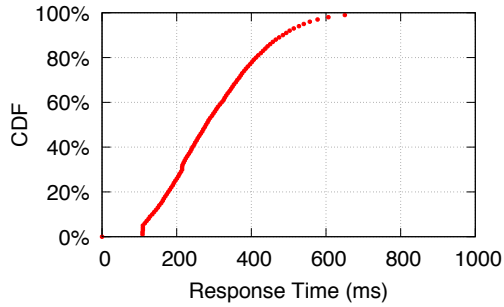

(b) Utilization 10% (QoS-relaxed)



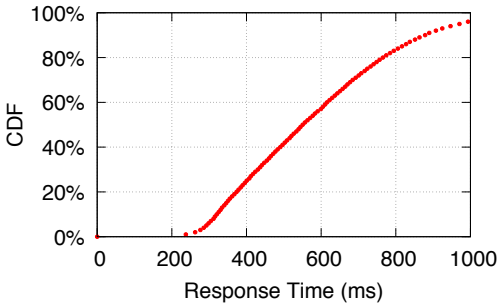

(c) Utilization 20% (QoS-tight)




(d) Utilization 20% (QoS-relaxed)




(e) Utilization 30% (QoS-tight)

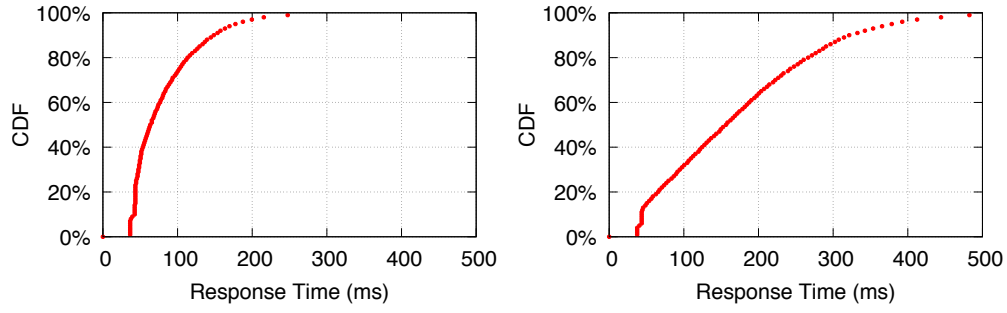



(f) Utilization 30% (QoS-relaxed)

Figure 3.21: Latency CDF for Bodytrack under 10%, 20% and 30% utilization using TS-Bat's temporal batching.

state such as C6. Regardless, we can see considerable improvement in low-power state residency that will eventually reflect as energy savings.

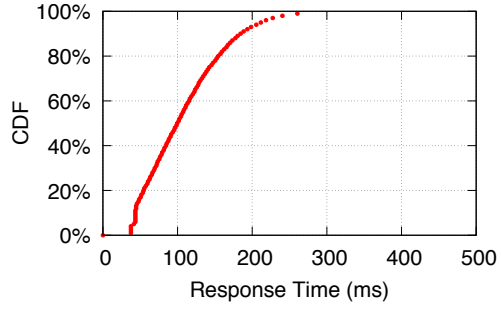
Figure 3.21 and Figure 3.22 demonstrate the response time CDF for Bodytrack and Vips. We find that the temporal batching engine is able to meet the target constraint. For example, the actual tail latencies for Bodytrack (with average service time of 108ms) are 557ms and 986ms under 20% system utilization for QoS-tight and QoS-relaxed receptively. For Vips (with average service time of 42ms), the achieved tail latencies are 211ms and 358ms under 20% utilization using 5x and 10x QoS. Notably,



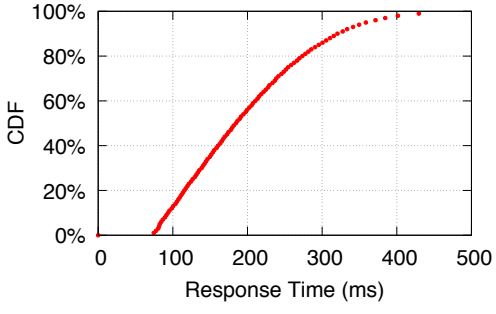

(a) Utilization 10% (QoS-tight)



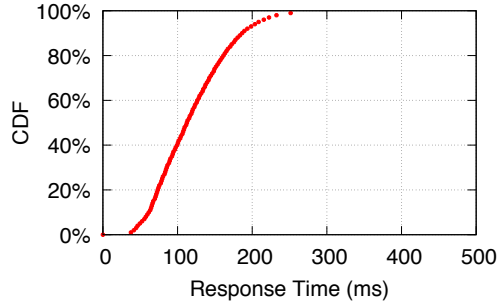

(b) Utilization 10% (QoS-relaxed)



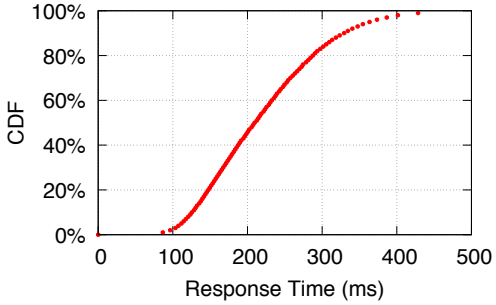

(c) Utilization 20% (QoS-tight)




(d) Utilization 20% (QoS-relaxed)




(e) Utilization 30% (QoS-tight)




(f) Utilization 30% (QoS-relaxed)

Figure 3.22: Latency CDF for Vips under 10%, 20% and 30% utilization using TS-Bat's temporal batching.

TS-Bat can effectively shift the response time for various workloads regardless of the actual loads. We note that TS-Bat's batching algorithm can successfully bound the target latency through batching for all the benchmarks.

Table 3.10 summarizes the energy savings of temporal batching for all five of PARSEC benchmarks in our study. Consistently, the energy saving increases as the QoS constraint is relaxed (e.g., from QoS-5x to QoS-10x) as we have observed before. As the utilization level increases, the energy saving reduces under all QoS settings in general. This is due to the fact to the processor idle intervals tend to be shortened

| | Utilization 10% | | Utilization 20% | | Utilization 30% | |
|---|---|---|---|---|---|---|
| | QoS-5x | QoS-10x | QoS-5x | QoS-10x | QoS-5x | QoS-10x |
| bodytrack (108ms) | 31.9% | 48.2% | 33.4% | 34.3% | 20.3% | 24.7% |
| fluidanimate (33ms) | 13.5% | 41.1% | 18.2% | 22.9% | 8.7% | 11.8% |
| vips (42ms) | 21.0% | 44.3% | 27.6% | 30.3% | 16.0% | 20.6% |
| ferret (21ms) | 13.0% | 41.9% | 25.7% | 32.2% | 12.9% | 22.6% |
| raytrace (79ms) | 25.5% | 46.7% | 32.8% | 34.8% | 20.5% | 26.4% |

Table 3.10: Power savings for all benchmarks using TS-Bat's temporal batching. Energy savings are normalized to the baseline (OS default C state management) energy consumption.

with higher utilizations. It is also observed that batching can be more beneficial for applications with relatively larger job sizes (e.g. Bodytrack). Interestingly, when changing the system load from 10% to 20%, for each benchmark, TS-Bat achieves more relative energy savings under QoS-tight than under QoS-relaxed. This is because under QoS-5x, the amount of batching is constrained by the target job latency especially at lower utilization levels. Under QoS-10x, job batching is constrained by the higher job arrivals especially at the higher system utilization. Overall, we can see that temporal batching can save between 8.7% and 48% CPU energy depending on the workloads, server utilization levels and QoS constraints.

#### *3.4.6.2 Combined Temporal and Spatial Batching*

We perform both temporal and spatial batching on all the benchmarks as mentioned in Section 3.4.5 at the utilization level of 30%. The experiment is conducted using 15 Apache servers. The target tail latency is set to 5× for all the benchmarks. Figure 3.23 shows the overall energy savings for the entire cluster. Across all the benchmarks, temporal batching is able to achieve energy improvement between 48%-51%. TS-Bat, that combines temporal and spatial approaches, can provide up to 16% additional energy savings, and achieves up to 68% energy improvement compared to the baseline while meeting the target QoS constraints. We note that, with spatial batching, TS-Bat is able to pack the loads onto a small subset of processors. Differently, in the baseline approach, short latency jobs will incur more frequent arrivals, which prevents the processors from entering deep package sleep, thus significantly

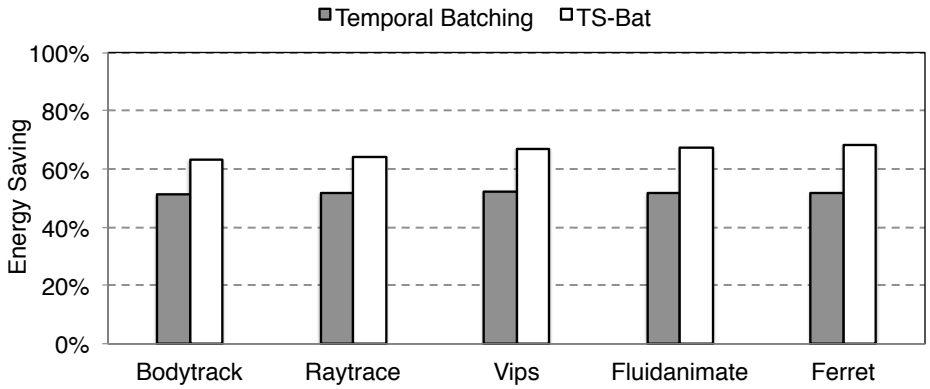



Figure 3.23: Energy savings for various benchmarks with Temporal Batching and TS-Bat at 30% utilization. Baseline has no batching.

increasing the system power consumption.

### 3.4.7 Discussions

**Scalability of TS-Bat**. In the evaluation, we use a centralized controller for the temporal and spatial batching in TS-Bat. This may cause some scalability issues when the data center has thousands of servers. We note that in these large scale data centers, TS-Bat can be easily adapted with minimal modification. Specifically, we can divide the data center into multiple clusters, and each cluster will have its own TS-Bat controller that coordinates the batching operation. This scheme works well with many existing data center application as a lot of data center service are stateless [57]. That is, one user request can be serviced by any of the application servers in the pool. Eventually, each server cluster acts as a logical data center that can be effectively managed by TS-Bat.

**Energy optimization for both servers and networks**. TS-Bat largely considers energy optimization and QoS management for data center servers. We envision that as improvements of server energy efficiency continue, the energy consumption from network devices will become more influential. While resource management in server networks is a well-studied problem [93, 144–152, 159, 160], the implications of network managements on overall energy efficiency are not fully understood. Some works have proposed active power management mechanisms on network devices using techniques such as dynamic link rate adaptation [6, 113]. However, merely reducing active power

alone is not sufficient for network devices as a large portion of switch power is consumed simply by keeping the major components ON (e.g., line cards) [117] (similar to the case of servers). Recent study has demonstrated the promise of using low-power states for both switches and servers to achieve high energy savings [104]. We note that TS-Bat essentially proposes an effective scheduling framework that can be augmented to create idleness in both servers and network devices to achieve comprehensive energy savings server-network systems.

## 3.5 Related Work

Offering server farm applications with performance guarantee has been widely studied in the literature [10, 12, 13, 155–158]. Due to the rapid increase of server system size, energy efficiency becomes a prominent issue [31, 47, 119, 126, 129]. The trends in server energy proportionality have been studied by Ryckbosch et al. [123]. Since many server system workloads have service level agreements, judicious tradeoffs need to be made between energy efficiency and tail latency [81].

Existing works that target improving energy efficiency in large-scale server systems can be broadly summarized into two classes: server-level and cluster-level energy managements. Dynamic voltage and frequency scaling is shown to save processor power (e.g., [64, 126]) and has been widely utilized in for server-level energy optimizations. Several studies have developed techniques that leverage DVFS to dynamically tune processor performance for higher energy efficiency while satisfying application tail latencies for online latency critical workloads [66, 82, 101, 102]. These studies are largely aimed at applications with higher degrees of pipeline parallelism, and where the dynamic range for every single request involves the coordination of many (or all) servers. Although DVFS is effective in this domain, it's capability in reducing system energy consumption is limited due to the fact that only active processor power is saved. Cluster-level energy management performs global control and packs the workload to run on a minimal subset of servers to achieve high energy proportionality for the entire system. Gandhi et al. [55] have proposed a delayed-off mechanism that turns off a server after it is idle for a preset period of time. Autoscale [57] reduced

multi-server system energy by controlling the number of active servers while satisfying the QoS. A cluster management mechanism that maximizes resource utilization while meeting the QoS constraints for each workload was presented in [41]. In our study, we show how system sleep states can be smartly utilized instead of physically turning a server off since this may introduce unacceptable spikes in job latencies, especially for small jobs. Studies by Kanev et al. [81] highlighted the need for comprehensive sleep state selection. Sleepscale [100] modeled a single-core server system and utilized frequency scaling with CPU sleep states jointly to reduce the average power. Our work shows that in order to improve energy-latency tradeoff, effective energy management and adaptive system configurations of sleep states and their transitions are equally important in *multi-core multi-server* systems.

Similar to batching, Meisner et al. proposed architectural support to facilitate sleep state by delaying and preempting requests that create common idle and busy periods across cores of a server [108]. The proposed mechanism requires additional hardware to coordinate the sleep periods across the cores. Differently, our proposed work uses off-the-shelf hardware and its effectiveness is evaluated on physical systems with real power measurement. Finally, we note that server energy optimization as proposed in our work can be integrated with more energy-efficient data center network topologies [6, 104, 173] to achieve holistic energy savings for server-network systems.

We note that other techniques have explored more specialized approaches such as exploiting heterogeneity of processors or domain-specific accelerators to considerably improve energy efficiency for workloads with QoS constraints [91, 142, 143]. For example, Knightshift [142] incorporated servers with two execution modes– one providing high performance while consuming higher power for high utilization periods, the other being less power-hungry but has reduced hardware performance for low-utilization periods to save power. The framework was further extended in [143] to offer higher cluster-wide energy proportionality. To preserve generality and limit the cost of our solution, we model homogeneous servers and cores with the same capability. We note that when we combine our proposed approach with energy improvement solution approaches on heterogeneous servers, we can further boost energy savings.

Workload characteristic of data center application is a critical factor that influences energy and latency optimization. Typically server farm workloads exhibit various levels of burstiness. Casale et al. [24] analyzed different types of spikes in realistic web server traces. In [28], the authors proposed a single metric, *index of dispersion* to characterize burstiness of web application workloads. Bodik et al. [24] built a *MMPP2* model and evaluated the accuracy in fitting *MMPP* from workload traces. We leverage both MMPP and realistic traces to evaluate the performance of our proposed techniques.

Sever works have looked into minimizing the interference between co-located workloads that might result in application performance degradations [105, 166]. Delimitrou et al. [40] utilized classification techniques to find the impact of server heterogeneity and interference between co-located workloads for resource assignment while satisfying the performance requirements. Dirigent [180] developed a performance-management runtime that predicts and controls the contention-related variance of foreground response times.

Finally, fine-grained software-level tuning has also been studied for efficient power management. Energy saving could also be achieved by carefully tuning applications for usage of processor resources [32, 36] or through load-balancing tasks across cores in multicore processor settings to avoid keeping cores unnecessarily active [114]. Hao et al. studied the use of machine learning techniques to reduce energy consumption for NoC [177]. Meanwhile, several recent works have proposed application-specific energy optimization mechanisms such as approximation and content caching [51, 88, 137]. These techniques are complementary to our work in this dissertation.

## 3.6 Summary

### 3.6.1 Dual Delay Timer

The Dual-$\tau$ algorithm makes smart use of the existing processor and platform sleep states to achieve higher energy savings in comparison to existing approaches that simply use a single delay timer strategy to enter and exit sleep states. We evaluate

our energy-saving techniques for different job arrival patterns and various job sizes. Our experimental results with four synthetic workloads and a real system job trace from Wikipedia servers [130] show that Dual-$\tau$ achieves up to 71% savings in energy over naive energy management without the use of low-power sleep states, and up to 31% energy savings over a relatively smarter energy management mechanism with just a single delay timer to enter the sleep state. We also show that the normalized job latency with our Dual Delay Timer strategy is similar to the latency in the case when the servers are always active and ready to execute jobs. The simulation-based scalability study shows that Dual-$\tau$ is able to achieve consistent energy savings for different sizes of server farms.

### 3.6.2 WASP

To further improve energy efficiency and enable adaptive control in low-power state managements, we explored a system framework (WASP) that makes smart use of processor C states and system sleep states combined with adaptive techniques to orchestrate the entry and exit from these low-power states. WASP also considers the variability of job arrivals for bursty workloads where local spikes in the arrival patterns are monitored. This information is used to dynamically guide its approach to provision servers in shallow sleep states such that they can be woken up faster to meet QoS constraints. Through extensive simulation, the Pareto-optimal energy-latency tradeoff in server farms is explored under different system utilization levels and workloads, which is then used by WASP for parameter selection. We evaluated WASP on a web server testbed and conduct real energy measurements. Our adaptive techniques provide improved Pareto-efficiency in energy latency. With the QoS constraint of $90^{th}$ percentile normalized latency to be under $2\times$ the job execution time, WASP exhibits up to 57% energy saving over a naive policy that uses a shallow processor sleep state when there is no job to process, and 39% energy saving on a delay-timer based approach.

### 3.6.3 TS-Bat

With a comprehensive power profiling on a real multi-core processor (Intel Xeon v2), we observe that the power saving with increasing number of cores entering C state is non-linear. Particularly, more than 90% of CPU power could be saved if the entire processor package is put into idle state. Our findings indicate that creating opportunities for entire package sleep can yield even higher energy saving compared to approaches that are ignorant of this effect. Motivated by the observation, we designed TS-Bat, an energy optimization framework that judiciously integrates a temporal batching engine and a spatial batching engine to save the energy of server farms. To create opportunities for entering processor-level low-power states, the temporal batching engine accumulates just the right amount of jobs before dispatching them to an individual server. To effectively bound the response latencies, the temporal batching engine builds a job performance model based on the wakeup latency values from individual processor low-power states and the available amount of parallelism in the platform (i.e., number of cores per processor). The spatial batching engine then dispatches the ready-to-execute job batch to a server that is estimated to be currently idle. This further saves energy by packing the workloads on to just a subset of processors. We implement a proof-of-concept system of TS-Bat in a testbed with a cluster of servers, and evaluate the effectiveness of our proposed framework on different types of workloads and various system utilization levels. By combining temporal and spatial batching, TS-Bat increases the CPU energy savings by upto 68% with various QoS constraints.

# Chapter 4 Information Leakage in Multi-core Server Systems: Characterizations and Defenses

In this chapter, we systematically investigate the information leakage vulnerabilities in multi-core server platforms that could be exploited as covert channels. We target timing channels due to the design of multi-core server architectures. Specifically, Section 4.1 introduces the background of non-uniform memory architectures and cache coherence protocols, which serve critical roles in multi-core processing. In Section 4.2 and Section 4.3 we demonstrate and characterize two newly discovered timing channels: the NUMA-based timing channels and coherence state-based timing channels, and study quantification and defense techniques for these timing channels. Finally, Section 4.4 and Section 4.5 present the related work and summary of this chapter.

## 4.1 Background

### 4.1.1 Non-Uniform Memory Architectures

The memory hierarchy in recent processors includes several levels of caches and DRAM, some that are used privately by the individual cores and some that are shared between multiple cores. Also, with the use of multi-socket CPUs communicating via high speed interconnect (e.g., Quick Path Interconnect (QPI) in Intel architectures [3] and HyperTransport links in AMD architectures [39]), processor cores can now share cache contents across processors. Non-uniform memory architecture is the mainstream multi-processor configuration where each processor has it local cache/memory, and all processors operate on the same physical address space. Figure 4.1 illustrates the cache hierarchy for a dual-socket system with NUMA setting. When processor issues a load or store operation, it will try to find the data block either in local cache or from a remote processor's cache hierarchy. It is possible that the requested data block does not reside in the entire cache hierarchy. In such cases, either the local or remote memory controller will issue a memory access, depending on which one owns

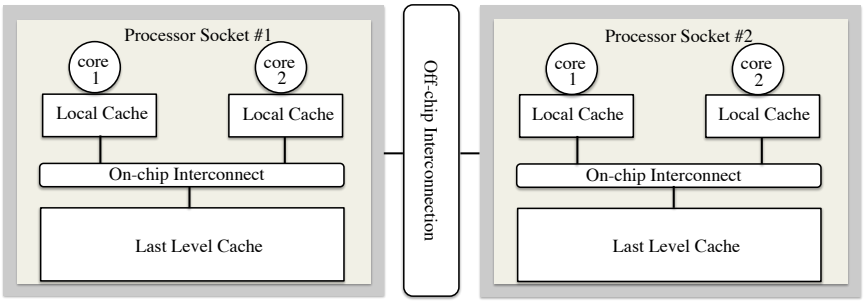



Figure 4.1: Local and remote cache accesses in NUMA system.

the data block.

### 4.1.2 Cache Coherence Protocols

Most modern processors, including Intel Xeon and AMD Opteron families, support *slight* variants of *MESI* cache coherence protocol to preserve data coherence in private caches [3, 39]. The MESI protocol has four states, namely:

1. *Modified* (M) state, where the cache block is present only in one private cache and is dirty, i.e., the data has been modified compared to the value in main memory. This also implies that the current core/processor has write permission to modify the block. 2. *Shared* (S) state, where the cache block is present in more than one private cache and is clean, i.e., the data matches the value in the main memory. This implies that current CPU only has read permission to the block. 3. *Exclusive* (E) state, where the cache block is present only in the current cache, but is clean, i.e., the data matches the contents in main memory. In this state, the current CPU only has read permissions. However, since the cache block is present only in the current cache, it lets the owner CPU to acquire write permissions and upgrade to M state without the need to generate invalidation requests to other cores. Also, any read misses to this block by other cores will downgrade the coherence state in the current CPU to S state. This dual-intent coherence state improves the performance by enabling quick transitions to M or S from the E state depending on the memory operation. 4. *Invalid* (I) state, where the cache block is invalid, and does not have read or write permissions.

Figure 4.2 illustrates the state transitions for the four coherence states in MESI-

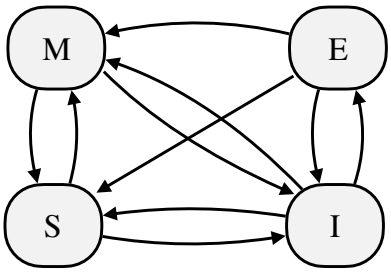


Figure 4.2: State transitions for the MESI protocol

based protocols. Depending on the processor family, there are other specialized cache coherence states to further optimize performance. For instance, the Intel Xeon processor family implements the MESIF protocol, where the F state is meant to designate the processor that will forward the cache block to the requestor. AMD processor family implements MOESI, where O state is created to designate the owner processor after a modified cache block transitions to shared state. This avoids write-back operations to memory whenever the modified blocks are shared between processors. We note that *such additional states simply serve to improve performance, and do not fundamentally add new functionality to M, E, S, or I states*. For clarity, we do not consider such performance-optimizing coherence states in our current work.

Furthermore, Intel Xeon and AMD Opteron support cache coherence across multiple sockets (processors) through high speed links between the processors [3, 38, 39]. This enables multiple processors (multiple CPUs) to share data among them using the underlying coherence protocol.

## 4.2 Information Leakage Attack exploiting Non-uniform Memory Architectures

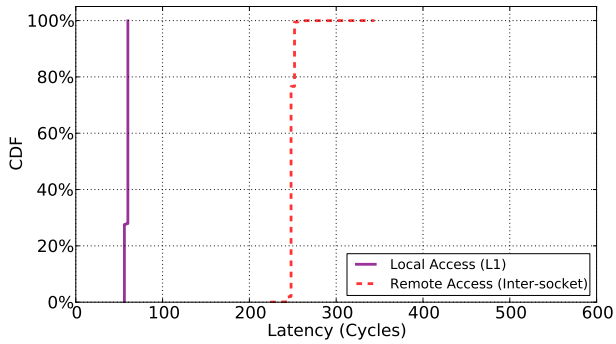
In this section, we demonstrate a new type of covert timing channel attack that exploits the cache access latency differences in NUMA-based architectures. We further propose a statistical metric-Degree of Sparseness- that can effectively quantify the presence of the proposed NUMA covert channels.

### 4.2.1 NUMA Latency Profile

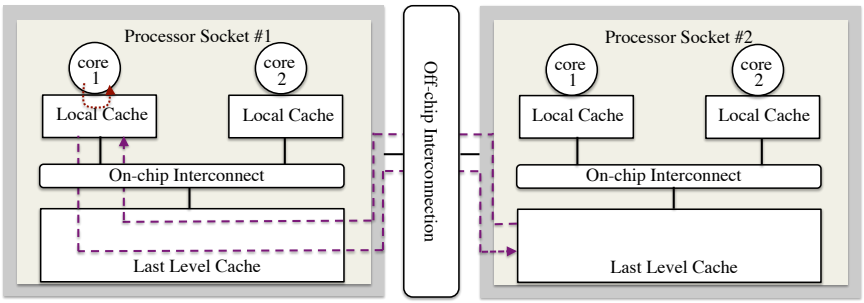
Due to the hierarchical memory levels in NUMA configuration, there is usually a shorter latency period for read/write requests satisfied by a cache situated locally within the processor (local socket) when compared to memory requests that are satisfied by a cache belonging to a different processor (cross-socket or inter-socket). An unintended side-effect of these timing differences in NUMA-based machines is that, a malicious hacker can exploit such timing variations to force data to be placed in different caches, and ultimately implement covert timing channels.

To understand the access latency variations in NUMA, we perform experiments that read data from local and remote processor caches in a multi-socket processor. For this study, we use a dual-socket Intel Xeon X5650 server with 6 cores in each socket running at 2.67 GHz frequency. Each processor has a 32 KB private L1, 256 KB private L2 caches, 12 MB shared L3 cache within each socket. To model real system settings, applications such as browser, dropbox, code editors are run alongside our experiments.

We generate 1,000 memory read operations that target caches in the local socket (specifically, L1 cache) and remote socket (specifically, Last Level Cache or LLC in another socket). To target local caches for reads, our test profiler runs a tight loop with repeated reads that are satisfied by the local L1 cache. To target remote LLC for reads, our profiler issues periodic read requests. Meanwhile, a control program, that is pinned to a socket different from where the test profiler resides, explicitly issues a flush operation (using x86 ISA-supported instructions such as *clflush* that clears the block from all caches) and then loads the block into corresponding socket's cache hierarchy (that includes its LLC). When the test profiler issues its read access to the cache block, this block is guaranteed to be routed to the remote socket where the control program is located. All of the cache accesses and the associated instructions are timed using *rdtsc* instruction. Figure 4.3a shows the Cumulative Distribution Function (CDF) for cache access latencies from local and remote sockets. Figure 4.3b illustrates the difference between cache accesses that are satisfied locally within the




(a) CDF of NUMA access latency




(b) NUMA access patterns

Figure 4.3: Cache access patterns and cumulative distribution function for local (Level 1) and remote/cross-socket (Last Level) cache access latency in NUMA systems.

same socket (local cache hit– shown using a dotted line) versus those that are satisfied by the remote cache (cache hit in another socket– shown using a dashed line). As we can see, the cache accesses to local and remote caches show distinct bands of latency distributions. This indicates the viability for exploiting the latency difference between caches in different sockets.

### 4.2.2 Threat Model

In our timing channel attacks, we assume that the trojan and spy are running on the same machine that features two or more multi-core processors. The trojan intentionally modulates the cache access timings through issuing a flush operation, and later places a data block in its local cache so that the spy can infer the covertly transmitted information. We assume that a compromised trojan, that has sufficient privileges to access sensitive data, is able to run inside the target multi-socket CPU. Our attack model fits into *flush+reload* category of attacks [176], where the trojan

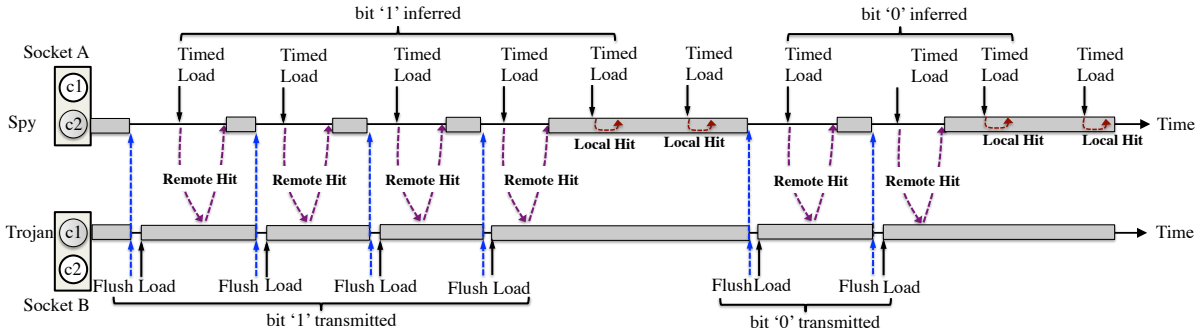



Figure 4.4: Illustration of the communication protocol between the spy and trojan showing a transmission of bit sequence '10'.

clears its memory blocks and reloads them to alter the access times.

As software-level protections continue to offer stronger isolation between applications, hardware structures will become natural targets for covert timing channels. In this vein, we illustrate the vulnerabilities exposed by NUMA-based architectures (specifically, multi-socket CPU systems) that provide for timing difference in cache accesses depending on the caches that satisfy the memory requests.

### 4.2.3 NUMA-based Timing Channel Construction

In this section, we demonstrate a covert timing channel implementation that exploits the cache access timing differences in NUMA. The trojan and the spy are two separate physical processes with the trojan having higher privileges than the spy in terms of access to sensitive secrets such as Operating System- or user-related data. In order to sufficiently modulate the timing of cache accesses, the trojan process is run in a socket different from the spy.

Figure 4.4 illustrates the communication protocol between the trojan and the spy for bit transmission. The spy issues the *load* or read instructions periodically to a memory address that is known (and accessible) to both the trojan and spy. Typically, this memory address points to a shared code region such as a library function shared between processes [176]. There are two possibilities in NUMA-based CPUs when load operations are issued by the spy: 1. the load hits in the local L1 cache or LLC of the spy's socket, 2. the load misses in the cache hierarchy of the spy's socket, and the

requested memory address is resident in the remote socket or DRAM. Our experiments show two distinct bands of latencies for the above two possibilities (Figure 4.3a). The spy *times* these loads to infer whether the loads resulted in a *local cache hit* or a *remote cache hit*. If the loads are satisfied by the DRAM (i.e., neither socket's cache can satisfy), the spy observes a longer latency compared to remote cache hit and ignores them.

The trojan manipulates the access timing to the shared memory address to communicate the bits covertly to the spy. Whenever the trojan wants to communicate a bit, it performs a *cache flush* operation using instructions such as *clflush*. This clears the target memory address from all of the caches that are kept coherent in the multi-socket CPU, including that of the spy located in another socket. The trojan, then immediately, performs a load from that memory location. This populates the shared memory address data in the cache hierarchy of the trojan's socket (i.e., the corresponding L1 cache and LLC). When the spy performs its periodic load from this memory address, the spy's cache cannot service this load since the corresponding contents have been previously flushed by the trojan. Effectively, this enables the spy to detect the timing difference in cache access compared to its local cache hit.

Now that, we have seen how the trojan manipulates the timing of cache access for the spy, it is important to understand how the transmission of '1' and '0' bits can be accomplished. In our current implementation, this is done through trojan performing a specific number of *consecutive* flush+reload operations to be observed by the spy. In our current design, for transmitting a '1', the trojan performs *four* consecutive flush+reload; and for transmitting a '0', the trojan performs *two* consecutive flush+reload operations. The spy infers the end of bit transmission when it observes its load operation result in a local cache hit (via timing the load), i.e., the trojan has not performed flush+reload on the memory block.

Finally, we note that our implementation does not have explicit synchronization between the trojan and the spy. Based on our experimental measurements, a flush+reload operation by the trojan took 350 cycles. To allow for sufficient time, the spy performs its periodic timed load accesses every 1,000 cycles. This minimizes

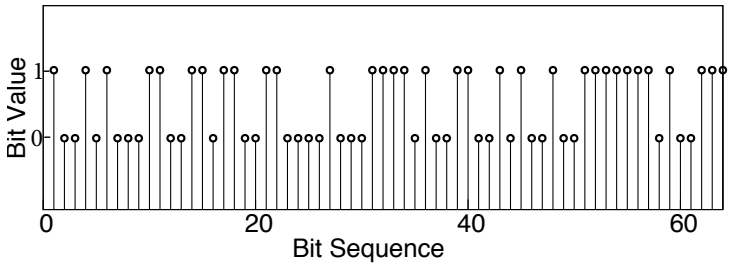



Figure 4.5: Bit pattern (64 bits) transmitted by the trojan.

the possibility of the spy missing the trojan's transmitted bit. We note that, in order to further improve the reliability of transmission while maximizing the effective bit rate of covert channel, the spy could reduce the time interval between its load requests, and the trojan could utilize one or more of the following features: 1. add parity bits, 2. packetize the transmission and include packet headers containing metadata, 3. synchronize using existing system features such as CPU clock.

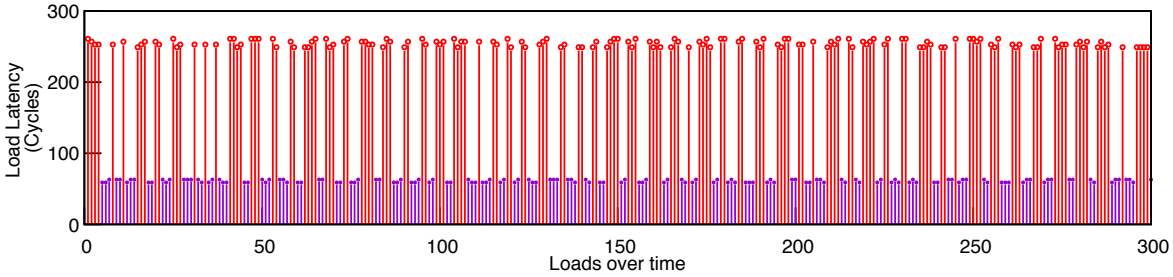



Figure 4.6: Latency sequence for load operations measured by the spy. The (taller) red bar and (shorter) purple bars correspond to remote cache and local cache hits respectively.

### 4.2.4 NUMA-based Timing Channel Demonstration

Using the Linux system call *sched_setaffinity*, the spy and the trojan thread are pinned to two different sockets of Intel Xeon X5650 server. To create shared memory pages, we programmed the trojan and spy to load a shared page from the *libgcrypt library* [2], a widely used Linux cryptography library.

The spy runs a *while* loop with *timed* loads to a cache block in the shared libgcrypt library. Figure 4.5 shows a random 64 bits that are transmitted from the trojan. Correspondingly, Figure 4.6 shows the sequence of latencies measured using *timed*

*load* by the spy. As we can see, there are several consecutive tall bars (corresponding to *remote cache hits*) fenced by consecutive short bars (corresponding to *local cache hits*). The spy deciphers the bit based on the number of consecutive tall bars (see Section 4.2). In our experiments, the spy observes 1-2 tall bars during '0' transmission, and 3-4 tall bars during '1' transmission. This small variance stems from tail latencies in remote cache latency distribution and the background noise from other processes. Despite this minor variation, the spy could still correctly distinguish between '1' and '0' with 100% accuracy. Overall, the transmission achieves an effective bit rate of 190 Kbit/sec. Note that it is possible to further improve the speed through additional optimizations to the trojan and spy described in Section 4.2.

### 4.2.5 NUMA-based Timing Channel Analysis

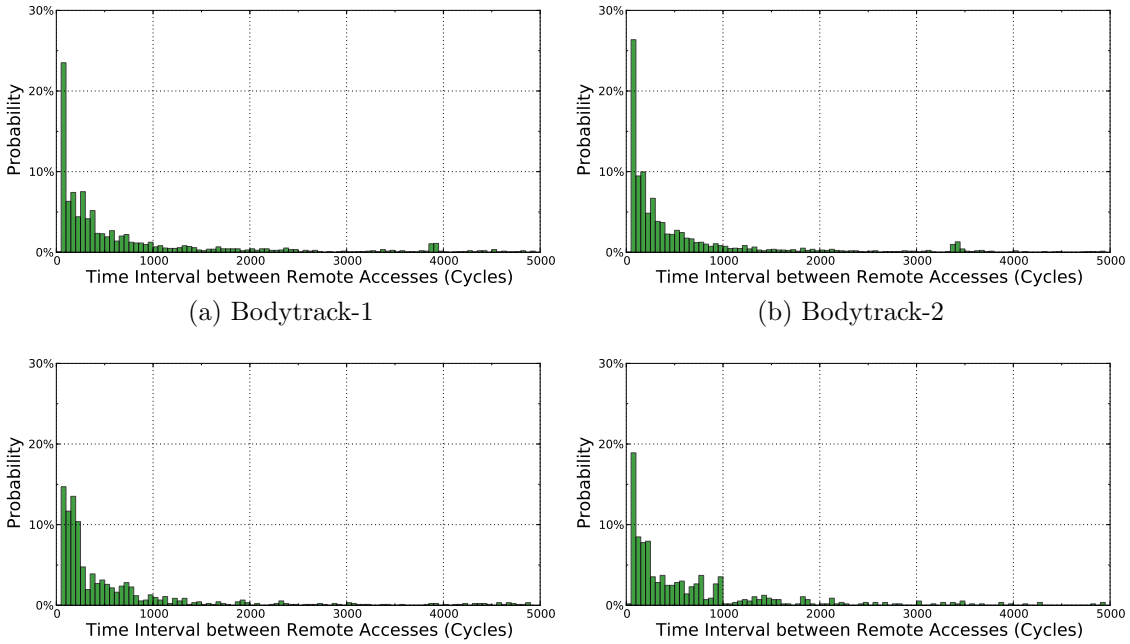
To prepare for an effective defense, we first attempt to characterize the NUMA-based timing channels. We envision that any detection strategy would need to monitor the activities of multiple NUMA components, and capture the interactions between individual components (e.g., cache miss requests and data transfers). Through analyzing the activity of cache hierarchy during covert transmission, we made an important observation: *The time-interval between two consecutive remote cache accesses (i.e., time-interval between inter-socket cache data transfers) exhibits a concentrated distribution in the case of a covert channel attack compared to regular applications with legitimate communication.* The concentration is due to the fact that the spy relies on observing a consecutive number of remote load accesses in order to infer the transmitted bits. As such, the spy issues a number of load operations at a pre-determined sampling interval that can be observed over the interconnect. Our experiments show that covert channels have only a *few possible values for time-intervals between remote cache accesses* (see Section 4.2.5.1 for details). We note that it is conceivable for a spy to change the sampling interval at runtime to evade from being noticed. However, in asynchronous environments, the spy may begin to see a rapid rise in bit error rate due to lack of synchronization. On the contrary, for regular applications that do not intentionally manipulate the timing between inter-socket data transfers, a higher

level of randomness is expected in time between inter-socket cache transfers, which is verified through our experiments.

#### *4.2.5.1 Time-Intervals between Remote Accesses*

Since real hardware does not support measuring the time-intervals between remote cache accesses, we setup *Gem5* [23], a cycle-accurate, full system simulator to perform our measurements. We configure Gem5 with eight x86 cores, and use a minimal Linux distribution with kernel version 2.6.32. We build Parsec-2.1 benchmarks [22] with 8 threads, where each thread is pinned to a separate core. We also configure the spy and trojan to run on Gem5, where we record the timestamps for inter-core cache data transfers. Since any core pair can be used in a covert channel attack, the monitor filters the timestamps associated with cache data transfers for each source-destination core pair. We generate histograms for time-intervals between inter-socket cache accesses.

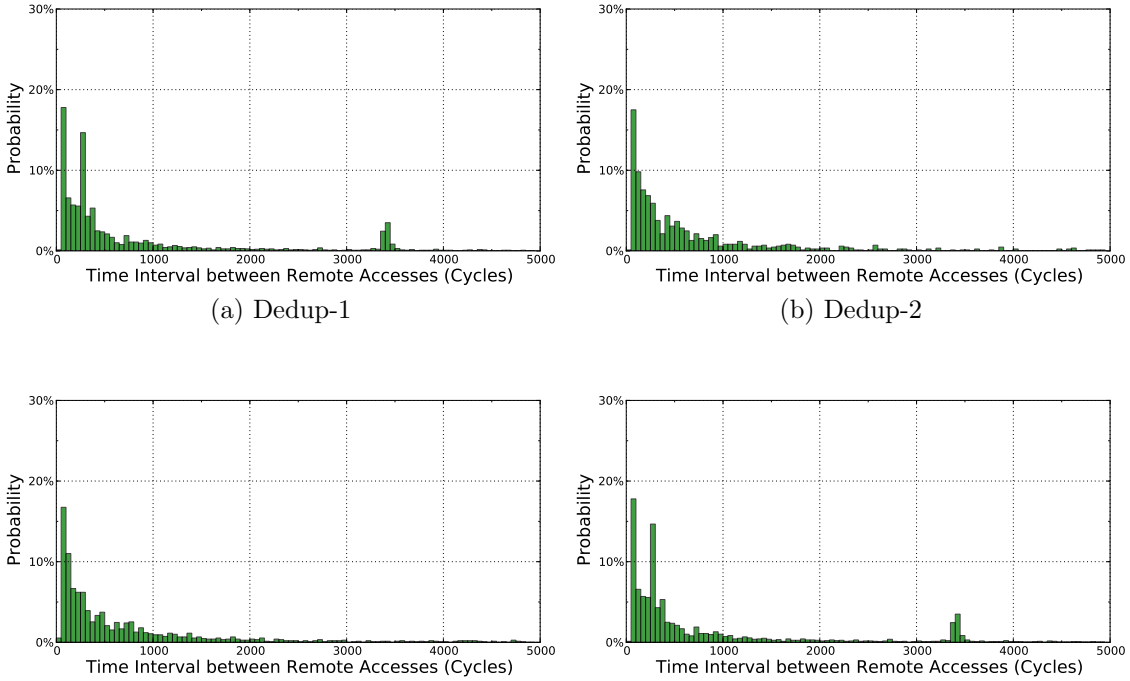
As expected, our experiments in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11, and Figure 4.12 show that regular (Parsec-2.1) benchmarks exhibit higher randomness in time-intervals. We show results for four representative core pairs in each benchmark. Also, we eliminate time-intervals greater than 5,000 cycles since they represent lengthy idle periods without inter-core cache transfers, and collectively constitute less than 0.5% of the population. We observe that, though some time interval bins are more favored than other bins (i.e., higher probability), the probability of any single bin does not exceed 25% and the expected probability in almost all of the bins are non-trivial. In contrast, for covert channel scenario (Figure 4.13), the time-interval distribution is highly concentrated and is non-zero only for a very few bins. Notably, the bins between 1,500 and 2,000 cycles correspond to *bit transmission phases* when more frequent remote cache accesses are observed, and the bins between 4,300 and 5,000 cycles correspond to *idle phases* when time interval between remote cache accesses are long. *We note that, though the absolute locations of the histogram bins corresponding to bit transmission and idle phases may change, the characteristic of concentration in certain bins over others remains, as they are inherently needed to*




(a) Bodytrack-1 (b) Bodytrack-2

(c) Bodytrack-3 (d) Bodytrack-4

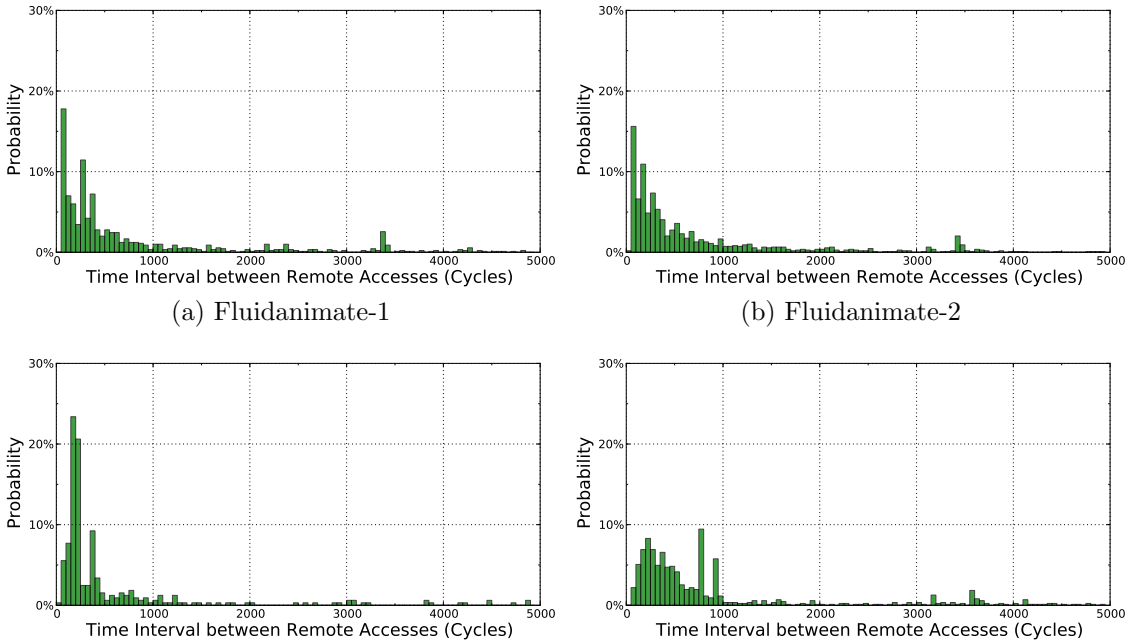
Figure 4.7: Histograms of time-intervals between remote cache accesses in **Bodytrack** for 4 representative core pairs.




(a) Dedup-1 (b) Dedup-2

(c) Dedup-3 (d) Dedup-4

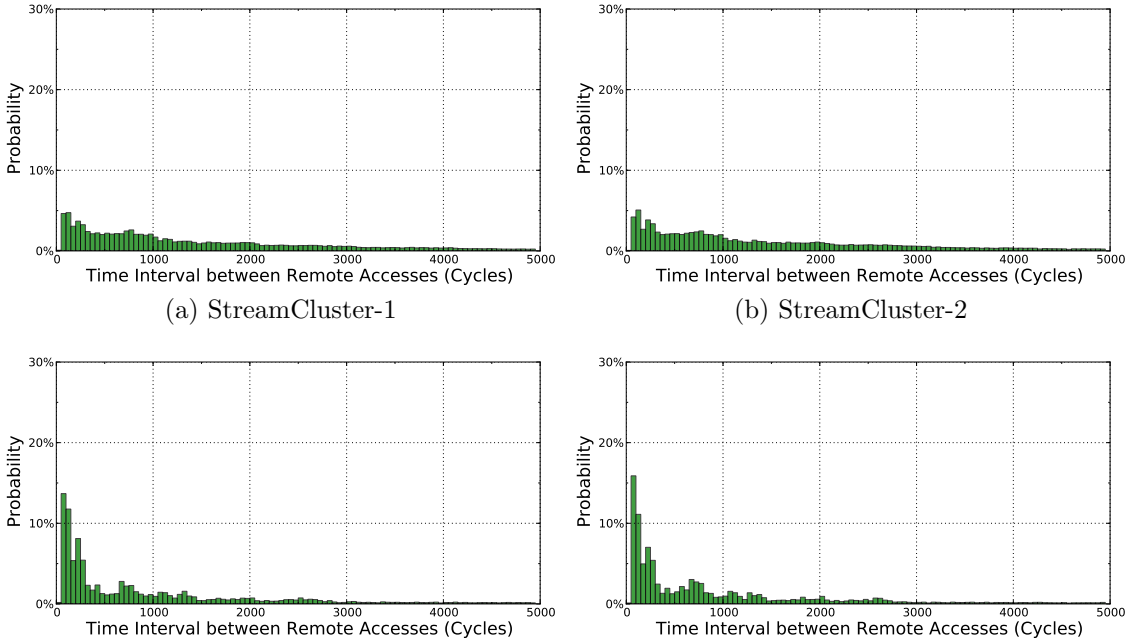
Figure 4.8: Histograms of time-intervals between remote cache accesses in **Dedup** for 4 representative core pairs.




(a) Fluidanimate-1 (b) Fluidanimate-2

(c) Fluidanimate-3 (d) Fluidanimate-4

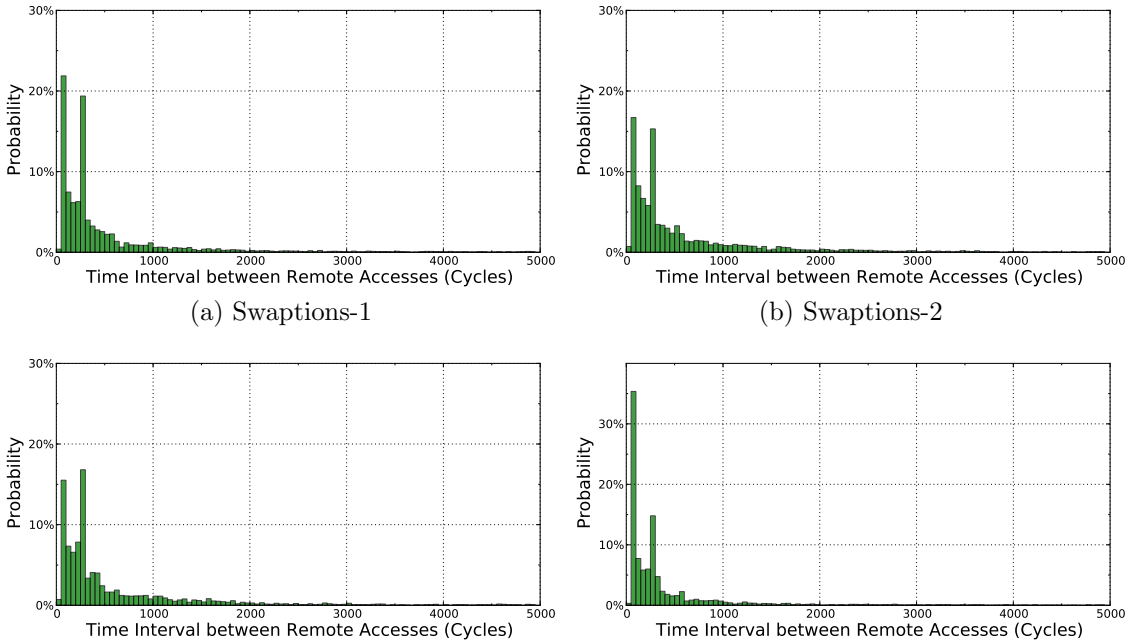
Figure 4.9: Histograms of time-intervals between remote cache accesses in **Fluidanimate** for 4 representative core pairs.




(a) StreamCluster-1 (b) StreamCluster-2

(c) StreamCluster-3 (d) StreamCluster-4

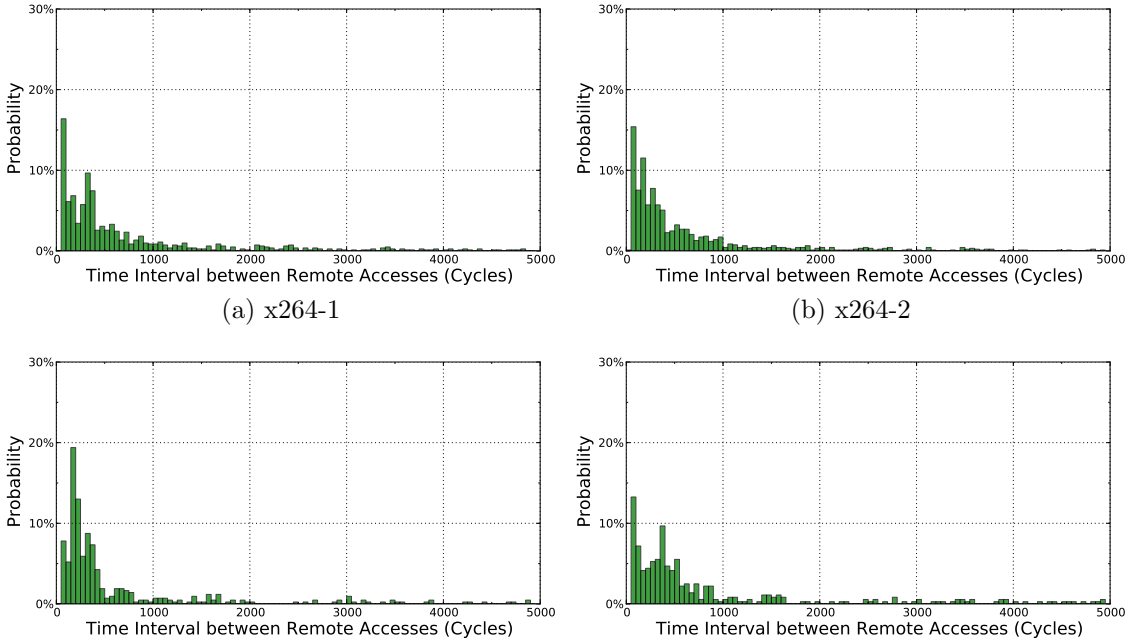
Figure 4.10: Histograms of time-intervals between remote cache accesses in **StreamCluster** for 4 representative core pairs.




(a) Swaptions-1 (b) Swaptions-2

(c) Swaptions-3 (d) Swaptions-4

Figure 4.11: Histograms of time-intervals between remote cache accesses in **Swaptions** for 4 representative core pairs.




(a) x264-1 (b) x264-2

(c) x264-3 (d) x264-4

Figure 4.12: Histograms of time-intervals between remote cache accesses in **x264** for 4 representative core pairs.

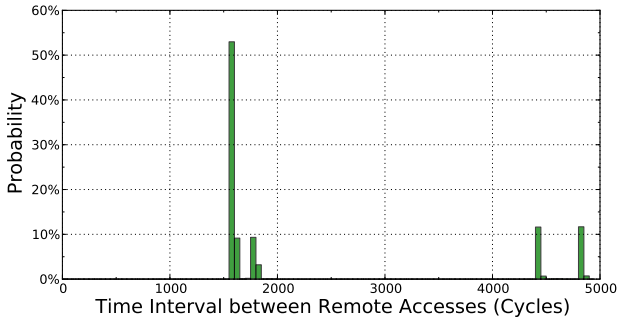



Figure 4.13: Histogram of time-intervals between remote cache accesses in the covert channel.

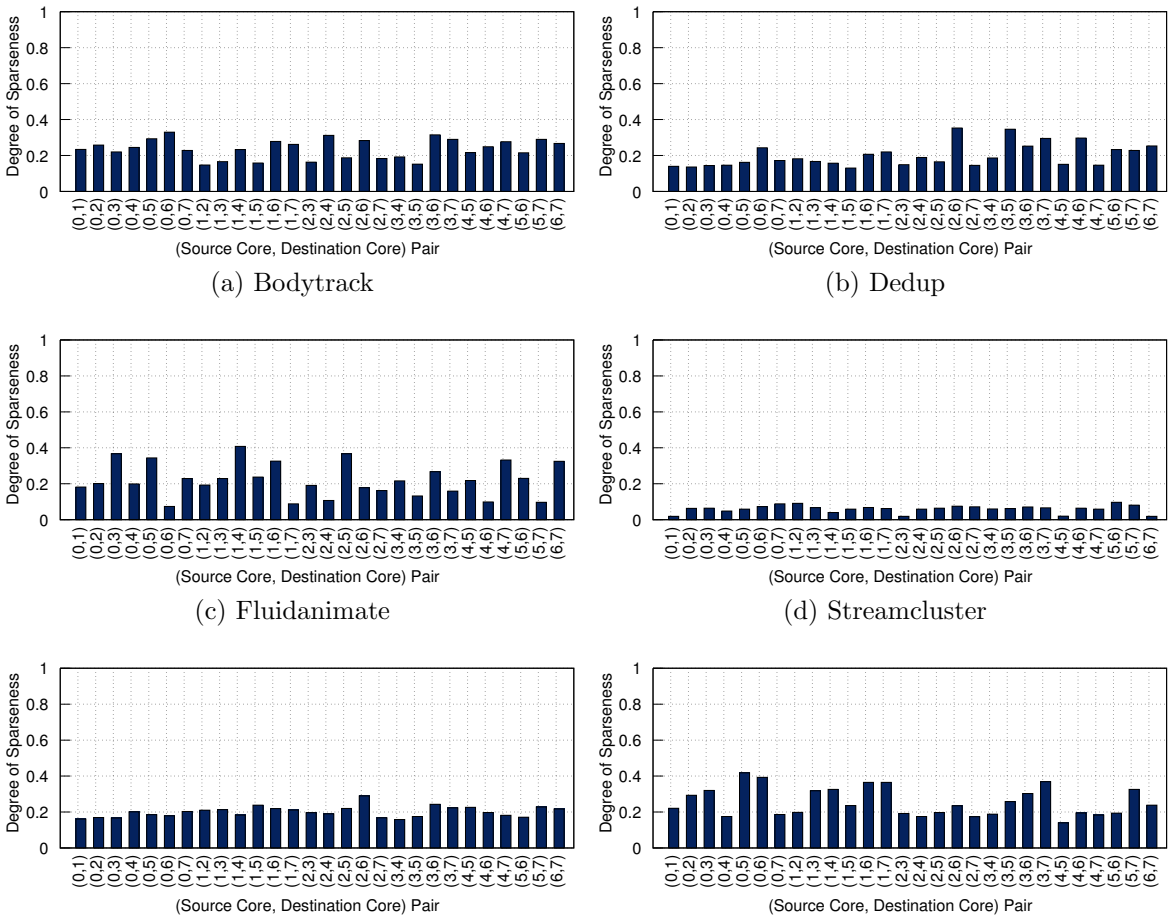
*covertly communicate bits between the trojan and spy.*

#### *4.2.5.2 Quantifying NUMA Covert Channels*

From Section 4.2.5.1, we see that the time-interval histograms for regular applications and covert channels are significantly different. Specifically, the time-interval distribution in covert channel is highly concentrated within a small number of bins to improve *bit inference accuracy* for the spy. In other words, the highly probable bins are *sparse* for covert channels compared to regular applications. Therefore, we use the metric *Degree of Sparseness* ($S$) [103] to capture this phenomenon. An $S$ value of 1 denotes that the distribution is very sparse, and a value of 0 means that the distribution is not sparse (i.e., uniform). Given the probability vector for the histogram bins as $P$, Degree of Sparseness ($S$) is defined as:

$$S = \frac{M}{M - \sqrt{M}}(1 - \frac{\|P\|_1}{\sqrt{M} \times \|P\|_2}) \tag{4.1}$$

where $M$ equals to the number of histogram bins, and $\|P\|_1$, $\|P\|_2$ are norm-1 and norm-2 for vector $P$ respectively. It is worth noting that there are a number of histogram bins in regular applications which have *near-zero* probabilities. Since $S$ depends on the absolute probability values, the histogram bins with near-zero probabilities may be outweighed. To account for such bins and amplify their influence during computation of $S$, we pre-process the probability values with the $\mu$-law com-




(a) Bodytrack (b) Dedup

(c) Fluidanimate (d) Streamcluster

(e) Swaptions (f) x264

Figure 4.14: Degree of Sparseness for (Source, Destination) pairs for the Parsec-2.1 Benchmarks.

pression function [124] to get a new vector $Q$:

$$Q_i = \frac{\log(1 + \mu P_i)}{\log(1 + \mu)} \tag{4.2}$$

where $\mu$ is a tunable parameter that controls the degree of compression for large values while increasing the level of amplification for small values. We set $\mu = 500$ in our experiments.

We compute values of $S$ on our histograms for time-intervals for remote cache accesses. Figure 4.14 shows the *Sparseness* measurement for each of the (source, destination) core pairs for six Parsec-2.1 benchmarks. We observe the $S$ values to be

less than 0.4 in all regular benchmarks. On the other hand, $S$ value is very high, and around 0.8 for covert channels (which is at least 2× compared to regular applications). This proves that our quantification technique can indeed be applied as an effective indicator for the possible presence of NUMA-based covert timing channels.

### 4.2.6 Discussions on Mitigations

As demonstrated above, timings of inter-socket cache data transfers corresponding to the adversaries' manipulation exhibit a unique statistic profile as compared to benign applications that do not intentionally modulate the cache transfer timings. We note that due to the existence of many variants of communication protocols, hardware-based mitigation techniques that attempt to catch all such attacks can be both expensive and inflexible (i.e., unable to guarantee high coverage). Therefore, it is desirable to have agile identification methods before adopting strict damage control mechanisms that can potentially worsen system Quality of Service.

Our statistical quantification methodology can be served as a low-cost and effective first step for mitigating NUMA-based covert timing channels. Notably, our proposed technique only requires timestamp traces for inter-socket cache transfer, which can be easily incorporated in contemporary performance monitoring infrastructures (e.g., PMUs) supported by all major processors. Furthermore, the statistical analysis can be implemented as lightweight software that maximizes robustness and minimizes time to resolution. Finally, real systems can exhibit various levels of noise from many sources of system activities (as we have observed in our experiments). Therefore, to achieve high usability, detection mechanisms should be equipped with effective noise isolation techniques. We envision that future mitigation approaches will potentially employ methodologies from the *signal processing* domain (e.g., [15–17, 20, 48]) to enhance performance in realistic and noisy settings.

## 4.3 Information Leakage Attack exploiting Cache Coherence States

In this section, we demonstrate the profile of cache access latencies for cache blocks in various read-only coherence states. A communication protocol is proposed that manipulate cache coherence states to build a covert channel between a trojan and spy. Additionally, we evaluate the channel capability in each communication scenarios and quantify the bitrate in the presence of noise. Finally, to protect the cache coherence fabric and to avoid adversely affecting the latency-critical read operations, we propose a defense scheme that eliminates the latency differences among the corresponding read transactions. The modified cache coherence scheme is able to defend against the exploits involving cache coherence states.

### 4.3.1 Cache Coherence States and Access Latencies

To understand the effect of cache coherence states and the corresponding cache access latency, we perform experiments that load (read) data in specific cache coherence states (S and E) from specific cache locations (local and remote caches with respect to the requestor). We construct a micro-benchmark with threads that could be pinned to either one or multiple cores. Each requestor thread periodically issues load operations to local and/or remote cache blocks that are in one of the two coherence states: S or E[1]. In this study, we use a dual-socket Intel Xeon X5650 server, each with 6 cores running at 2.67GHz frequency. Each processor has a 32 KB private L1, 256 KB private L2 caches, 12 MB shared L3 cache within each socket. All of the caches are kept coherent in hardware. Our experiments were conducted on a system with a representative workload for a typical desktop server (i.e., applications such as browser, dropbox, code editors were running alongside our code as we made our measurements).

For our measurements, we generate 1,000 memory read (load) operations for each combination pair of (location, coherence state), and time these loads using *rdtsc*

[1]Note that other coherence states such as M may also exhibit different latency profiles. However, change to M state will require writes to the cache blocks. Since writes to shared memory will annul silent page sharing (created using KSM), we do not consider these other states.

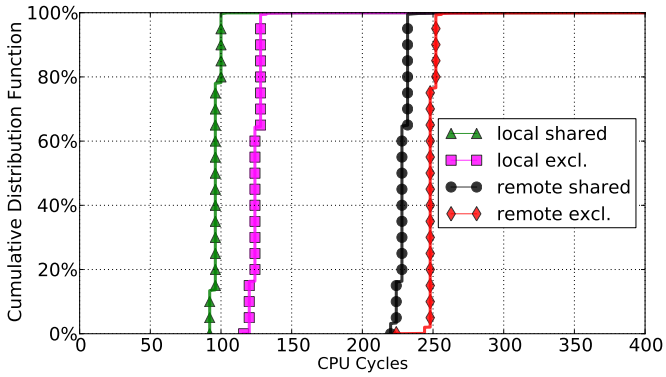



Figure 4.15: Load operation latency in various (location, coherence state) combinations.

| Cache/Coherence State | Min Latency | Average Latency | Max Latency |
|---|---|---|---|
| Local Exclusive | 116 | **124** | 128 |
| Local Shared | 92 | **96** | 101 |
| Remote Exclusive | 244 | **248** | 253 |
| Remote Shared | 220 | **228** | 236 |

Table 4.1: Load operation latency (CPU Cycles) in various (location, coherence state) combinations. Location is with respect to Spy.

instruction. We note that coherence transactions are generated in each case. For example, in *Local Shared* configuration, the requested data is a local L2 cache miss and is fetched from L3 cache in the same (local) chip where the data is present in the *S* state. In *Local Exclusive*, the requested data is local cache miss and is fetched from another core's L1 or L2 cache belonging to the local chip where the data is present in the *E* state. Similarly, in *Remote Shared*, the requested data is present in the S state in the L3 cache of a different (remote) processor chip. In *Remote Exclusive*, the requested data is present in the *E* state in a L2 cache belonging to a remote chip. Figure 4.15 shows the cumulative distribution function (CDF) for the various (location, coherence state) combination pairs. Additionally, Table 4.1 shows the minimum, average and maximum access latencies for the corresponding configurations. Our results show that these combination pairs show distinct bands of latency distributions. We observe that accessing a cache block in the *E* state incurs longer latency than accessing data block in *S* state (e.g., 124 cycles for accessing local E state block and 98 cycles for local S state data block) triggered by cache lookup in different coherence states

(described in Section 4.3.4). Similar latency difference could also be observed for accessing blocks in remote caches as well. Our experiments demonstrate that the latency values are contained within a relatively narrow band for each configuration, and the bands corresponding to different configurations are sufficiently distinct from each other. This clearly demonstrates the viability of exploiting the latency difference between these combination pairs to implement timing channels.

### 4.3.2 Sharing Physical Memory

Before constructing timing channels, the trojan and the spy should first have shared physical memory such that timing of accesses to these addresses can be manipulated. Prior techniques [74, 176] have shown their timing channel implementations by explicitly sharing library code and data between the trojan and spy. In effect, the coherence protocols would maintain states on such blocks to keep a coherent memory view supported by the underlying hardware. Our attack model would work with a similar setup. However, this setup could imply that we assume the trojan (with access to sensitive data) and the spy (that can't access sensitive data) to have shared code or data, which could be difficult in systems where strict isolation guarantee policies are enforced.

We note that a more agile adversary could circumvent the explicit code or data sharing requirement by exploiting a feature called memory deduplication supported by the OS. Kernel Same Page Merging (or KSM) is a kernel feature inside the OS that allows the system to share identical memory pages (i.e., pages with the same memory contents) between different processes. This feature is routinely used to enhance system performance and avoid having to duplicate physical memory pages holding identical data. In current Linux systems, the KSM is a kernel thread that periodically scans the entire memory to identify identical memory pages and make them to be candidates for merging. After the merging process is over, a single physical copy of the page is kept and all of the duplicate copy pages are updated to point to this single physical page in the page table. The physical pages belonging to the duplicate pages are then released back to the system that can be used later for storing more physical

pages with distinct memory contents. The single physical copy (at the end of the merging process) is marked as *copy on write* and resides in *read-only* sharing mode. In other words, write operations to these read-only shared pages are not possible since the kernel will separate them into two separate pages if one of the sharers happen to modify the contents of the page, preventing any unexpected direct communication between the sharer processes. This feature is widely adopted to compact memory, avoid unnecessary memory duplication and reduce memory page misses [21, 134, 154].

From the above description, it is clear that the processes may begin to share pages *unknowingly behind the scenes* due to the OS merging of identical physical pages belonging to different processes. This feature can be now exploited by the trojan and spy to force create shared memory even without explicitly having to share any library code or data between them. In particular, since covert channels are created by colluding parties, the trojan and spy could intentionally generate physical pages with identical bit patterns known to both of them ahead of time. KSM scans the process memory spaces in the order of their starting times (earliest first). To avoid noise from external processes that may accidentally have the exact same bit patterns, the trojan and spy will have to go through a trial communication phase where they perform a series of cache flushes and reloads on this page to make sure that no other process is currently sharing this page as a result of memory deduplication. If an external sharing of this page is detected (via timing measurements), the trojan and spy may repeat creating shared memory through deduplication using another set of identical bit patterns known to both of them.

### 4.3.3 Threat Model

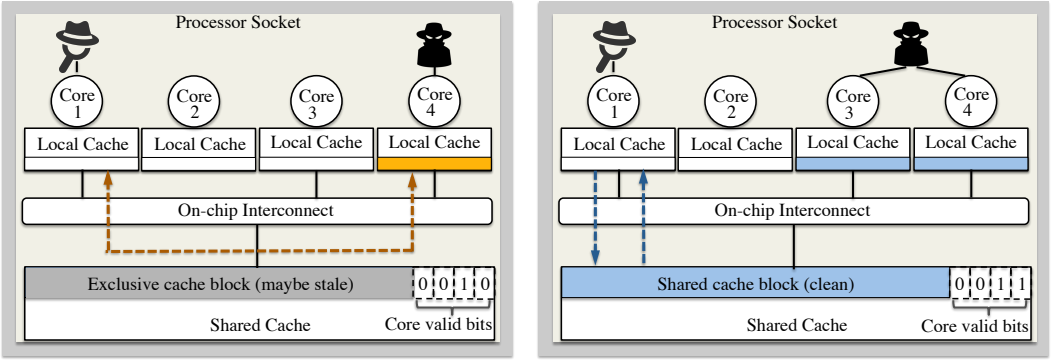
We assume that the trojan and spy share (*one or more*) multi-core processors. The trojan has access to secretive information and it desires to transmit the secrets to the spy. The spy process who does not have accesses to the secrets, will then cause damages to users by exfiltrating the information to the outside world (e.g., identify theft). The trojan can be an application that is downloaded from untrusted sources. Note that the trojan is unable to send secretive data to the outside by itself since it

either lacks sufficient permissions to do so or its transmission activity can possibly be identified and suspended by the system confinement mechanism at the software level. Meanwhile, due to existing isolation techniques such as sandboxing, the trojan and spy processes are explicitly prohibited from any form of direct communication. Such settings exist currently in a variety of real-world scenarios, especially in multi-tenant cloud environment where virtual machines from multiple users can be co-scheduled on the same physical machine to increase resource utilization. Note that compared to the attack model in Section 4.2.2, cache coherence state-based vulnerability expose broader attack surface as it does not necessarily require a remote socket and local socket setting.

The trojan and spy are able to create shared DRAM pages by either explicitly mounting shared libraries or by leveraging KSM to silently merge two identical pages. We note that the latter method is more stealthy since it does not explicitly request memory sharing between the trojan and spy. Also, KSM is widely used in various server systems to improve memory usage efficiency, which is less likely to be disabled due to its high performance advantages. We further assume that the trojan is capable of spawning multiple threads that would run on multiple cores either within the same socket or across multiple sockets. Through this capability, the trojan can intentionally modulate the cache access timing through placing the shared data block in different coherence states and possibly in various levels of the memory hierarchy (local processor's caches, another processor's caches in a multi-processor). The pattern of timing differences between cache block accesses in different coherence states and locations enables a spy to infer trojan's transmission. To synchronize, the trojan and spy can initiate a pre-transmission process where the trojan and spy checks if a pre-determined series of activities (such as flushes) are observed on the shared cache blocks.

### 4.3.4 Exploiting Cache Coherence

In this section, we show some practical ways that the trojan and spy processes can exploit cache latency differences to exfiltrate sensitive data.




(a) Cache block in E state

(b) Cache block in S state

Figure 4.16: Trojan explicitly controlling Cache Coherence States as E or S by running on one or two cores within the multi-core processor. The dotted lines show the service path for a data block residing in E and S states respectively.

#### *4.3.4.1 On-chip Cache Coherence*

Figure 4.16 shows an illustration of the attack using on-chip coherence. Here we assume a multi-core processor where each core has a private write-back cache kept coherent using a variant of MESI protocol, and all of the cores have access to a shared last level cache (LLC).

During a read miss in the private cache, the miss request is first sent to the shared LLC. The LLC maintains the *core valid bits* vector for each block that denotes which of the coherent private caches have a copy of the cache block [5]. An *1* bit value indicates that the corresponding core *caches* that block currently, and a *0* indicates that the corresponding core does not have that block.

If the total number of *1*'s in the vector is greater than one, it indicates that two or more sharers exist for this block. In other words, this denotes that the cache block is in the *S* state in the memory subsystem, and the cache copy in the LLC is *clean*. Since the LLC has a clean data copy, it can directly service the cache miss request from the requesting core.

If the total number of *1*'s in the *core valid bits* vector is equal to one, it indicates that only one cache currently has the block (i.e., owns the block). Note that this cache may have the block in the *E* or *M* coherence states. Also, this may mean that the LLC copy of the block is *possibly stale*, since the current owner could have

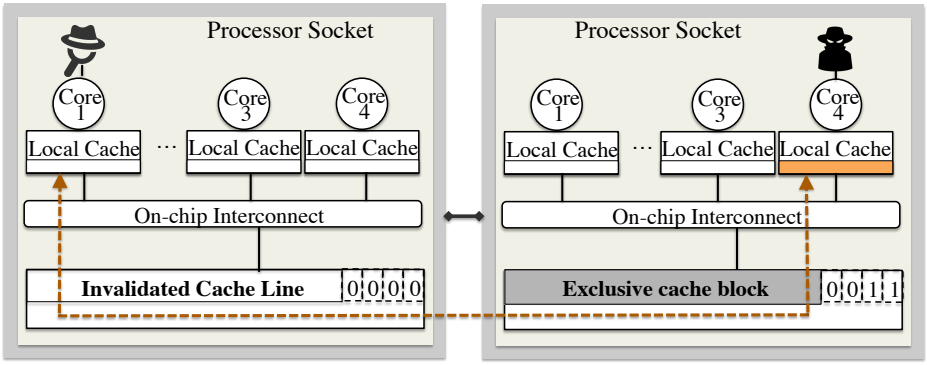
modified the block contents during cache residency. To avoid sending possibly stale data back to the requesting cache, the LLC forwards the cache request to the owner. The owner cache responds to the requesting core with the latest copy of the cache block, and downgrades itself to the $S$ state. The owner also performs a write-back to the LLC to leave a clean copy for future read misses on this block. At the end of this transaction, note that the *core valid bits* vector is updated to reflect the new sharer (the requesting core), and the total number of *1*'s (sharer caches) increases to two.

If the total number of *1*'s in the *core valid bits* vector is equal to zero, it indicates that none of the caches currently have the block. If the LLC has a clean copy of the data (i.e., cache valid is 1), the LLC can service the miss request. Otherwise, the miss request is forwarded to the lower level memory, e.g., DRAM. This case does not generate any coherence activity.

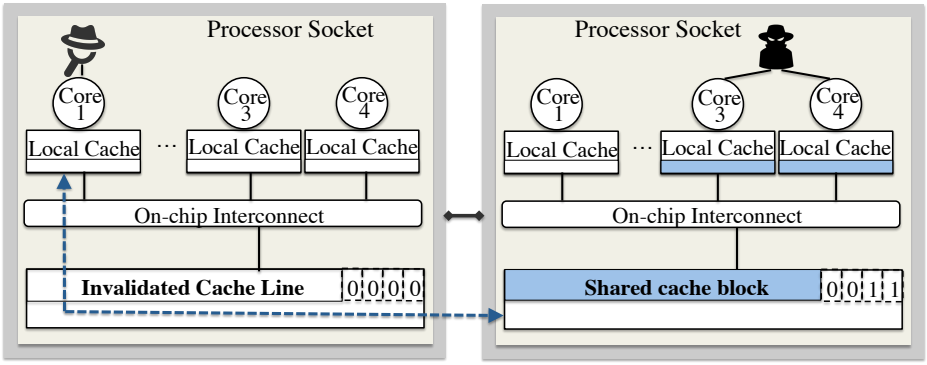
From the above discussion, it is clear that the cache blocks in $S$ and $E$ states result in different read latencies due to the cache miss being serviced by different paths.

In order to communicate covertly, the trojan has to place a cache block, B (that can be read by the spy as well) in either of $S$ or $E$ coherence states, and let the spy observe B's access latency (using rdtsc or an equivalent instruction). The trojan spawns two reader threads on two different cores, and lets both of these trojan threads access the cache block B such that the LLC will record at least two *1*'s in its *core valid bits* vector. When the spy generates a read miss on B, its miss is serviced by the LLC since a clean copy will be available there.

Similarly, to intentionally place B in $E$ coherence state, B will be flushed from all coherent caches. The trojan spawns one reader thread, that will then place a read miss for B. The LLC's *core valid bits* vector will record that only one sharer exists for B. When the spy generates a read miss on B, its miss will routed to the trojan's local (private) cache. The spy's read miss on a cache block in E state creates a different latency profile compared to a read miss on B that is in $S$ state.




(a) Cache block in E state




(b) Cache block in S state

Figure 4.17: Trojan explicitly controlling Cache Coherence States as E or S by running on one or two cores within the multi-socket, multi-core processor. The dotted lines show the service path for a data block residing in E and S states respectively.

#### *4.3.4.2 Inter-chip Cache Coherence*

Many well-known family of processors provide inter-socket cache coherence through high speed point-to-point links, e.g., AMD's HyperTransport bus [39], and the Intel's Quick Path Interconnect [3]. Such high speed links provide for efficient data sharing between the sockets including the ability to maintain coherence between the caches.

The inter-socket coherence works similar to the on-chip cache coherence (see Section 4.3.4.1) with slight modifications to how the data miss requests are routed. When a core requesting a cache block B generates a read miss and the corresponding core's LLC does not have B, the read miss request is sent to other remote sockets first instead of DRAM.

If B is in *S* state in a remote socket, then a clean copy of B is present in the

corresponding remote LLC. The data reply is sent back from this remote LLC to the requesting core's LLC that is then propagated up the memory hierarchy to the requestor core. If B is in $E$ state in a remote socket, then the corresponding remote LLC routes the data miss request up to the current owner (remote) core, which then responds with the data reply. The current owner (remote) core then downgrades its cache copy to $S$ state.

Similar to covert timing channels exploiting on-chip coherence states, the trojan-spy pair can exploit block B's presence in $E$ or $S$ states in remote caches and the resulting access timing differences. Figure 4.17 shows an illustration of this exploit. To explicitly place a block B in $S$ state on a remote cache, all existing copies of B must be flushed from all of the caches (through *clflush* or an equivalent instruction, or through eviction of all the ways in the set [95]). The trojan spawns two threads of itself on one of the sockets participating in hardware cache coherence, and places a block in $S$ state. On a different socket, the spy spawns its thread, and generates a read miss to B to observe its access latency. To explicitly place a B in $E$ state on a remote cache, all existing copies of B are flushed. The trojan spawns its thread on one of the coherent sockets, and places the block in $E$ state similar to how we described for the on-chip scenario (see Section 4.3.4.1). On a different socket, the spy spawns its thread, and generates a read miss to B in order to observe its latency. By exploiting this latency difference, the trojan can modulate the access latency to the block and communicate the data to the spy.

We note that inter-chip cache coherence offers a higher degree of flexibility for the trojan and spy by providing *more* possibilities for varied cache access latency. This is because, through supporting inter-chip coherence, each socket offers intra-chip coherence as well. This implies that the trojan can now place the block in one of $S$ or $E$ states either within the local or a remote processor chip, that in turn provides more distinct bands of latency values for the trojan-spy to exploit.

### 4.3.5 Timing Channel Construction

In this section, we describe the trojan and spy construction process, and show how they would exploit cache access latency differences created by combination pairs of cache location and coherence state associated with the cache block.

We illustrate a template for the trojan and spy that can be eventually integrated into a *real-world adversarial setting* designed to exfiltrate sensitive secrets. For example, let us consider a scenario where a spy process has the ability to observe encrypted communication transmitted over a public network between two processes with access to sensitive information. As per the system security policy, the spy cannot directly communicate with either of these entities due to it being on lower security stratum, nor can it decipher the communicated bits without knowing the decryption key. However, a malicious insider trojan (that has access to secrets) could collude with the spy to circumvent the system security and communicate secrets covertly as follows:

1. To compromise symmetric cryptography techniques (e.g., AES, DES), a trojan transmits symmetric encryption/decryption key covertly to the spy through modulating accesses to the coherent caches on shared physical memory blocks. With the already captured encrypted text and the now-obtained decryption key, the spy could *covertly* receive the message without any direct communication with the trojan.

2. To compromise asymmetric encryption standards (e.g, RSA), a trojan and the spy intentionally sign up for the RSA service under the pretense of encrypting their own texts. Since trojan-spy share the same coherence fabric, the trojan could covertly transmit its decryption key through modulating accesses to the coherent cache. The spy could decrypt the encrypted text and gather sensitive data.

#### *4.3.5.1 Pre-transmission*

In order to construct the convert timing channel using cache coherence states, there are two considerations: 1. shared physical read-only memory between trojan and spy to enable covert communication (Section 4.3.2), 2. synchronization between trojan and spy prior to transmission.

In our experiments, read-only shared memory is implicitly created through KSM when the trojan and spy intentionally write identical data to their individual pages with a deterministic, pseudo-random number generator function that begins with the same seed. Specifically, the trojan and spy create shared memory as follows: 1. Allocate memory through system calls such as *alloc()* and populate them with identical contents. The allocated pages with similar content are merged through invoking the system call *madvice()*. 2. The trojan and spy will wait for a certain period (e.g, 30 seconds) for the merging process to be complete. We note that this creation of shared memory needs to be done exactly *once* prior to entire trojan-spy communication. In very rare occasions, where a third independent process has its page merged with the memory page that is actively utilized by trojan/spy for covert communication, we have to discard such a page, and create another shared page that will be uniquely accessed by *just* the trojan and spy without external interference. Such situations can be prevented by creating a *spare* shared page initially, thus avoiding any necessity to re-invoke KSM.

For synchronization, the trojan issues flush of the shared cache block and then reloads the same block continuously for a number of times (e.g., about 20 in our experiments), and the spy periodically issues load instruction to the same cache block. The trojan and spy time their respective load instruction latency. Synchronization is considered complete when the trojan observes a series of long latencies because of having to load data from memory, and when the spy notices a sequence of latencies eventually converging to a stable band of values. This process indicates that the trojan and spy uniquely share a block and that the spy is able to decipher the block's presence in trojan's cache through timing the cache block accesses. The actual transmission can start once the synchronization is successful. Our experiments show that it takes, on average, 90 milli-seconds for trojan-spy synchronization. We note that this step needs to be performed prior to covertly transmitting the first bit or after every OS context switch that involves either the trojan or the spy.

**Algorithm 4:** Trojan Communication Protocol

```
   Input: read-only cache block: B, Txbit[], CS_c, CS_b;
   /* CS_c is the coherence state used in bit communication          */
   /* CS_b is the coherence state used for bit boundary              */
 1 spawn trojan threads;
 2 synchronize with spy using shared cache block, B;
   /* B could be created implicitly via KSM or through explicitly shared
      data or library code                                           */
   /* Spy-trojan communication protocol defines three counters: C_1, C_0
      and C_b for communicating 1, 0 and boundary respectively       */
 3 i = 0;
 4 while Txbit[i] != -1 do
 5     Repeat C_b times: put B in CS_b state;
 6     if Txbit[i] == 1 then
 7         Repeat C_1 times: put B in CS_c state;
 8     else
 9         Repeat C_0 times: put B in CS_c state;
10     i++;
```

#### *4.3.5.2 Trojan and Spy*

To implement covert timing channels using coherence states, the trojan and spy pick a (location, coherence state) combination pair to modulate timing and communicate bits (1 or 0), and another distinct (location, coherence state) combination pair to delineate bit transmission *boundaries* (i.e., to say that a bit transmission has ended and another will start at the end of boundary). These two combination pairs are denoted as $CS_c$ and $CS_b$ respectively, where $c$ stands for communication and $b$ denotes boundary. Correspondingly, we assume that the bands of cache access latency values $T_c$ and $T_b$ are already known to the trojan and spy through self-measurements on cache hardware (Figure 4.15). Within the bit transmission period, the trojan and spy will also know how many consecutive times a block B will be seen in $CS_c$ state to distinguish between the transmission of bit values '1' and '0', denoted by $C_1$ and $C_0$ respectively. We note that having distinct communication and boundary values remove the need for synchronization on each bit transmission.

Algorithm 4 describes our implementation for the trojan. The trojan is multi-threaded to *explicitly* control the placement of blocks in S or E state either locally or

**Algorithm 5:** Spy Communication Protocol

```
   Input: read-only cache block: B, Tvalues[]=-1;
   /* Two access latency bands, T_c and T_b; T_s is the sampling interval  */
   /* wait for the trojan to synchronize with trojan using shared cache
      block, B                                                             */
   /* B could be created implicitly via KSM or through explicitly shared
      data or library code                                                 */
 1 while true do
 2     flush B from cache;
       /* wait for T_s sec until trojan has an opportunity to reload       */
 3     load B and time the load (T);
 4     if T is within T_b then
           /* transmission has started                                     */
 5         break;
 6 //reception period
 7 i = 0;
 8 while true do
 9     flush B from cache;
       /* wait for T_s for trojan to reload                                */
10     load B and time the load (T);
11     record T into Tvalues[i++];
12     if T is outside of T_c and T_b for N consecutive times then
           /* N is defined by the trojan and spy                           */
13         break;
14 //translation period (interpret 1's and 0's) read Tvalues[] vector from index 0 to N;
15 i = 0; j = 0; k=0; count[] = 0;
16 while Tvalues[i]! = -1 do
17     Repeat until Tvalues[i] is within T_b band: i++;
18     bit_c = 0;
19     Repeat until Tvalues[i] is within T_c band: bit_c++; i++;
20     count[j++] = bit_c;
   /* Thold, Threshold separates C_1 and C_0 and helps decipher bits       */
21 while count[k] != 0 do
22     if count[k++] > Thold then
23         //Infer that the transmitted bit is 1;
24     else
25         //Infer that the transmitted bit is 0;
```

remotely. For every '1' bit to be transmitted, it puts the cache block in $CS_c$ coherence state for $C_1$ times, and for every '0' bit transmission, the trojan places the cache block in $CS_c$ for $C_0$ times. In-between every bit transmission, the trojan places the cache block in $CS_b$ for $C_b$ times to denote bit boundaries.

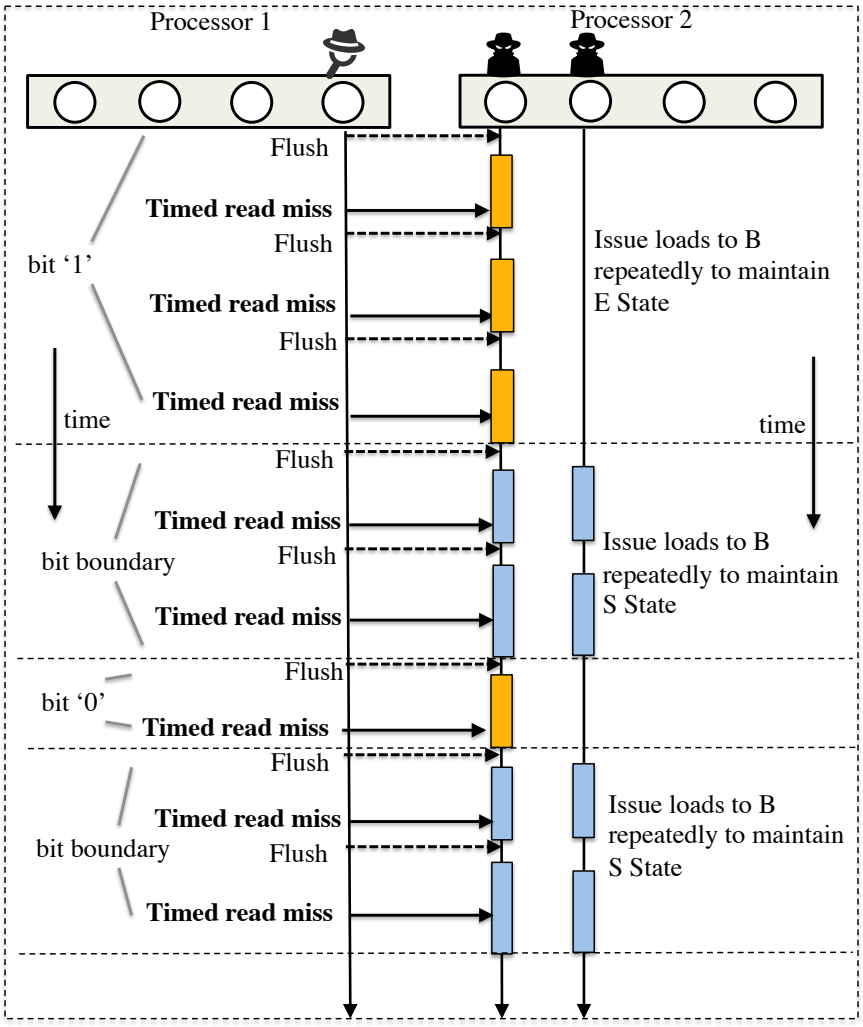



Figure 4.18: Illustrative example of '1' and '0' transmission protocol between trojan(s) and spy.

The spy process is a single-threaded observer that times the cache block accesses using repeated patterns of flushes and reloads on them. Algorithm 5 describes our implementation for the spy. We see that the spy has three phases: 1. Polling for start of transmission by repeated flush and reload of a shared block B. 2. Reception of transmitted bits by timing each access to B, and recording latencies into *Tvalues[]* vector. 3. Translation of *Tvalues[]* by accumulating the consecutive T values belonging to the same band, and distinguishing them into bits '1' and '0', and 'bit boundaries'.

Figure 4.18 gives a diagrammatic illustration of an *example communication pro-*

| Cache Location and Coherence State for bit communication and boundary | Notation | Number of Trojan threads |
|---|---|---|
| (Local Exclusive, Local Shared) | $LExcl_c - LShared_b$ | 2 (local) |
| (Remote Exclusive, Remote Shared) | $RExcl_c - RShared_b$ | 2 (remote) |
| (Remote Exclusive, Local Exclusive) | $RExcl_c - LExcl_b$ | 2 (1 local, 1 remote) |
| (Remote Exclusive, Local Shared) | $RExcl_c - LShared_b$ | 3 (2 local, 1 remote) |
| (Remote Shared, Local Exclusive) | $RShared_c - LExcl_b$ | 3 (1 local, 2 remote) |
| (Remote Shared, Local Shared) | $RShared_c - LShared_b$ | 4 (2 local, 2 remote) |

Table 4.2: Trojan implementation along with states used for bit communication and boundary. 'Remote' and 'Local' are with respect to the spy's location.

*tocol between the trojan and spy.* In this example, the trojan is located in a processor different from that of the spy (Note that the trojan and spy could be in the same processor as well). The trojan modulates the cache access timing for the spy by placing a block B in $E$ state when it wants to transmit a bit, and through placing B in $S$ state to indicate boundaries between bits. The trojan spawns 2 threads on the remote socket, and issues load requests to B from just one thread to explicitly place it in $E$ state and issues load requests to B from both threads to explicitly place it in $S$ state. In particular, between cache block flushes initiated by the spy, the trojan places B in $E$ state for 3 consecutive times to signal a '1' bit, and places B in $E$ state for just 1 time to signal a '0' bit. For bit boundaries, the trojan places B in $S$ state for 2 consecutive times between flushes initiated by the spy.

Table 4.2 shows 6 cases where the trojan and spy use two distinct (location, coherence state) combination pairs for bit transmission and bit boundary identification. The location identifiers 'local' and 'remote' are with respect to the spy, since it measures the load latencies and deciphers the bit values/boundaries on its end.

### 4.3.6 Experimental Results

We conduct experiments on a Intel Xeon X5650 2-socket server with a total of 12 cores, the configuration described in Section 4.3.1. We pin the trojan and spy threads onto specific cores using the *sched_setaffinity* API. All of the reported load latencies were obtained by inserting the *rdtsc* instruction. We implement the 6 attack scenarios listed in Table 4.2 and study their bandwidths. Additionally, we implement a covert

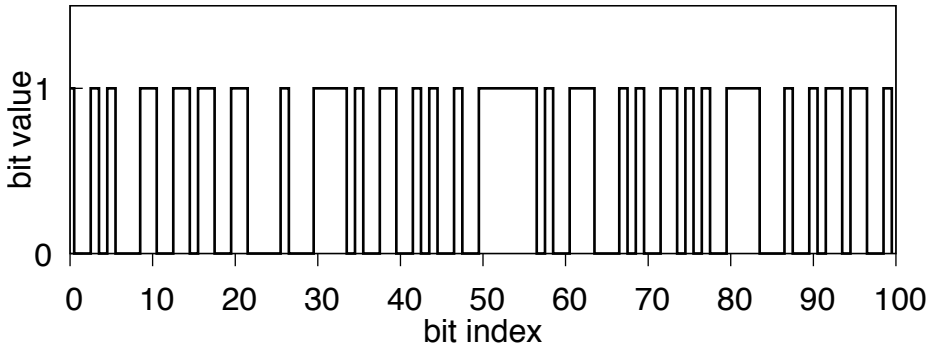
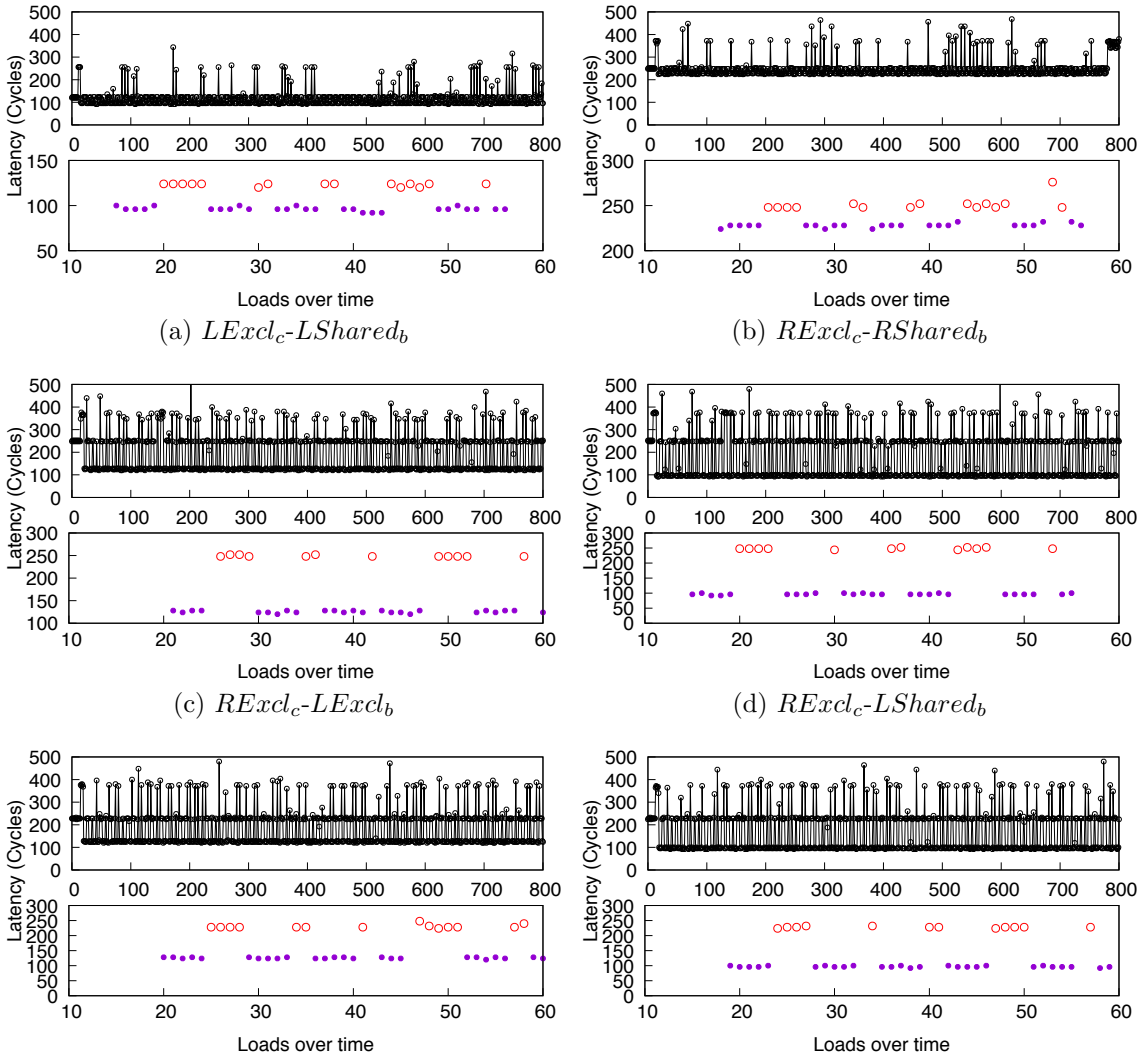



Figure 4.19: Bit pattern (100 bits) covertly transmitted by the trojan.

timing channel with *symbols encoding multi-bits* by leveraging combination pairs of (location, coherence state) and encoding data in *larger-than-binary* representations.

#### *4.3.6.1 Spy's Reception*

Figure 4.19 shows the secret (bit) pattern that the trojan intends to covertly communicate with the spy. Figure 4.20 shows the results of load latencies observed on the spy side. For each combination pair of (location, coherence state), we show two sets of results: the top portion shows the load latencies observed throughout the entire reception period, and bottom portion shows a magnified view illustrating the communication of the first five bits in the top figure for clarity. In this magnified view, we observe that for each '1' bit transmitted, the spy observes the load latency in the $T_c$ band, corresponding to $CS_c$, for four or five consecutive times (each dot in the figure denotes a 'timed' load operation); for each '0' bit transmitted, the spy observes load latency in the $T_c$ band for one or two consecutive times (See discussion in Section 4.3.5.2). These are shown as red dots in the bottom portion of each figure. Similarly, the boundary between bit values are deciphered by the spy when it observes load latency in the $T_b$ band, corresponding to $CS_b$, for four to five times consecutively. Our experiments show that the spy is able to correctly decipher the transmitted bits for all 6 attack scenarios with 100% accuracy.




(a) $LExcl_c$-$LShared_b$ (b) $RExcl_c$-$RShared_b$

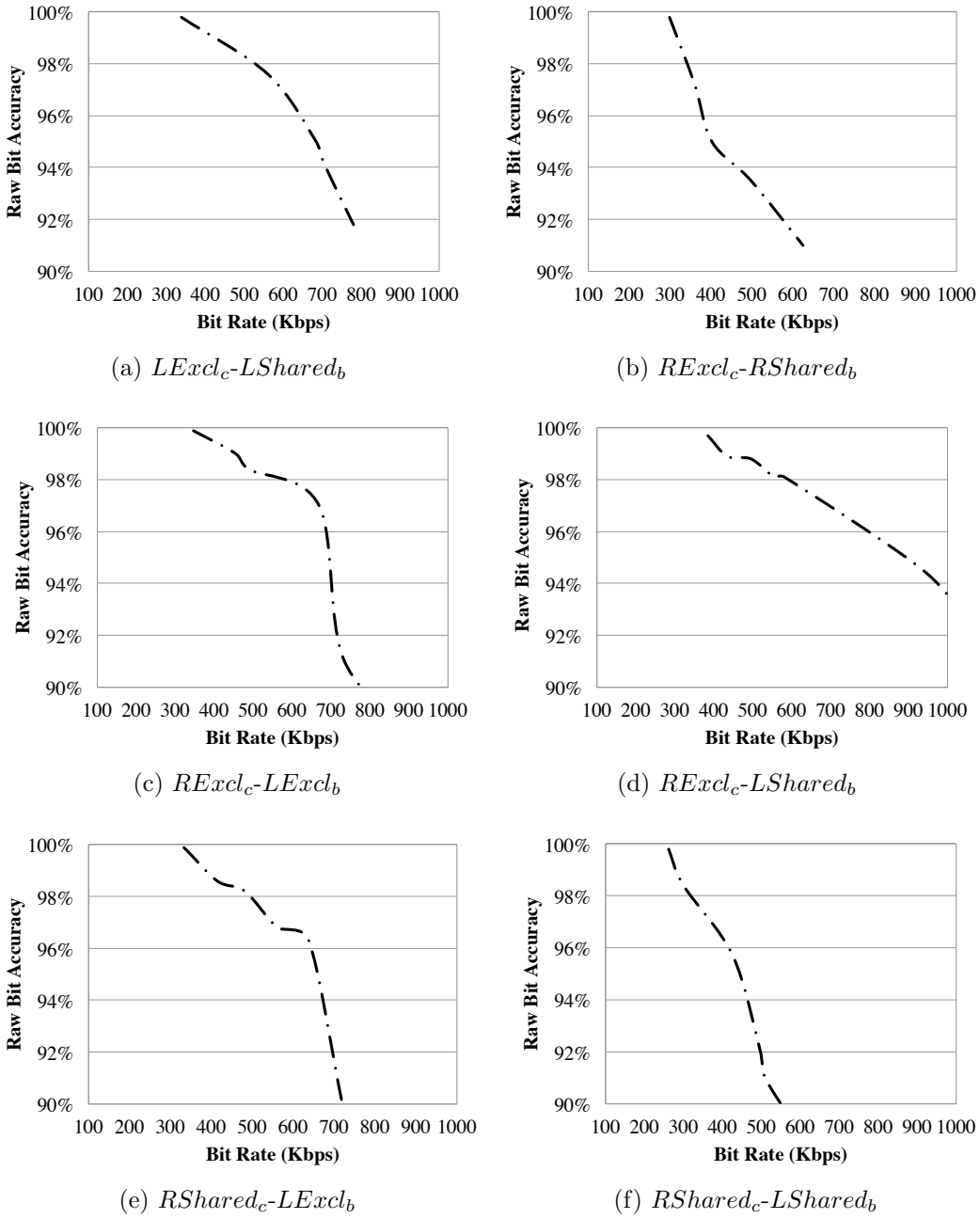
(c) $RExcl_c$-$LExcl_b$ (d) $RExcl_c$-$LShared_b$

(e) $RShared_c$-$LExcl_b$ (f) $RShared_c$-$LShared_b$

Figure 4.20: Bit Reception by the Spy (corresponding to the bits transmitted in Figure 4.19) through measuring load latency (in CPU cycles). The top portion in each subfigure shows the entire reception period, and the bottom portion shows a magnified view for the reception of first five bits.

#### *4.3.6.2 Transmission Bandwidth*

We conduct experiments to study the raw bit accuracy with increasing transmission bit rates between the trojan and spy. We perform this study by tuning two knobs: 1. Reduce the number of consecutive caching operations for shared blocks that communicate bit values and boundaries, i.e., values of $C_1$, $C_0$ and $C_b$. 2. Reduce the interval between shared cache block loads by the spy, i.e., the value of $T_s$. Refer




(a) $LExcl_c$-$LShared_b$ (b) $RExcl_c$-$RShared_b$

(c) $RExcl_c$-$LExcl_b$ (d) $RExcl_c$-$LShared_b$

(e) $RShared_c$-$LExcl_b$ (f) $RShared_c$-$LShared_b$

Figure 4.21: Raw bit accuracy as captured by the spy with increase in transmission rates.

to Algorithms 4 and 5 for further details on these parameters. Figure 4.21 shows our results. In this study, we note that there are 3 possibilities for raw bit error on the reception side: 1. certain bits may be lost, 2. extra bits may be added due to duplication (very rare and we did not observe any such occurrence in our exper-

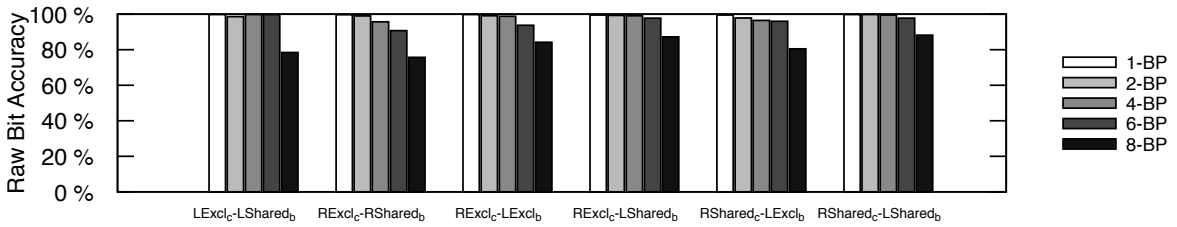



Figure 4.22: Raw bit accuracy captured by the spy when co-located with external processes (kernel-build [1]).

iments), and 3. certain bits may be flipped (1 mis-interpreted as 0, or vice versa). Accuracy is defined as the ratio of number of raw bits correctly received by the spy to total number of raw bits transmitted by the trojan. As we increase the bit rate to beyond 500 Kbps, we see that most cases experience a rapid drop in raw bit accuracy. However, there are two exceptions: 1. $RExcl_c - LExcl_b$ begins with a high initial bit rate of over 400 Kbps and declines to below 90% accuracy only beyond 800 Kbps. 2. $RExcl_c - LShared_b$ shows high immunity and a good raw bit accuracy of 96% even at 800 Kbps. We note that the *effective 'information bit' accuracy rates* can be kept potentially high by leveraging higher raw bit transmission rates especially when the underlying transmission protocol incorporates error correcting codes. Methods to recover information bits due to omission and bit flips is a well studied topic [54], and is outside the scope of our work.

#### *4.3.6.3 External Noise and Error Correction*

To observe noise effects from co-located memory-intensive applications, we run a highly memory-intensive workload, kernel-build [1], that compiles a Linux kernel to benchmark a system or test its stability. This application supports a variety of options including multi-threaded implementation. Note that this experiment simulates an *extreme stress-test* case where a very high memory-intensive multi-threaded workload is co-located with trojan/spy. In this setting, alongside our trojan and spy processes, we spawn a different number of kernel-build threads (1 to 8). Figure 4.22 shows our experimental results where we observe that, with the increase in number of memory-intensive threads, the bit accuracy levels on the spy side experience a range

of degradation.

Specifically, even with six background processes, the spy processes in all of the 6 attack variants are able to achieve fairly high bit accuracy (above 90% on average). However, with 8 external kernel-build processes, we see an observable impact on the trojan-spy communication (11% to 23% increase in raw bit error rate). Meanwhile, we observed subtle differences between different cases. For example, since kernel-build processes saturate the internal bus (L2-LLC) bandwidths, load latency values to E state blocks in remote caches were highly varied while remote LLC accesses (S state blocks) do not suffer from high latency swings when measured by the spy.

To illustrate mechanisms that can improve bit accuracy under noise, we propose and implement a simple error encoding and retransmission protocol. For each packet (64 bytes), 16 parity bits are added to catch any bit flips within 4 Byte chunks. After each packet transmission, the spy checks for parity bits and if errors are detected, it will request for packet resend by covertly transmitting NACK bit. This is achieved by reversing the roles of spy as the transmitter and trojan as the receiver just for transmitting the NACK bit. This process is repeated until successful receipt of the packet. Figure 4.23 shows the achievable bit rates for trojan-spy transmission without noise and the effective rate with retransmission scheme under medium noise (with 4 kernel-build processes) and high noise (with 8 kernel-build processes) levels. Overall, we can see that the retransmission scheme suffers less than 10% reduction in transmission rate, and incurs 24% worst-case reduction in transmission rate under high noise levels in return for guaranteeing 100% bit recovery. Conceivably, the same NACK mechanism can be used to track non-reception by the spy as well. If the spy was context switched out, the trojan will not receive acknowledgment packet (NACK bit), and hence will retransmit until a successful acknowledgment (NACK=0) is received.

Our experimental results provide a useful insight that the covert timing channels introduced due to coherence states can be robust in terms of bit accuracy and high transmission rates. Also, incorporating even a fairly simple error detection and retransmission scheme can significantly improve bit accuracy with a relatively small impact on peak bit rate.

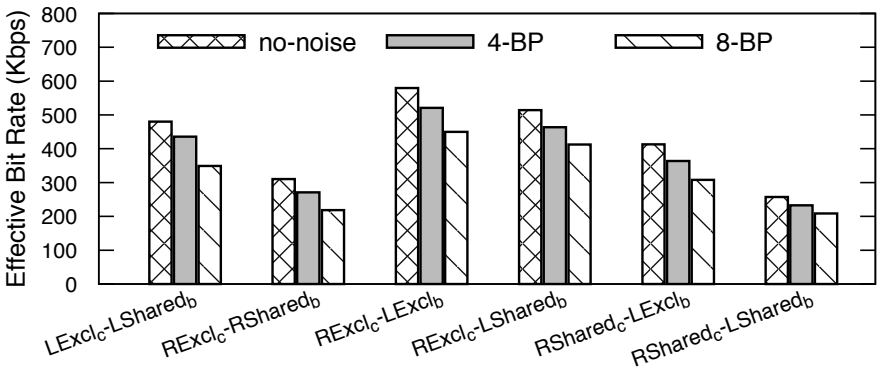



Figure 4.23: Effective information bit transmission rate with error correction scheme under medium (4 co-located kernel-build processes) and high (8 co-located kernel-build) noise levels.

#### *4.3.6.4 Symbols Encoding Multi-bits*

Besides just increasing the transmission speed, the volume of information transmitted by a covert channel can be increased by encoding multiple bits using symbols. Due to the presence of multiple distinct latency bands corresponding to (location, coherence state) combination pairs, we implement a covert timing channel that transmits symbols encoding 2-bits in every transmission. We utilize four combination pairs ($RExcl_c$, $LExcl_c$, $RShared_c$, $LShared_c$) to encode one of four distinct symbol values. The spy infers the symbol by issuing load instructions (similar to our algorithm in Section 4.3.5.2) and timing the load operation latency corresponding to combination pairs.

Our experiments demonstrate a peak transmission rate of around 1.1 Mbps, which is significantly higher than the 700 Kbps observed when using just one combination pair of (location, coherence state) for encoding binary data for transmission. Figure 4.24 shows spy's reception of symbols through timed load operations along with a magnified view of the first 9 symbols or 18 bits ($\underbrace{10}\,\underbrace{01}\,\underbrace{01}\,\underbrace{00}\,\underbrace{01}\,\underbrace{10}\,\underbrace{01}\,\underbrace{10}\,\underbrace{11}$), in which all four distinct symbols are observed. We note that more sophisticated symbol encoding mechanisms may achieve even higher transmission rates, and our main goal here is simply to demonstrate alternative ways that an adversary can exploit in order to achieve higher bandwidths.

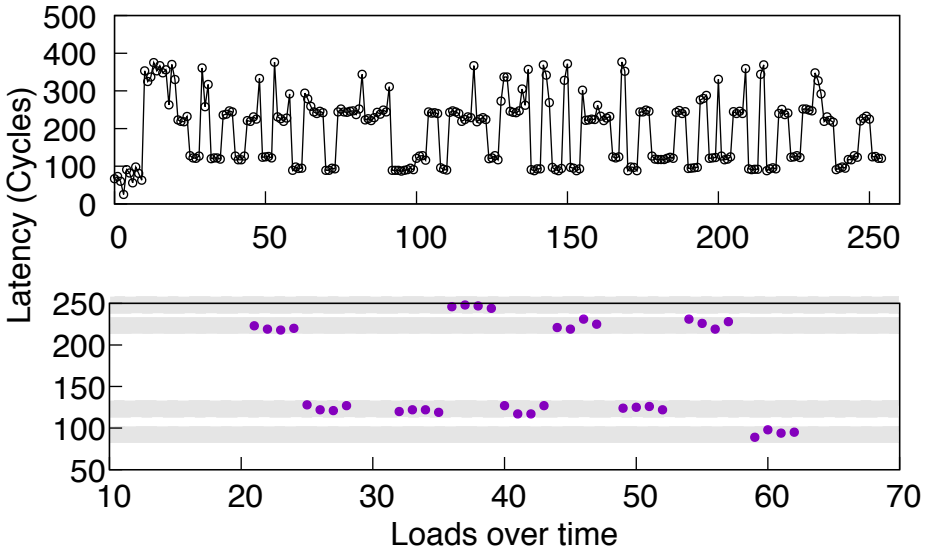



Figure 4.24: Multi-bit symbol transmission using 4 combination pairs to encode 2-bit symbols. Magnified view of first 18 bits reception is shown, that captures all 4 possible symbol values.

| **Coherence Protocol and Cache Inclusiveness** | *Exclusive* Cache Block | *Shared* Cache Block |
|---|---|---|
| Snoopy, Inclusive | Requestor→Owner cache → Requestor | Requestor→LLC →Requestor |
| Snoopy, Non-inclusive | Requester→Owner cache →Requestor | Requestor→MemCtrl →Requestor |
| Directory, Inclusive | Requestor→Directory →Owner cache→Requestor | Requestor→LLC →Requestor |
| Directory, Non-inclusive | Requestor→Directory →Owner cache→Requestor | Requestor→MemCtrl →Requestor |

Table 4.3: Sequence of coherence controllers that interact in order to service the cache blocks in $E$ and $S$ state under different classes of cache coherence protocols. 'LLC' and 'MemCtrl' denote Last Level Cache and Memory Controller respectively.

### 4.3.7 Vulnerability Analysis on Variants of Coherence Protocols

The processors used in our evaluation deploy a variant of directory-based coherence protocol (with LLC's core-valid-bits) that directs the coherence messages to specific cores in order to service these cache misses. In this section, we systematically study the coherence state vulnerabilities in different variants of coherence protocols.

Two factors play a key role in determining the cache access latency, namely the family of coherence protocols and the inclusiveness property of caches. Different

coherence protocol implementations have varying set of transactions when accessing a cache block in a specific coherence state. When a requestor cache controller issues a cache miss request for a block, coherence messages are sent over the interconnection fabric. The owner (e.g., cache controllers, coherence directory or memory controller), that *owns* the requested block, will respond with the data reply. Generally, for an E state cache block, the cache controller that currently holds the private copy of the date is designated as the owner, and responds with the data reply. Differently, the cache directory (usually, LLC) or the memory controller typically own cache blocks in S state. Table 4.3 illustrates the sequence of coherence controllers associated with servicing data miss requests on E- and S-state cache blocks in the four variants of cache coherence protocols[2].

For snoopy protocols run on inclusive caches, read operations on E-state blocks will involve snooping into the private owner cache, while reads on S-state blocks are satisfied by the lower level LLC that has a clean copy of the cache block and acts as the owner [127]. Although both coherence transactions need two hops, they involve different paths. As a result, the cache access latencies for S and E states can be distinguished by adversaries that are monitoring for such information. Similarly, for directory-based protocols run on inclusive caches, a read on E state cache blocks requires a coherence request first sent to the directory module (that is typically maintained in the LLC on many modern processors). The directory then forwards the request to the owner cache, which will subsequently respond with the data. On the other hand, reads on S state cache blocks are replied by the LLC. These two coherence transactions differ in the hops traversed, which results in the distinct latency bands as demonstrated in Table 4.1.

Additionally, the cache inclusion policy influences read operations to S-state cache blocks. Specifically, for inclusive caches, the LLC always owns the S-state blocks and will respond to cache controller directly with the data. In non-inclusive caches, the memory controller is set to own the cache block and accordingly requests to S-state

[2]When the LLC holds a copy of the cache block, the coherence transactions on non-inclusive caches are similar to that of exclusive caches. Therefore, the coherence transactions we listed for non-inclusive caches in Table 4.3 may be applied to a strictly exclusive cache hierarchy as well.

blocks will be serviced by the main memory since the LLC may not potentially have a copy. Such design decisions are made to avoid multiple data transfers from the various sharers.

In summary, we note that all four variants of cache coherence protocols can be vulnerable to exploits due to differences in their timing profile (as discussed in Section 4.3.4). Table 4.3 elaborates the coherence transactions for accessing Exclusive and Shared cache blocks under the four variants of cache coherence protocols.

### 4.3.8 Securing Cache Coherence Protocols

As we know, E-state aims to reduce the coherence transactions and the corresponding latency for writes that immediately follow the read operation to that memory block. Most existing cache coherence protocols allow cache blocks to transition from E to M state without initiating coherence transactions (silent upgrade). Due to this optimization, the cache directory and memory controller will not own these E-state blocks and assume that their data copies are stale.

As discussed in Section 4.3.7, across all the four variants of cache coherence implementations, private caches claim ownership of E-state blocks. These design considerations result in latency differences corresponding to the read-only coherence states, namely E and S, and potentially enable construction of timing channels using read-only states.

#### *4.3.8.1 Modifying E→M Transition*

To remove the read latency differences between E and S states, a potential solution is to service the read requests to cache blocks in read-only states (E and S) uniformly by the directory (or memory controller). This means that all E→M upgrade requests from the cores should be forwarded onto the directory or memory controller every time. This makes the LLC to be aware of the precise coherence state for the corresponding data block (i.e., helps distinguish E vs. M). The LLC can then store the correct coherence state for that block. This enables the LLC to have ownership of E-state cache blocks, since they are guaranteed to be clean until being notified by

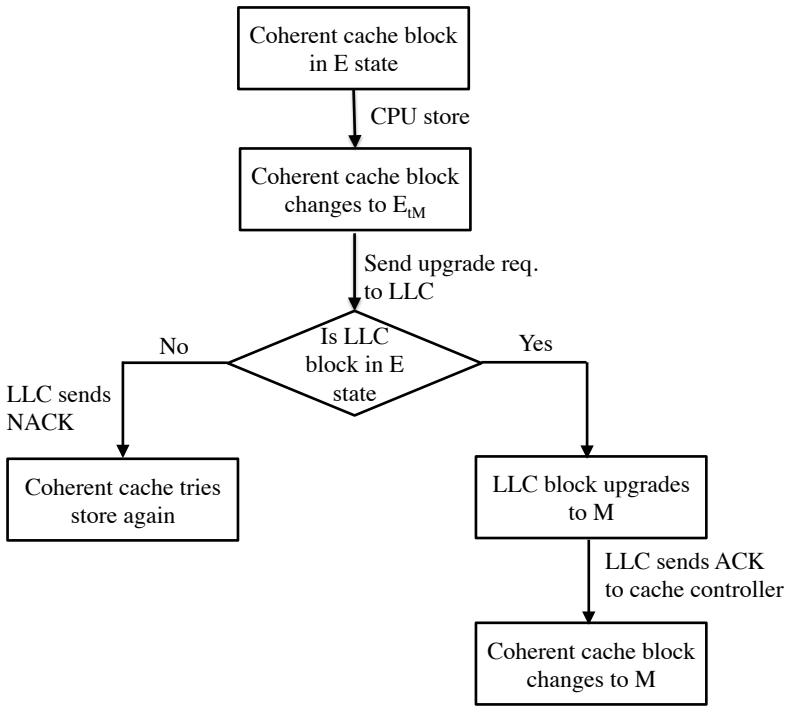



Figure 4.25: Handling E→M transition in directory-based protocols. *Coherent Cache* denotes private caches kept coherent using the coherence protocol hardware.

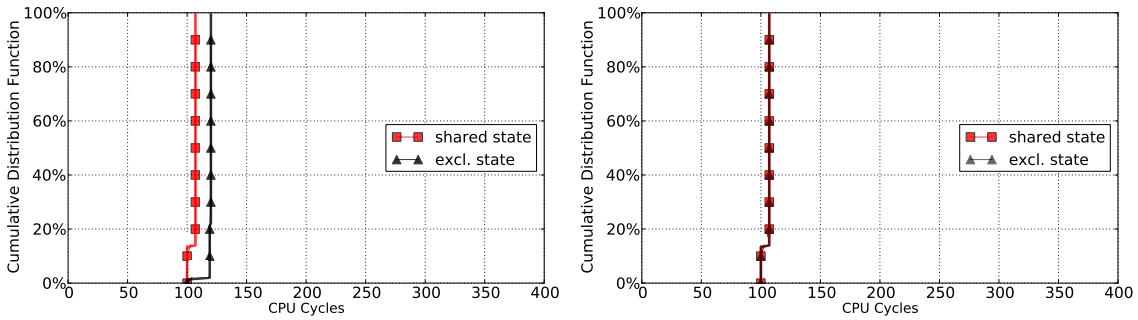
the owner core. Also, the read requests to both E- and S- state blocks can now be serviced by the LLC, and the *read* timing difference between these two states for an external requestor will be zero.

*Modifications needed in Directory-based protocols.* In order to make LLCs own E- and S- state cache blocks, directory-based protocols need an additional transient coherence state. Figure 4.25 shows our solution approach. Upon receiving a write command on E-state blocks in the private coherent caches, the corresponding coherence state transitions to $E_{tM}$. The write upgrade request is then forwarded to the LLC that maintains the directory information corresponding to the cache block. If the current coherence state in the LLC is also E, then the LLC modifies its coherence state to M, approves the E→M, and forwards the acknowledgment to the requestor core. Otherwise, the write request is denied, and the requestor core re-initiates the write operation all over again by sending invalidation requests to other cores.

*Modifications needed in Snoopy-based protocols.* In snoopy protocols, write upgrade requests are typically sent over the system bus when transitioning from S to E state.

| Coherence State | Min Latency | Average Latency | Max Latency |
|---|---|---|---|
| Exclusive & Shared | 119 | **119.4** | 120 |

Table 4.4: Load operation latency (Cycles) for S- and E- state blocks within the socket using the modified directory-based protocol.




(a) Original protocol

(b) Modified protocol

Figure 4.26: Distributions of latencies for accessing E- and S- state cache blocks under original MESI protocol and the modified protocol with changes to E-state cache blocks

Since memory controller monitors all of coherence traffic on the system bus, we note that the following modifications can be made to avoid read latency differences between E and S-state blocks. 1. All read requests to E- and S-state blocks can be replied directly by the memory controller. 2. The upgrade miss requests can be issued for E-state blocks for E→M transitions, instead of S-state blocks for S→E transitions.

#### *4.3.8.2 Latency Profiles with the Modified Coherence Protocol*

To evaluate the effectiveness of our proposed mechanism, we model the modified E→M coherence transition and measure the latency profiles using Gem5, a cycle-accurate full system simulator [23]. We configure Gem5 with eight x86 cores, and use a minimal Linux distribution with kernel version 2.6.32. Each core has a 32 KB private L1 and all cores share a 2MB L2 cache. The microbenchmark (described in Section 4.3.1) is used to profile the cache access latencies. Figure 4.26 shows the CDFs of cache access latencies for the E- and S- state cache blocks under the original coherence protocol and the modified coherence protocol with changes to E-state cache blocks. We can see that under the original MESI-based protocol, the latency profiles for E and S cache blocks accesses are easily distinguishable as the distributions form

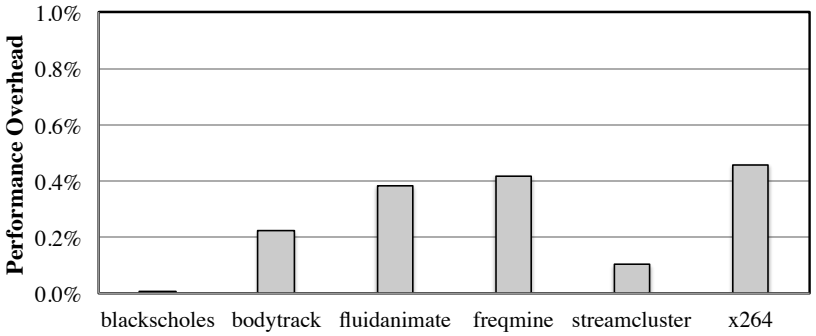



Figure 4.27: Performance overhead for the modified cache coherence protocol in PARSEC benchmarks

two narrow bands that do not overlap (Figure 4.26a). Figure 4.26b demonstrates the same latency profiles in the modified protocol that aims to close the latency gap between the E and S cache block accesses. In fact, the two distribution are exactly the same as the E and S cache block accesses now involve the same coherent transactions. Table 4.4 lists the latency statistics including minimum, average and maximum latencies for accessing E- and S-state blocks. Obviously, an attacker would not be able to build a covert channel by manipulating the two latency values.

#### *4.3.8.3 Implications on Application Performance*

The modifications to the cache coherence protocol require additional messages sent to the directory or memory to upgrade cache blocks from E to M as writes to E-state block will be blocked before the upgrade transaction is completed. This may potentially affect the application performance. To evaluate the performance overhead involved in the modified coherence protocol, we run several multi-threaded PARSEC benchmarks [22] that have various levels of cache coherence activities. Each benchmark is configured to run with four threads. Figure 4.27 shows the performance overheads in terms of execution time for each benchmark's region of interest (ROI). Notably, we observe less than 0.5% overhead for these applications. The performance impact is negligible for the following two reasons: First, the number of stores to E-state cache blocks is only a relatively small portion of all store instructions; Second, the additional transaction for the write to E-state block is lightweight as it

only involves notifications to the last level cache, unlike writes to S-state blocks that typically generate *getM* requests as well as invalidation messages [127]. Specifically, we see that *blackscholes* has very few stores to E-state blocks and our mechanism only incurs less than 0.01% overhead. On the other hand, *fluidanimate* performs a considerable number of writes to E-state blocks (that introduce longer latencies) and as well as many reads to remote E-state blocks (that have reduced delays), and the overall influence of the modified coherence protocol is less than 0.4%.

Moreover, our secure cache coherence is designed to protect the systems where untrusted processes are running on the same machine and are sharing copy-on-write pages (e.g., through KSM). We note that under this context, the latency of E-block upgrade is essentially hidden by the latency of copying the physical page during the first write operation to copy-on-write page. To avoid performance slowdown in regular applications, a simple switch between the performance version (unmodified protocol) and secure version (our modified protocol) can be designed in hardware such that we can achieve trade off between performance and security. When the switch is enabled, E→M transitions will undergo additional steps before actual transition that are described in our Section 4.3.8.1.

## 4.4 Related Work

With the rapid advancements in software confinement mechanism, adversaries are turning to target hardware for information leakage attacks. Lampson et al. were the first to propose the concept of covert channels [86]. In recent years, there are a plethora of studies that demonstrate covert/side channels on various shared hardware resources including caches [61, 95, 122, 161, 176], function units [8, 138], memory bus [153], processor frequency settings [11] and branch predictors [7, 46].

Among various types of processor components, caches are widely exploited for information leakage attacks due to the fact that they are one of the most shared hardware resources. Cache timing channels can be broadly categorized into two classes. Contention-based cache timing channels transfer secrets by creating intentional contention on certain cache sets [95, 115]. The adversaries do not need to have shared

memory between the trojan/victim and spy. The second class of attacks depends on the flushing operations that on shared (typically read-only) memory between the two communicating processes [60, 176]. By controlling whether the accesses to shared cache lines result in cache hits or misses through flushing, the trojan is able to transfer secrets stealthily. We note that most of these attacks rely on modulating the access timing behavior of a single hardware resource that may potentially be addressed through *carefully monitoring the unit*, and if possible, isolating them. In contrast, our work illustrates an attack that leverages the oft-used NUMA architecture and hardware cache coherence mechanism operating on multiple caches with various coherence states. Our study highlights the need for a more careful understanding of such hardware vulnerability in order to devise effective defense strategy against such attacks.

Irazoqui et al. [74] demonstrated a side channel implementation that takes advantage of the cache access timing difference exposed by the high-speed point-to-point interconnect between processors compared to DRAM accesses. This attack manipulates accesses to the remote cache and DRAM. Different from this attack, our NUMA-based covert timing channel involves cache activities on both local (private cache) and remote resources (remote last level cache). This makes it even harder for detection as any effective defense technique would require analysis of inter-socket activities. Prior studies [62, 111] on Intel and AMD processors have shown that the cache access latencies are usually within a stable band of values, which has been observed in our work. Our work demonstrates that cache coherence states can be leveraged to build high bit rate information leakage channels.

Additionally, there are some recent works showing information leakage attacks using specific un-core and off-chip resources. Evtyushkin et al. [45] have shown a covert channel attack that relies on applications using random number generation. DRAMA [121] leveraged DRAM row buffer conflicts to implement timing channels. These attacks also work across CPUs and are powerful in the specific application domain. Moreover, with the wide deployment of GPUs due to its enormous performance in acceleration [68, 69, 96–99, 110, 163], attackers begin to carry out information leak-

age attacks on GPUs. For instance, Jiang et al. [79] demonstrated a side channel to recover AES encryption keys using correlation analysis on GPU platforms. Kadam et al. [80] later generalized the vulnerabilities and proposed a series of coalescing randomization mechanisms that can greatly improve information security on GPUs. We note that cache-based timing channel is potentially even more detrimental due to the fact that caches are commonly shared and mostly used in multi-core systems.

Several existing works have studied the detection and defense techniques for covert/side channel attacks. Demme et al. [43] introduced a metric to quantify the difficulty level to exploit a system for side channels. Venkataramani et al. [29, 30, 132] have proposed techniques that detect contention-based timing channels in functional units and caches. Hunger et al. [70] also studied contention-based cache timing channels and proposed anomaly-based detection. Yan et al. [165] built a record and replay framework that detects covert timing channels by analyzing the difference of caches miss patterns observed from the record and replay runs. Yao et al. [167] proposed techniques to detect Jump-oriented programming based code re-use attacks, which can be potentially applied to mitigate the recently-disclosed Spectre attack that leverages speculation and control flow hijacking [92]. In terms of defense, Wang et al. [139] proposed secure hardware cache designs with partition-locking and random permutation to thwart cache side channels. SHARP [164] redesigned shared cache line replacement policy to avoid inclusion property that is exploited by the spy to decipher the victim's activity. Prefetch-guard [49, 50] leveraged hardware prefetchers to obfuscate the latency measurements for the trojan process in covert timing channels. To defend against memory-based timing channels, Ferraiuolo et al. [52] designed a secure memory scheduling algorithm. Several other works have proposed mechanisms that offer memory safety protection using hardware support for memory access monitoring and tainting [83, 125, 133]. These mechanisms can be effectively utilized to protect systems from covert storage channels. Besides timing, the memory access addresses can also leak sensitive information. To thwart side channels on memory addresses, effective ORAM schemes are proposed that ensure memory access locations are independent of program inputs [135, 136]. Additionally, emerging non-volatile memory

technology is expected to replace DRAMs due to its advantages of increased capacity, data non-volatility as well as low energy consumption [14, 33–35]. Protecting these memory modules from adversaries that exploit data persistence for information leakage is a critical mission [18].

Camouflage [178] reshaped the timing of memory requests and responses to a deterministic distribution in order to eliminate memory access pattern snooping by untrusted parties. Recent works [9, 19] leveraged computation logic in emerging memory technology to cryptographically obfuscate memory addresses and memory bus timing to thwart memory bus attacks. Wassel et al. [140] proposed wave scheduling policy to prevent timing channels in NoC architectures. We note that none of these prior defenses are designed to protect system-wide coherence protocols that entail multiple caches.

CATalyst [94] leveraged the Cache Allocation Technology (CAT) to reserve static cache partitions where secure pages are pinned upon request from the application. Recently, Sprabery et al. [128] proposed a scheduling framework that assigns user-notified sensitive workload on isolated partitions using CAT. Running processes can request to load their sensitive data in the secure pages to avoid cache timing attacks. These solutions require either application-level or user-level cooperation to guide isolation, as a result, they do not defend against covert timing channels attacks where trojan and spy collude to perform information leakage.

CacheBar [179] applied copy-on-access physical page management and controlled the cache-ability of pages in individual containers to defeat side channel attacks. This method may introduce unnecessary performance degradation on benign applications due to increased cache misses. TimeWarp [106] and FuzzyTime [67] mitigated cache timing channel by adding noise to the system clocks, which will reduce the accuracy of spy's latency measurements. However, such mechanisms cannot defeat attacks that use self-clocking or large cache footprint to dilute noises. Additionally, there are some works that detect cache timing channel based on analysis of statistics from performance counters [37, 118]. Chiappetta et al. [37] analyzed LLC miss patterns to track adversaries. However, this can highly impact detection accuracy since the

trojan/spy may intentionally inflate evictions just to evade detection.

## 4.5 Summary

In this chapter, we first investigate and present a new type of hardware vulnerability to covert timing channels exposed by the difference in access timing across multiple levels of the cache hierarchy in Non-Uniform Memory Access (NUMA)-based architectures. We implement a realistic covert timing channel based on this vulnerability, and demonstrate the attack on a dual-socket Intel Xeon server where the trojan modulates the cache access timing by locating itself on a socket different from the spy. We then explore statistical techniques to characterize and quantify the possible presence of covert timing channel activity. We utilize Degree of Sparseness [103] to quantify and analyze the pattern of time intervals for inter-socket cache data transfers when observed during covert channel activity and regular application execution (with no known covert information leakage channels). Our simulation results on Gem5 demonstrates the proposed metric is highly distinctive between covert channel executions and benign application executions. Developing such quantification techniques will be a useful first step in mounting a successful defense against such timing channels.

Additionally, we systematically unravel the vulnerability exposed by cache coherence states to covert timing channels. For the first time, we show how exclusive and shared coherence states may present a significant vulnerability that can be taken advantage by adversaries for covert timing channel construction purposes. In contrast to prior works, we assume a broader adversary model where the trojan and the spy can force create coherence transactions through either explicitly created read-only shared physical pages (e.g., shared library code) or implicitly created shared physical memory pages through an OS feature named Kernel Same Page Merging (KSM). Our study presents novel insights into the behavioral characteristics of a class of covert timing channels that exploit coherence states, their peak bandwidths, and transmission rates in the presence of external noise. Finally, we have proposed a defense mechanism that proposes a secure cache coherence scheme with modest changes to existing coherence protocols, thereby, effectively closing the latency gap between cache accesses to read-

only states, namely E and S. Our evaluation shows that the modified cache coherence protocol is able to annul the timing difference associated with the two cache coherence states with minimal performance overhead for benign applications.

# Chapter 5 Conclusions

As software continues to grow in size and complexity, computer systems are rapidly evolving in order to meet the computation and storage need for end users. While advances in hardware platforms over the past decade bring enormous performance advantages, system administrators and service providers are facing new challenges, namely energy efficiency and information security on multi-core server infrastructures. The tremendous cost corresponding to energy consumption and data breach in server systems has raised significant concerns globally. Hence, enhancing energy efficiency and information security on multi-core server hardware is of great significance for the computer industry.

This dissertation offers in-depth understandings of the aforementioned two qualitative aspects of multi-core server systems. We come up with several techniques that improve the energy efficiency and information security for the next generation multi-core system design. All techniques share the common goal of achieving low costs in order to make them practically applicable in realistic settings.

We first proposed three novel techniques that are aimed at improving energy efficiency for server workloads with QoS constraints by judiciously leveraging server low-power states. The Duel $\tau$ technique makes smart use of system sleep states with two delay-timer configurations to reduce server farm energy consumption. The WASP framework jointly leverages processor and system low-power states, and adjusts its configuration parameters autonomously to achieve optimized energy and latency tradeoff for different workloads. TS-Bat is a power-aware and QoS-aware scheduling framework that integrates spatial and temporal job batching to improve residency of package-level low-power mode to generate considerable processor energy saving. We implemented several prototypes on simulations and in physical testbed with a cluster of servers, and evaluated our proposed techniques with a variety of workloads. Our results show that our proposed solutions are able to significantly improve the energy efficiency of multi-core servers with high flexibility and workload adaptivity.

Additionally, we systematically investigated the information leakage vulnerabilities associated in multi-core server architectures. We discovered the new NUMA-based covert timing channel that exploits the cache access timing difference in non-uniform memory architectures. We implemented the covert timing channel on real system platforms. We explored statistical analysis techniques to characterize and quantify the presence of the covert timing channel activity. We further revealed a critical vulnerability exposed by an oft-used feature in most modern multi-core and multi-socket processors, namely cache coherence protocol states. We showed how adversaries could exploit cache coherence states and construct covert timing channels in order to illegitimately transmit sensitive secrets to untrusted parties by violating the underlying system security policy. To thwart against such attacks, we studied defense mechanisms with slight cache coherence modifications that removes the read latency difference between read-only coherence states, and obstructs the adversaries from taking advantage of these states to implement their timing channels.

Over the past decade, improvements in multi-core server design have made possible a host of hardware optimizations that boost software performance. While performance-optimized designs are clearly beneficial, they can lead to considerably high energy consumption and potentially contribute to severe information security risks. Concerns about energy efficiency and information security can significantly influence the usability of server systems. As architects continue to invest efforts to push the performance envelope, it is equally important to take into account the energy efficiency and information security implications of the hardware design, and consider mechanisms that sustain both aspects with relatively low costs. We consider this dissertation as a step that offers useful insights for designing future low-cost energy efficiency and information security enhancing techniques on multi-core server systems.

Lastly, we summarize several critical future research directions. In terms of server energy efficiency, as computing systems are turning to be heterogeneous with the increasing integration of domain-specific accelerators and performance-varying processors, understanding and improving energy efficiency for workloads on heterogeneous platforms is becoming crucial. Also, with the growing complexity of server config-

urations, it is necessary to design intelligent solutions (e.g., with the help of recent advancements in machine learning techniques) that can perform energy-performance management automatically to eliminate human intervention. For information security, with the plentiful supports of off-the-shelf features in commercial processors, it is worthwhile to explore the potential for leveraging them to building robust information security assurance solutions with low costs. Finally, hardware designers and computer architects should look into secure architecture designs that offer a set of configurable security guarantees for software applications.

## Bibliography


[1] Kcbench. https://linux.die.net/man/1/kcbench.

[2] Libgcrypt project. https://www.gnu.org/software/libgcrypt/.

[3] Intel QuickPath Architecture, 2012. http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf.

[4] The Univ. of Waikato NLANR Projects, 2012. http://www.nlanr.net.

[5] Using Intel VTune Amplifier, 2013. https://goo.gl/E9Fp2m.

[6] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. *ACM SIGARCH Computer Architecture News*, 38(3):338–347, 2010.

[7] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the Symposium on Information, computer and communications security*. ACM, 2007.

[8] O. Aciicmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2007.

[9] S. Aga and S. Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 94–106. ACM, 2017.

[10] V. Aggarwal, M. Xu, T. Lan, and S. Subramaniam. On the optimality of scheduling dependent mapreduce tasks on heterogeneous machines. *arXiv preprint arXiv:1711.09964*, 2017.

[11] M. Alagappan, J. J. Rajendran, M. Doroslovacki, and G. Venkataramani. DFS covert channels on multi-core platforms. In *Proceedings of International Conference on Very Large Scale Integration*. IEEE, 2017.

[12] S. Alamro, M. Xu, T. Lan, and S. Subramaniam. Cred: Cloud right-sizing to meet execution deadlines and data locality. In *IEEE International Conference on Cloud Computing*, pages 686–693. IEEE, 2016.

[13] S. Alamro, M. Xu, T. Lan, and S. Subramaniam. Shed: Optimal dynamic cloning to meet application deadlines in cloud. In *IEEE International Conference on Communications*, pages 1–6. IEEE, 2018.

[14] M. Alshboul, J. Tuck, and Y. Solihin. Lazy persistency: a high-performing and write-efficient software persistency technique. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ACM, 2018.

[15] A. Ambaw, M. Bari, and M. Doroslovački. A case for stacked autoencoder based order recognition of continuous-phase FSK. In *Proceedings of Conference on Information Sciences and Systems*. IEEE, 2017.

[16] A. Ambaw, M. Bari, and M. Doroslovački. A convolutional neural network approach for order recognition of CPFSK signals. In *Proceedings of Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2018.

[17] A. B. Ambaw and M. DoroslovaCki. Feature based order recognition of continuous-phase fsk using principal component analysis. In *Asilomar Conference on Signals, Systems, and Computers*, pages 156–160. IEEE, 2017.

[18] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–276. ACM, 2016.

[19] A. Awad, Y. Wang, D. Shands, and Y. Solihin. Obfusmem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 107–119. ACM, 2017.

[20] M. Bari, N. Lughmani, A. Ambaw, and M. Doroslovački. Comparison of algorithms for raw handwritten digits recognition. In *Proceedings of Annual Asilo-*

*mar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, USA, Oct. 28-31 2018.

[21] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. Cain: silently breaking aslr in the cloud. In *USENIX Workshop on Offensive Technologies*, 2015.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *ACM International Conference on Parallel Architecture and Compilation Techniques*, 2008.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[24] P. Bodik, A. Fox, M. J. Franklin, et al. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of ACM Symposium on Cloud Computing*, 2010.

[25] L. Brown. ACPI in linux. In *Linux Symposium*, 2005. `https://www.kernel.org/doc/ols/2005/ols2005v1-pages-59-76.pdf`.

[26] R. N. Calheiros, R. Ranjan, A. Beloglazov, et al. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[27] J. Cao, Q. Li, Y. Ji, Y. He, and D. Guo. Detection of forwarding-based malicious urls in online social networks. *International Journal of Parallel Programming*, 44(1):163–180, 2016.

[28] G. Casale, N. Mi, L. Cherkasova, and E. Smirni. How to parameterize models with bursty workloads. *SIGMETRICS Performance Evaluation Review*, 36(2), Aug. 2008.

[29] J. Chen and G. Venkataramani. An algorithm for detecting contention-based covert timing channels on shared hardware. In *Proceedings of ACM Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2014.

[30] J. Chen and G. Venkataramani. CC-hunter: Uncovering covert timing channels on shared processor hardware. In *IEEE International Symposium on Microarchitecture*. IEEE, 2014.

[31] J. Chen and G. Venkataramani. A hardware-software cooperative approach for application energy profiling. *IEEE Computer Architecture Letters*, 14(1):5–8, 2015.

[32] J. Chen and G. Venkataramani. enDebug: A Hardware–software Framework for Automated Energy Debugging. *Journal of Parallel and Distributed Computing*, 96:121–133, 2016.

[33] J. Chen, G. Venkataramani, and H. H. Huang. RePRAM: Re-cycling PRAM faulty blocks for extended lifetime. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2012.

[34] J. Chen, G. Venkataramani, and H. H. Huang. Exploring dynamic redundancy to resuscitate faulty pcm blocks. *Journal on Emerging Technologies in Computer Systems*, 10(4):31:1–31:23, June 2014.

[35] J. Chen, Z. Winter, G. Venkataramani, and H. H. Huang. rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–202. IEEE, 2011.

[36] J. Chen, F. Yao, and G. Venkataramani. Watts-inside: A Hardware-software Cooperative Approach for Multicore Power Debugging. In *Proceedings of International Conference on Computer Design*. IEEE, 2013.

[37] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based

side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[38] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[39] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE micro*, 30(2):16–29, 2010.

[40] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 41(1):77–88, 2013.

[41] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[42] Dell, HP, Intel and others. The Intelligent Platform Management Interface (IPMI). `https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html`.

[43] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. *ACM SIGARCH Computer Architecture News*, 40(3):106–117, 2012.

[44] Department of Defense Standard. *Trusted Computer System Evaluation Criteria*. US Department of Defense, 1983.

[45] D. Evtyushkin and D. Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of Conference on Computer and Communications Security*, pages 843–857. ACM, 2016.

[46] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization*, 13(1):10, 2016.

[47] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23, 2007.

[48] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. A noise-resilient detection method against advanced cache timing channel attack. In *Proceedings of Asilomar Conference on Signals, Systems, and Computers*, 2018.

[49] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prefetch-guard: Leveraging hardware prefetchers to defend against cache timing channels (short paper). In *Proceedings of IEEE Symposium on Hardware Oriented Security and Trust*. IEEE, 2018.

[50] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, 2018.

[51] A. Farrell and H. Hoffmann. MEANTIME: Achieving both minimal energy and timeliness with approximate computing. In *USENIX Annual Technical Conference (USENIX ATC 16)*, pages 421–435. USENIX Association, 2016.

[52] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *IEEE International Symposium on High Performance Computer Architecture*, pages 382–393, 2016.

[53] M. Floyd, M. Allen-Ware, K. Rajamani, et al. Introducing the adaptive energy management features of the Power7 chip. In *IEEE Micro*, 2011.

[54] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[55] A. Gandhi and M. Harchol-Balter. How data center size impacts the effectiveness of dynamic power management. In *Proceedings of IEEE Conference on Communication, Control, and Computing*. IEEE, 2011.

[56] A. Gandhi, M. Harchol-Balter, M. Kozuch, et al. Are sleep states effective in data centers? In *Green Computing Conference*. IEEE, 2012.

[57] A. Gandhi, M. Harchol-Balter, R. Raghunathan, et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4):14, 2012.

[58] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[59] X. Gao, Z. Xu, H. Wang, L. Li, and X. Wang. Reduced Cooling Redundancy: A New Security Vulnerability in a Hot Data Center. In *Network and Distributed System Security Symposium*. IEEE, 2018.

[60] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ flush: A fast and stealthy cache attack. *arXiv preprint arXiv:1511.04594*, 2015.

[61] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[62] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of International Symposium on Microarchitecture*, pages 413–422. ACM, 2009.

[63] E. Harrell and L. Langton. *Victims of identity theft, 2014*. US Department of Justice, Office of Justice Programs, Bureau of Justice Statistics, 2015.

[64] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design*. IEEE, 2007.

[65] Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba. Advanced Configuration and Power Interface Specification. `http://www.acpi.info/`.

[66] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *IEEE International Symposium on High Performance Computer Architecture*, pages 271–282. IEEE, 2015.

[67] W.-M. Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[68] Y. Hu, P. Kumar, G. Swope, and H. H. Huang. Trix: Triangle counting at extreme scale. In *IEEE High Performance Extreme Computing Conference*, pages 1–7. IEEE, 2017.

[69] H. H. Huang and H. Liu. Big data machine learning and graph analytics: Current state and future challenges. In *IEEE International Conference on Big Data*, pages 16–17. IEEE, 2014.

[70] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In *International Symposium on High Performance Computer Architecture*. IEEE, 2015.

[71] Intel. Intel Xeon processor E5-1600/E5-2600/E5-4600 product families, 2012. `http://tinyurl.com/d7ma5nf`.

[72] Intel. Intel R 64 and IA-32 Architectures Software Developer Manual. *Volume 3b: System Programming Guide (Part 2)*, pages 14–19, 2013.

[73] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2016.

[74] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In *Proceedings of the 11th Asia Conference on Computer and Communications Security*, pages 353–364. ACM, 2016.

[75] C. Isci, S. McIntosh, J. Kephart, R. Das, J. Hanson, S. Piper, R. Wolford, T. Brey, R. Kantner, A. Ng, et al. Agile, efficient virtualization power management with low-latency server power states. In *ACM SIGARCH Computer Architecture News*, volume 41. ACM, 2013.

[76] Y. Ji, Y. He, X. Jiang, J. Cao, and Q. Li. Combating the evasion mechanisms of social bots. *computers & security*, 58:230–249, 2016.

[77] Y. Ji, Y. He, X. Jiang, and Q. Li. Towards social botnet behavior detecting in the end host. In *IEEE International Conference on Parallel and Distributed Systems*, pages 320–327. IEEE, 2014.

[78] Y. Ji, Q. Li, Y. He, and D. Guo. Botcatch: leveraging signature and behavior for bot detection. *Security and Communication Networks*, 8(6):952–969, 2015.

[79] Z. H. Jiang, Y. Fei, and D. Kaeli. A complete key recovery timing attack on a GPU. In *Proceeding of International Symposium on High Performance Computer Architecture*, pages 394–405. IEEE, 2016.

[80] G. Kadam, D. Zhang, and A. Jog. Rcoal: mitigating gpu timing attack via subwarp-based randomized coalescing techniques. In *IEEE International Symposium on High Performance Computer Architecture*, pages 156–167. IEEE, 2018.

[81] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Proceedings of IEEE International Symposium on Workload Characterization*, 2014.

[82] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proceedings of Intl. Symp. on Microarchitecture*. ACM, 2015.

[83] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic. Comprehensively and efficiently protecting the heap. In *Proceedings of the 12th In-*

*ternational Conference on Architectural Support for Programming Languages and Operating Systems*, pages 207–218. ACM, 2006.

[84] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, 2018.

[85] J. Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[86] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[87] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 9:1–9:14. ACM, 2014.

[88] Y. Li, Y. Chen, T. Lan, and G. Venkataramani. Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *IEEE International Conference on Distributed Computing Systems*, pages 1261–1270. IEEE, 2017.

[89] Y. Li, F. Yao, T. Lan, and G. Venkataramani. Semantics-aware rule recommendation and enforcement for event paths (short paper). In *International Conference on Security and Privacy in Communication Systems*, pages 572–576. Springer, Cham, 2015.

[90] Y. Li, F. Yao, T. Lan, and G. Venkataramani. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security*, 11(12):2748–2762, 2016.

[91] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 36–47. ACM, 2013.

[92] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, 2018.

[93] C. Liu, M. Xu, and S. Subramaniam. A reconfigurable high-performance optical data center architecture. In *IEEE Global Communications Conference*, pages 1–6. IEEE, 2016.

[94] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 406–418. IEEE, 2016.

[95] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[96] H. Liu and H. H. Huang. Graphene: Fine-grained io management for graph computing. In *USENIX Conference on File and Storage Technologies*, pages 285–300. USENIX Association.

[97] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpu servers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[98] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[99] H. Liu, J.-H. Seo, R. Mittal, and H. H. Huang. GPU-accelerated scalable solver for banded linear systems. In *IEEE International Conference on Cluster Computing*, pages 1–8. IEEE, 2013.

[100] Y. Liu, S. C. Draper, and N. S. Kim. Sleepscale: runtime joint speed scaling and sleep states management for power efficient data centers. In *Proceeding of IEEE International Symposium on Computer Architecuture*, 2014.

[101] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of IEEE International Symposium on Computer Architecuture*, 2014.

[102] D. Lo and C. Kozyrakis. Dynamic management of turbomode in modern multi-core chips. In *IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2014.

[103] P. Loganathan, A. W. Khong, and P. A. Naylor. A class of sparseness-controlled algorithms for echo cancellation. *IEEE Transactions on Audio, Speech, and Language Processing*, 17(8):1591–1601, 2009.

[104] B. Lu, S. S. Dayapule, F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. Popcorns: Power optimization using a cooperative network-server approach for data centers (invited paper). In *IEEE International Conference on Computer Communication and Networks*. IEEE, 2018.

[105] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2011.

[106] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[107] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *ACM Sigplan Notices*, volume 44. ACM, 2009.

[108] D. Meisner and T. F. Wenisch. Dreamweaver: architectural support for deep sleep. *ACM SIGPLAN Notices*, 47(4):313–324, 2012.

[109] D. Meisner, J. Wu, and T. F. Wenisch. Bighouse: A simulation infrastructure for data center systems. In *Proceeding of IEEE International Symposium on Performance Analysis of Systems and Software*, 2012.

[110] R. Mittal, J. H. Seo, V. Vedula, Y. J. Choi, H. Liu, H. H. Huang, S. Jain, L. Younes, T. Abraham, and R. T. George. Computational modeling of cardiac hemodynamics: current status and future outlook. *Journal of Computational Physics*, 305:1065–1082, 2016.

[111] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *International Conference on Parallel Processing*, pages 739–748. IEEE, 2015.

[112] D. Mosberger and T. Jin. httperf: a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[113] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *USENIX Symposium on Networked Systems Design and Implementation*, volume 8, pages 323–336, 2008.

[114] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic. LIME: a framework for debugging load imbalance in multi-threaded execution. In *International Conference on Software Engineering*, pages 201–210. IEEE, 2011.

[115] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[116] V. Pallipadi, S. Li, and A. Belay. cpuidle: Do nothing, efficiently. In *Proceedings of the Linux Symposium*, volume 2, pages 119–125. Citeseer, 2007.

[117] T. Pan, T. Zhang, J. Shi, Y. Li, L. Jin, F. Li, J. Yang, B. Zhang, X. Yang, M. Zhang, H. Dai, and B. Liu. *IEEE/ACM Transactions on Networking*, 24(3):1448–1461, 2016.

[118] M. Payer. Hexpads: a platform to detect stealth attacks. In *Proceedings of International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[119] S. Pelley, D. Meisner, T. F. Wenisch, and J. W. VanGilder. Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*, 2009.

[120] D. Perez-Palacin, J. Merseguer, and R. Mirandola. Analysis of bursty workload-aware self-adaptive systems. In *Proceedings of ACM/SPEC International Conference on Performance Engineering*, 2012.

[121] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *Proceedings of USENIX Security Symposium*, 2016.

[122] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of International Conference on Computer and Communications Security*, pages 199–212. ACM, 2009.

[123] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Trends in server energy proportionality. *Computer*, 44(9):69–72, 2011.

[124] K. Sayood. *Introduction to data compression*. Newnes, 2012.

[125] J. Shen, G. Venkataramani, and M. Prvulovic. Tradeoffs in fine-grained heap memory protection. In *Proceedings of ACM Workshop on Architectural and System Support for Improving Software Dependability*. ACM, 2006.

[126] D. C. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling: Myths and facts. 2005.

[127] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[128] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. H. Campbell. A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds.

[129] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering energy proportionality with non energy-proportional systems: Optimizing the ensemble. In *Proceedings of the Conference on Power Aware Computing and Systems*, pages 2–2. USENIX Association, 2008.

[130] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[131] E.-J. van Baaren. Wikibench: A Distributed, Wikipedia based Web Application Benchmark. *Master's thesis, VU University Amsterdam*, 2009.

[132] G. Venkataramani, J. Chen, and M. Doroslovacki. Detecting hardware covert timing channels. *IEEE Micro*, 36(5):17–27, Sept 2016.

[133] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization*, 2009.

[134] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[135] R. Wang, Y. Zhang, and J. Yang. Cooperative path-oram for effective memory bandwidth sharing in server settings. In *IEEE International Symposium on High Performance Computer Architecture*, pages 325–336. IEEE, 2017.

[136] R. Wang, Y. Zhang, and J. Yang. D-ORAM: Path-oram delegation for low execution interference on cloud servers with untrusted memory. In *EEE International Symposium on High Performance Computer Architecture*, pages 416–427. IEEE, 2018.

[137] Y. Wang, Y. Li, and T. Lan. Capitalizing on the promise of ad prefetching in real-world mobile systems. In *IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, pages 162–170. IEEE, 2017.

[138] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of Annual Computer Security Applications Conference*, pages 473–482. IEEE, 2006.

[139] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[140] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of International Symposium on Computer Architecture*, pages 583–594. ACM, 2013.

[141] T. Watanabe. ACPI implementation on freebsd. In *USENIX Annual Technical Conference, FREENIX Track*, 2002.

[142] D. Wong and M. Annavaram. Knightshift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2012.

[143] D. Wong and M. Annavaram. Implications of high energy proportional servers on cluster-wide energy proportionality. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2014.

[144] J. Wu, S. Subramaniam, and H. Hasegawa. Comparison of oxc node architectures for wdm and flex-grid optical networks. In *International Conference on Computer Communication and Networks*, pages 1–8, 2015.

[145] J. Wu, S. Subramaniam, and H. Hasegawa. Optimal nonuniform wavebanding in wdm mesh networks. In *International Conference on Optical Network Design and Modeling*, pages 86–91. IEEE, 2015.

[146] J. Wu, S. Subramaniam, and H. Hasegawa. Optimal nonuniform wavebanding in wdm mesh networks. *Photonic Network Communications*, 31(3):376–385, 2016.

[147] J. Wu, S. Subramaniam, and H. Hasegawa. Dynamic routing and spectrum assignment for multi-fiber elastic optical networks. In *Photonic Networks and Devices*, pages NeTu4F–1. Optical Society of America, 2018.

[148] J. Wu, M. Xu, S. Subramaniam, and H. Hasegawa. Evaluation and Performance Modeling of Two OXC Architectures (Invited Paper). In *IEEE Sarnoff Symposium*, 2016.

[149] J. Wu, M. Xu, S. Subramaniam, and H. Hasegawa. Joint banding-node placement and resource allocation for multi-granular elastic optical networks. In *IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[150] J. Wu, M. Xu, S. Subramaniam, and H. Hasegawa. Routing, fiber, band, and spectrum assignment (RFBSA) for multi-granular elastic optical networks. In *IEEE International Conference on Communications*, pages 1–6, 2017.

[151] J. Wu, M. Xu, S. Subramaniam, and H. Hasegawa. Joint banding-node placement and resource allocation for multigranular elastic optical networks. *Journal of Optical Communications and Networking*, 10(8):C27–C38, 2018.

[152] J. Wu, J. Zhao, and S. Subramaniam. Co-scheduling computational and networking resources in elastic optical networks. In *IEEE International Conference on Communications*, pages 3307–3312. IEEE, 2014.

[153] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security 12*, 2012.

[154] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang. Uksm: swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 325–339. USENIX Association, 2018.

[155] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. Cred: Cloud right-sizing with execution deadlines and data locality. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3389–3400, 2017.

[156] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. Laser: A deep learning approach for speculative execution and replication of deadline-critical jobs in cloud. In *International Conference on Computer Communication and Networks*, pages 1–8. IEEE, 2017.

[157] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. Optimizing speculative execution of deadline-sensitive jobs in cloud. In *ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, pages 17–18. ACM, 2017.

[158] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. Chronos: A unifying optimization framework for speculative execution of deadline-critical mapreduce jobs. In *International Conference on Distributed Computing Systems*. IEEE, 2018.

[159] M. Xu, C. Liu, and S. Subramaniam. Podca: A passive optical data center architecture. In *International Conference on Communications*, pages 1–6. IEEE, 2016.

[160] M. Xu, C. Liu, and S. Subramaniam. Podca: A passive optical data center network architecture. *Journal of Optical Communications and Networking*, 10(4):409–420, 2018.

[161] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd Workshop on Cloud Computing Security Workshop*, pages 29–40. ACM, 2011.

[162] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426. Springer, Cham, 2017.

[163] D. Yan and H. Liu. Parallel graph processing. *Springer Encyclopedia of Big Data Technologies*, pages 1–8, 2018.

[164] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of International Symposium on Computer Architecture*, pages 347–360. ACM, 2017.

[165] M. Yan, Y. Shalabi, and J. Torrellas. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. In *IEEE International Symposium on Microarchitecture*, 2016.

[166] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News*, 41(3):607–618, 2013.

[167] F. Yao, J. Chen, and G. Venkataramani. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Proceedings of International Conference on Computer Design*, pages 467–470. IEEE, 2013.

[168] F. Yao, M. Doroslovački, and G. Venkataramani. Covert timing channels exploiting cache coherence hardware: Characterization and defense. *International Journal of Parallel Programming*, 2018.

[169] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2017.

[170] F. Yao, G. Venkataramani, and M. Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of ACM Great Lakes Symposium on VLSI*, pages 155–160. ACM, 2017.

[171] F. Yao, G. Venkataramani, and M. Doroslovacki. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of Great Lakes Symposium on VLSI*, pages 155–160. ACM, 2017.

[172] F. Yao, J. Wu, S. Subramaniam, and G. Venkataramani. WASP: Workload adaptive energy-latency optimization in server farms using server low-power states. In *IEEE International Conference on Cloud Computing*, 2017.

[173] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. A comparative analysis of data center network architectures. In *IEEE International Conference on Communications*, 2014.

[174] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. A dual delay timer strategy for optimizing server farm energy. In *IEEE International Conference on Cloud Computing Technology and Science*, 2015.

[175] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. Ts-bat: Leveraging temporal-spatial batching for data center energy optimization. In *IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[176] Y. Yarom and K. Falkner. Flush+ reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[177] H. Zheng and A. Louri. Ez-pass: An energy performance-efficient power-gating router architecture for scalable nocs. *IEEE Computer Architecture Letters*, 17(1):88–91, 2018.

[178] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff. Camouflage: Memory traffic shaping to mitigate timing attacks. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 337–348. IEEE, 2017.

[179] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of Conference on Computer and Communications Security*, pages 871–882. ACM, 2016.

[180] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.