UNIVERSITY OF CALIFORNIA
RIVERSIDE

Power Efficient Scheduling for Network Applications on Multicore Architecture

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jilong Kuang

December 2011

Dissertation Committee:

Professor Laxmi Bhuyan, Chairperson
Professor Chinya Ravishankar
Professor Walid Najjar

Acknowledgments

It is my great pleasure to thank those who made this dissertation possible. I would never have been able to finish it without the guidance of my committee members, help from friends, and support from my family and landlord.

I would like to express my deepest gratitude to my advisor, Dr. Laxmi Bhuyan, for his excellent guidance, caring, patience, and providing me with a pleasant atmosphere for doing research. I would also like to thank Dr. Chinya Ravishankar and Dr. Walid Najjar for guiding my dissertation, giving precious advice and participating in my final defense committee.

I would like to thank my dear parents and the entire family. They were always supporting me and encouraging me with their best wishes.

I would like to thank all members from my lab, who were always willing to help, discuss ideas, and give helpful suggestions. It would have been a lonely lab without them. Many thanks to all my collaborators and friends for helping me complete my Ph.D study. Without them, it would have been difficult to write this dissertation.

Finally, I would like to thank my landlord, Gladys Deforest. She was always there in the past five years cheering me up and stood by me through the good times and bad.

ABSTRACT OF THE DISSERTATION

Power Efficient Scheduling for Network Applications on Multicore Architecture

by

Jilong Kuang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2011
Professor Laxmi Bhuyan, Chairperson

Explosive growth of Internet high-traffic applications, such as web browsing, online searching, video streaming, and gaming, requires orders-of-magnitude increase in system throughput. The advent of commodity multicore platforms in the market has opened a new era of computing for network applications due to their superiority in performance, availability and programmability. Along with increased throughput, however, comes significantly increased power consumption. Collectively, millions of servers in the global network consume a great deal of power. And chip manufactures continue to increase both the number of cores and their frequencies, substantially increasing power consumption. With higher power consumption, energy is expected to become more expensive. Higher power consumption also increases core temperature, which exponentially increases the cost of cooling and packaging, as well incurs indirect and life-cycle costs due to reduced system performance, circuit reliability and chip lifetime. Therefore, power efficiency has become and will continue to be a first-order design issue.

In this thesis, we focus on power-efficient scheduling for network applications on multicore architectures. Our goal is to improve the performance of network applications in terms of throughput, latency, power, energy and temperature when deployed on multicore servers. More specifically, we first propose a latency and throughput-aware scheduling scheme based on parallel-pipeline topology. Then, we propose a throughput and latency optimization scheme under given power budget for the parallel-pipeline scheduling topology. We also present a power-optimal scheduling algorithm with regard to traffic variation via the use of per-core Dynamic Voltage and Frequency Scaling (DVFS), power gating and power migration. Further more, we explore temperature related issues by proposing a predictive model-based thermal-aware scheduling scheme. We design, implement, and evaluate our novel schemes on real systems (e.g., Intel Xeon E5335 and AMD Opteron 2350) with benchmark applications ranging from micro level (e.g., CRC checksum calculation and switching table look-up) to IP level (e.g., IP forwarding, routing, and flow classification) to application level (e.g., encryption/decryption and URL-based switching). Through extensive experiments, we observe that our schemes outperform existing approaches substantially.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The explosive growth of network bandwidth and Internet high-traffic applications, such as web browsing, online searching, video streaming, and gaming, require orders-of-magnitude increase in packet processing throughput. The advent of commodity multicore platforms, such as Caviums OCTEON [2], Ciscos AON [3], and IBMs BladeCenter [6], has opened a new era of computing for network applications to take advantage of these low-cost machines due to their superiority in performance, availability and programmability. More and more network packet processing systems have been developed on such platforms ranging from general-purpose processors (e.g., Intel's Xeon [9] and AMD's Opteron [1]) to network processors (e.g., Intel's IXP platform [8]) and programmable logic devices (e.g., NetFPGA [14]). To exploit available parallelism for better throughput, network applications running on multicore platforms usually take one of the following three forms:

1. Spatial parallelism, where multiple concurrent packets are processed in different processors independently. Typical examples can be found in work for TCP (Transmis-

sion Control Protocol) parallelism [20], scalable DPI (Deep Packet Inspection) design [42,73], flow-level packet processing [78] and parallel multimedia transcoding [55].

2. Temporal parallelism (pipelining), where multiple processors are scheduled into a pipeline to overlap periodic executions from different threads. It has been widely adopted in network processors, including Shangri-La [25], auto-partitioning [30], statistical approach [66] and Greedy [95].

3. Hybrid parallelism, which integrates both spatial and temporal parallelism to benefit from the advantages of both sides. It forms a parallel pipeline core topology, where each stage contains multiple parallel cores, such as Random [91] and Bipar [98].

Although the above approaches aim to maximize throughput, none of these systems has considered latency reduction in packet processing because they employ coarse-grained packet-level parallelism. As emerging latency-sensitive applications become popular, such as online gaming, VoIP, fast IP-lookup and real-time DPI, latency plays a more important role than throughput [74]. For example, online gaming requires very low latency, but requires only limited throughput as low as a few kbps. The state of the players changes rapidly and a player could be dead before knowing if the latency is high. Therefore, it is necessary to design a packet processing system that can attain high throughput under given latency constraints.

Traditional task scheduling schemes, such as list-based scheduling [17] and clustering-based scheduling [94], are capable of reducing program latency by exploiting fine-grained task-level parallelism. However, because they do not apply pipelining, they suffer from significant throughput deterioration when executing periodic packet processing tasks. Papers [101] and [61] have presented some research results on reducing protocol latency for high-speed gateways and telecommunication systems based on hybrid parallelism. Developing a packet processing system that considers both latency and throughput for multicore architectures is both interesting and challenging. Thus, in Chapter 3, we present a latency and throughput-aware scheduling scheme based on parallel-pipeline topology.

Along with increased throughput and reduced latency, however, comes increased power consumption for network applications running on multicore architecture [39]. Collectively, millions of servers in the global network consume a great deal of power. And chip manufactures continue to increase both the number of cores and their frequencies, substantially increasing both dynamic and static power consumption. Higher power consumption increases costs, both directly and indirectly. Energy itself is expected to become more expensive, especially if environmental impacts are factored into consumption. Higher power consumption also increases core temperature, which exponentially increases the cost of cooling and packaging [83]. Higher temperatures also increase indirect and life-cycle costs due to reduced system performance, circuit reliability, and chip lifetime. Therefore, power management is a first-order design issue.

At the hardware level, there are two main techniques to reduce power consumption: either by scaling it down or by nearly powering it down altogether, at appropriate times. The first technique, Dynamic Voltage and Frequency Scaling (DVFS), reduces or increases processor voltage/frequency just enough to meet performance requirements. DVFS can be either chip-wide, where the entire chip is scaled as one unit (e.g., IBM's TPMD [36], Intel's Foxton technology [67]), or per-core, where individual cores on the chip can be scaled at different rates (e.g., AMD's Opteron processor [1], IBM's POWER7 systems [7]). Per-core DVFS adjusts the frequency of each core individually at runtime with on-chip voltage regulators, achieving nanosecond-scale voltage switching and per-core voltage control [50]. Although it is more complex to implement, per-core DVFS achieves greater power savings with its fine-grain control over individual cores.

The second hardware-level technique, called power gating, minimizes leakage current when a core is inactive by powering it down almost completely. Power gating has been introduced only recently by major chip manufacturers (e.g., Intel's Nehalem [57]). A power-gated core can be considered inactive with near-zero power consumption while other cores continue to work undisturbed. The transition time to wake up a power-gated core can be in the order of microseconds with integrated power switch [37], which is suitable for practical use with negligible performance cost.

As we propose the parallel-pipeline scheduling on task-level, we realize that there has been no existing work considering the power budget issues for it. Previous power-aware algorithms either have not considered latency [51] [64], or have not explored the parallel-pipeline topology for task scheduling [40] [44] [70]. Since power gating can not be directly applied to task scheduling [64], we resort to DVFS to integrate power-awareness into parallel-pipeline scheduling. In Chapter 4, we propose a throughput and latency optimization scheme under given power budget for the parallel-pipeline scheduling based on per-core DVFS.

In order to determine when to reduce power to the core, various application run-time characteristics must be exploited, such as program phase analysis [46,47], degree of parallelism [60], and time slack detection [54, 70, 100]. We see great potential for power-saving opportunity, in an additional application run-time characteristic: *network traffic variation.* Computing power needs fluctuate dramatically with the large fluctuations in network traffic. For example, Figure 1.1 shows real-time network traffic in a typical day monitored by Equinix data center [4] at San Jose, CA. Different colors represent the breakdown of different packet types. The traffic rate varied from 320K packets/s to 720K packets/s at different times of the day implying that the power consumption could be greatly reduced when traffic is low.

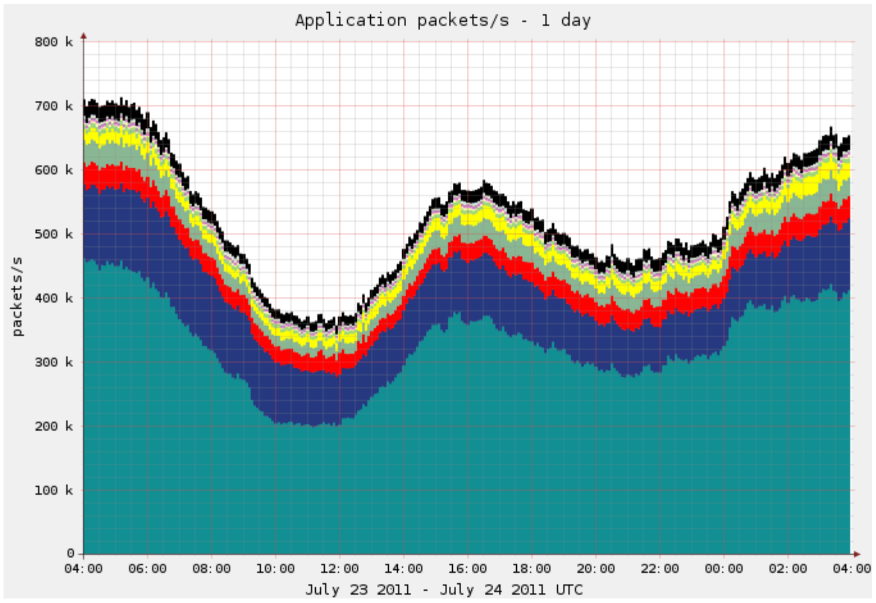

Figure 1.1: Illustration of traffic variation in 24 hours.

Several existing studies do consider traffic variation, but they are limited in the following two ways:

1. **Dynamic Power Only**. They assume that dynamic power dominates total power consumption, and that static power can be ignored [51, 65, 81]. However, static power has increased dramatically with increases in device speed and chip density. According to a projection by the International Technology Roadmap for Semiconductors, leakage power increases its dominance of total power consumption as semiconductors progress toward 32nm [10]. Therefore, static power can no longer be ignored in power models for multicore servers.

2. **Single Dimensional**. Traffic-aware studies focus either on single-core platforms and chip-wide DVFS [81], or adopt power gating only [51, 65]. These approaches cannot be applied to multicore systems that support both per-core DVFS and power gating. Much more fine level optimization is possible.

Using a combination of per-core DVFS and power gating can potentially minimize power consumption when network traffic is low. With this approach, cores perform different amounts of work because all cores do not run all the time, or each core may run at a different frequency. Some cores may then be stressed more than others, and overworked cores will generate excess heat, increasing static power consumption exponentially with temperature [85]. It is therefore advisable to migrate active cores periodically to maintain lower peak core temperature and less static power consumption. A software approach called *power migration* can be used to achieve thermal load balancing across the cores.

Locations of more- and less-active cores can be dynamically changed according to some policy while keeping the same system operating level. Given the same amount of heat generation depending on the number of active cores and core frequency, power migration can redistribute the generated heat, reduce peak core temperature and improve thermal uniformity [27]. Accordingly, we present a power-optimal scheduling algorithm with regard to traffic variation via the use of per-core DVFS, power gating and power migration in Chapter 5.

Besides power and energy, temperature and thermal constraints also pose a significant challenge to future system design [43]. It is known that temperature gradients and thermal hotspots adversely affect not only system performance and leakage power, but also circuit reliability and chip lifetime. In addition, cooling and packaging cost for heat dissipation increases exponentially with power and peak temperature [83].

There are two different approaches to thermal management design. The **reactive approach** refers to applying various thermal techniques (e.g., Stop&Go, hardware toggling or throttling, dynamic frequency and voltage scaling (DVFS), power/clock-gating and task migration [21, 26, 35, 43, 60, 71, 76, 83, 88]) to solve thermal issues based on prior information. However, this approach can only passively respond to thermal emergencies, which causes serious problems including long response time, performance degradation and thermal constraint violation. The **proactive approach** overcomes such problems by using thermal models (e.g., [18,19,48,56,86,89,92,93,96,97,99]) or other means (e.g., [27,58,72]) to estimate system temperature "on-the-fly" and dynamically apply appropriate thermal management techniques.

Traditionally, schedulers for network applications only care about metrics such as throughput, load balancing and real time constraint. It is up to stand-alone thermal management techniques to address thermal issues, most likely following a reactive approach. In this paper, we choose a proactive approach for network applications by integrating thermal awareness into the scheduler design. Our motivation comes from the following two observations: 1) Decoupling scheduling and thermal management as in traditional approaches (reactive) will inevitably cause performance deterioration and thermal constraint violations. 2) Network applications feature in periodic packet processing that allows the scheduler to distribute incoming tasks on different cores in a thermal-aware fashion. Unlike task migration which incurs expensive data copy and communication cost, packet migration has negligible overhead.

There are three challenges to be addressed to design such a thermal-aware scheduler. First, how to dynamically obtain core temperature in a fast and accurate manner? It is known that simulation (e.g., HotSpot [85]) is too slow and impractical for online use. Directly reading on-chip thermal sensors suffers from coarse granularity issues (e.g., AMD's processor does not support per-core temperature reading and Intel's Coretemp driver only reads one temperature per second). Our solution is to build a verifiable thermal model to derive core temperature. Second, how can one build a predictive thermal model for network applications that feature periodic tasks? The existing predictive thermal models [96, 97] are limited to a single task, whose temperature will simply rise to saturation point and then stabilize. Building a time-based predictive thermal model for periodic tasks that can characterize both temperature rise and fall is novel. Third, how to define a good thermal behavior? Some studies set a temperature threshold [60, 76, 89, 97], whereas others focus on thermal balancing and minimizing peak core temperature [26, 27, 71]. We factor both aspects in our design, since otherwise the former suffers from thermal imbalance and the latter degrades the throughput. Thus, in chapter 6, we propose a predictive model-based thermal-aware scheduling for periodic packet processing on multicore architecture.

We design, implement and evaluate all our novel schemes on real systems (e.g., Intel Xeon E5335 and AMD Opteron 2350) with benchmark applications ranging from micro level (e.g., CRC checksum calculation and switching table look-up) to IP level (e.g., IP forwarding, routing, and flow classification) to application level (e.g., encryption/decryption and URL-based switching). We believe that real experimental results are more convincing that simulation results only. Through extensive experiments, we observe that our schemes outperform existing approaches substantially.

The rest of this thesis is organized as follows. Chapter 2 presents related work and motivation for our work. In Chapter 3, we introduce latency and throughput-aware scheduling. In Chapter 4, we present optimization of both throughput and latency under a given power budget. Chapter 5 focuses on traffic-aware power optimization for network applications. We propose a predictive model-based thermal-aware scheduling scheme for periodic packet processing in Chapter 6. Finally, Chapter 7 concludes this thesis.

Chapter 2

Related Work

2.1 Parallelism in Multicore Architecture

Generally speaking, there are three forms of parallelism for network applications running on multicore architecture. The first type is spatial parallelism, which means each core or processor independently executes packets/streams so that multiple concurrent packets/streams can be processed in parallel. Spatial parallelism can be found in many existing works, such as [20, 42, 55, 73, 78]. More specifically, [20] presents a multiprocessor implementation for parallel protocol processing. A processor-per-message paradigm is used to partition the work across processors. It means that any processor can process the whole protocol stack for one message. [42] proposes a highly scalable multi-threaded DPI system (L7-filter) for multicore servers. It explores potential connection level parallelism in pattern matching and develops an affinity-based scheduler to enhance the scalability of multithreading. Similarly to [42], [73] proposes and implements two new DPI packet scheduling algorithms. One is designed to maximize work balance and the other cache affinity. Their observation confirms that scheduling packets for cache affinity is more important than balancing the workload. In addition, [78] presents a design of high-performance flow-level packet processing system based on multicore network processors, which includes a high performance flow classification algorithm, an efficient flow state management scheme, and two hardware-optimized packet ordering strategies. In [55], an adaptive hash scheduler is proposed for stream processing on multicore servers. In particular, it studies video transcoding application and shows that the scheduler can achieve stream locality and load balancing at both the stream and packet level.

The second type is temporal parallelism, which means multiple cores are organized into a pipeline to overlap periodic executions from different threads. It has been widely adopted in network processors, including Shangri-La [25], auto-partitioning [30], statistical approach [66] and Greedy [95]. First, [25] describes the Shangri-La compiler, which applies a throughput-driven heuristic algorithm to merge tasks written in a C-like high-level language into stages from bottom up. Hot code paths identified by profiling are mapped across processing elements to maximize processor utilization. Second, [30] proposes an auto-partitioning C compiler to automatically partition a sequential packet processing application into coordinated pipelined parallel subtasks. Their transformation technique ensures that packet processing tasks are balanced among pipeline stages and that data transmission between pipeline stages is minimized. Third, in [95], they partition network applications into different stages with the consideration of limited instruction memory of the processing elements (PEs). They greedily pack the tasks in sequential order until the code size exceeds the instruction memory size. In addition, they also develop a theoretical approach to determine an optimal topology of the PEs via multiple pipelines to exploit the task/packet level parallelism. Last, in [66], they rely on the inherent modular nature of the network applications and intelligently distributes modules among different execution cores. They selectively replicate modules to parallelize execution of tasks having longer processing time based on statistical analysis of probability distribution of the execution times of different modules.

The third type is hybrid parallelism, which integrates both spatial and temporal parallelism to benefit from the advantages of both. It forms a parallel pipeline core topology, where each stage contains multiple parallel cores, such as Bipar [98] and Random [91]. [98] introduces a multilevel balancing and refining algorithm for NP program mapping. They use a divide-and-conquer approach to recursively bipartition the task graph into disjoint subdomains until the code of the tasks can be fit into the instruction memory of PEs, which guarantees a minimum number of pipeline stages to keep communication cost low. Then their algorithm iteratively refines the solution by migrating tasks from the bottleneck stage to other stages in a hybrid pipeline topology. In [91], they present a methodology to explore the design space for the most suitable system topology (from pipelined to multi-processor solutions) through performance modeling, which determines the system performance and considers the effect of on-chip communication as well as off-chip memory accesses. They propose a random mechanism for mapping the workload optimally to an arbitrary topology based on run-time traces. However, this approach naturally suffers from a time-consuming search process and memory explosion.

All these existing studies only work on coarse-grained packet-level parallelism, which does not allow them to reduce packet latency. In contrast, our work focuses on fine-grained task-level parallelism with parallel pipeline core topology, which is capable of achieving low latency as well as attaining high throughput under given latency constraints.

2.2 Power/Energy-aware Scheduling

Much work has been done on reducing power/energy consumption of computer systems. Based on targeted application types and approaches, we can classify these works into four categories: 1) traditional task scheduling, 2) aperiodic applications, 3) periodic real-time/network applications, and 4) traffic-aware network applications.

The first group combines traditional task scheduling with power awareness. [41] is based on a list-scheduling heuristic with dynamic recalculation of priorities. It minimizes energy usage by choosing the best combination of voltages for each task. [80] applies genetic list scheduling algorithms (GLSA) to schedule and map tasks. Besides simply exploiting available slack time, it also considers the PE power profile during a refined voltage section. [49] presents CASPER (Combined Assignment, Scheduling, and PowER-management) for task mapping and scheduling using a genetic algorithm. It employs two power management techniques (PDP-SPM for homogeneous system and PV-DVS for heterogeneous one) in the fitness function of the genetic algorithm. Although these works address task scheduling, they do not deal with parallel-pipeline topology. Thus, we are motivated to propose a solution that can optimize throughput and latency performance under a given power budget for parallel-pipeline scheduling at the task-level.

In the second category, research communities have extensively studied program execution characteristics for aperiodic applications. Some works focus on differentiating program execution phases based on different CPU/memory-bound ratios and apply Dynamic Voltage and Frequency Scaling (DVFS) accordingly, such as [46,47]. This is because reducing processor frequency when the program is in the memory-bound phase will not affect performance. Specifically, [46] analyzes various global power management policies using per-core DVFS to maximize performance for a given power budget. In [47], they propose a runtime phase predictor that works cooperatively with DVFS. [29] regulates concurrency and changes processors/threads configuration as the program executes by hardware event-driven profiling. [63] optimizes a parallel workload by dynamically changing the number of active processors and the voltage/frequency levels. It applies chip-wide DVFS rather than per-core DVFS. [89] proposes a chip-level power control algorithm that is systematically designed based on optimal control theory. Other works in this category consider processor variation in terms of static power consumed and maximum frequency supported, such as [87], where a variation-aware algorithm for application scheduling and power management is proposed. In addition, there are also some works that integrate both DVFS and power gating in their approaches, such as [60], where they optimize throughput using chip-wide DVFS and power gating for given applications with various degree of parallelism under power and thermal constraints. However, these works do not consider periodic packet processing in network applications on multicore servers, nor do they take into account traffic variation, which is an important characteristics for network applications.

In the third category, a wide variety of DVFS algorithms have been proposed for periodic real-time/network applications. By exploiting deadline slack or changing workload demand, they can reduce power consumption without violating time constraints. [54] proposes a novel scheduling algorithm to optimize both throughput and latency given a power budget for network packet processing on multicore architectures. Their algorithm addresses power-aware parallel-pipeline scheduling problem by applying per-core DVFS to optimally adjust frequency on each core. Some works use more robust DVFS algorithms to schedule power based on multiple future task deadlines such as cycle-conserving and look-ahead earliest deadline first DVFS [75] and Feedback Control-based DVFS [100]. In addition, some strategies based on static power management (SPM) on task scheduling have been proposed in real-time community. [40] uses simple static power management (S-SPM), which distributes global static slack proportionally to the length of the schedule. [70] proposes both greedy static power management (G-SPM), where the entire global static slack is allocated to the first task on each processor, and static power management for parallelism (P-SPM), where the degree of parallelism is taken into consideration. [44] introduces the static power management with proportional distribution and parallelism (PDP-SPM) which exploits both global and local slack. This scheme is so far the best among other SPM mechanisms. However, no parallel-pipeline topology is considered. Other work focuses on multimedia applications, such as [45], which develops an integrated algorithm to control both architectural adaptation and DVFS to save energy. In contrast to this category, we mainly exploit 1) parallel-pipeline scheduling at the task-level and 2) traffic variation for potential power savings as opposed to exploring time slacks.

In the fourth category, the research focus lies in network applications with traffic variation, either with or without real-time constraints. First, [51] adapts the number of activated processors based on queuing theory. It achieves minimized total energy consumption while maintaining a bounded delay. [65] proposes a dynamic power gating scheme for network processors, which adjusts the activities of processing engines (PEs) according to varying traffic volume. It reduces energy consumption by turning off unnecessary PEs when traffic is light. Second, [81] proposes a DVFS policy in addition to dynamic power management (DPM) for a single-core multimedia system to achieve larger power savings. The adjustment of voltage/frequency is based on an M/M/1 queuing model with constant average frame delay. However, all these works consider dynamic power only, which results in sub-optimal power savings. On the contrary, our work proposes a new power model which considers both dynamic and static power and applies a combination of per-core DVFS, power gating, and power migration techniques to optimally adjust per-core frequency configuration "on-the-fly".

2.3 Thermal Management Techniques

Much work has been done in thermal-related research area. We classify related work in this section into two groups based on their approaches in thermal management design, namely, reactive approaches and proactive approaches. Below, we introduce past work in each group respectively.

2.3.1 Reactive Approaches

Reactive approaches can be grouped into three classes [24, 31]: 1) dynamical thermal management (DTM), 2) migration techniques, and 3) combination of the two.

In the first class, [21] is the first paper to investigate different DTM techniques. In this work, five response mechanisms are considered, namely, I-cache-toggling, speculation control, decode bandwidth throttling, clock frequency scaling and a combination of clock frequency scaling and voltage scaling. [83] applies control-theoretic techniques to control DTM to avoid thermal emergencies while minimizing performance loss. In [90], they develop a thermal model called Matrix Model to derive core temperature and present a novel slack allocation algorithm using DVFS to minimize peak temperature. [43] proposes to use both frequency scaling and task-to-core allocation schemes to optimally increase the performance of multicore processors under thermal constraints. [60] proposes to optimize throughput by applying both per-core power gating and DVFS to power- and thermal-constrained multicore processors. [35] proposes a global scheduling algorithm for real-time applications to minimize the peak temperature by deriving an ideally preferred speed for each core.

As regard to migration techniques, [76] proposes heat-and-run SMT thread assignment to increase processor-resource utilization by co-scheduling threads that use complementary resources and heat-and-run CMP thread migration to migrate threads away from overheated cores to alternate cores. [26] investigates the trade-offs between temporal and spatial hot spot mitigation schemes and thermal time constants, workload variations and microprocessor power distribution. By leveraging spatial and temporal heat slacks through task migration, their thermal-aware scheduling schemes enable lowering of on-chip unit temperatures. In [69], they propose and study a thread migration method that maximizes performance under a temperature constraint, while minimizing the number of migrations and ensuring fairness between threads. In [71], they design a lightweight thermal balancing policy for multiprocessor stream computing platforms, which bounds on-chip temperature gradients via task migration.

Last, [24] proposes to use combined thermal techniques to solve temperature-related issues. In this paper, they analyze the impact of a wide range of parameters that influence the performance of DTM for multicore designs, and study different thread migration, DVFS, and combined schemes to provide insight for thermal management on multicore architecture.

All these works are reactive approaches. In contrast, our work based on a predictive thermal model for periodic tasks belongs to proactive approaches.

2.3.2 Proactive Approaches

In this category, some schemes do no rely on any thermal models. [27] proposes proactive power migration to reduce spatial and temporal temperature difference by redistributing the heat generating locations. They consider a predefined migration frequency and evaluate two migration techniques: Cyclic Multiplexing and Global Coolest Replace. In [72], they present a convex optimization based method that proactively controls the temperature of the cores by applying DVFS depending on current workload and current maximum temperature on the chip. They claim to minimize the power consumption, satisfy application performance constraints and guarantee that core temperatures are below a user-defined threshold at all times. In [28], they propose a low-cost temperature management strategy to reduce the adverse effects of hot spots and temperature variations. Their technique utilizes online learning to select the best policy for the current workload characteristics among a given set of *expert* policies. [58] investigates the effects of various task scheduling policies on thermal behavior based on the hypothesis that on-chip thermal sensors are able to feed instant temperature to the scheduler in millisecond time granularity. They claim that the proposed MinTemp policy provides significant alleviation on chip temperature with minimum overhead.

For other works that use thermal models, the prediction of temperature can be based on: 1) power consumption [89,92], 2) hardware event counters [19,56,93], 3) workload characteristics [86] and 4) time [96,97].

1) [89] proposes a chip-level power control algorithm that is based on optimal control theory. Their algorithm can precisely control the power of a CMP chip through DVFS to the desired set point while maintaining per-core temperature below a specified threshold. In [92], they present a temperature-aware scheduling algorithm for soft real-time multicore systems. According to the core temperature and thread thermal contribution, their scheme performs thread migration and exchange to avoid thermal saturation and to maintain temperature equilibrium.

2) [19], [56] and [93] present event counters-based approach to estimate core temperature dynamically. Given this thermal model, [19] is able to throttle execution of individual tasks and reduce CPU time slices of "hot" processes. [56] proposes a methodology for thermal management using both software (priority scheduling) and hardware (clock gating)

techniques. They lower the priority of the hot jobs and raise the priority of the cool jobs for every new epoch. In [93], they leverage the natural discrepancies in thermal behavior among different workloads and schedule them to keep the temperature below a given budget.

3) [86] proposes a predictive DTM algorithm that exploits certain properties of multimedia applications. Based on different frame types, they can dynamically determine the highest performing, thermally safe architectural configuration.

4) Both [96] and [97] use a time-based thermal model, which allows them to predict future core temperature. Based on this prediction, [97] can maintain system temperature below a desired level by moving the running application from the possible overheated core to the future coolest core (migration) and reducing processor resources (priority scheduling); whereas [96] proposes a temperature-aware scheduler to balance heat and reduce peak temperature based on applications' thermal behavior groups classified by a K-means clustering method.

Among these works, [92], [96] and [97] are most relevant to our work because [92] is also targeting periodic tasks and both [96] and [97] use a time-based thermal model. However, compared to them, our work differs substantially in the following three aspects: 1) We are the first to build a predictive thermal model for periodic tasks. In contrast, [96] and [97] only focus on single task, whereas [92] does not have a time-based thermal model. 2) We are the only one to integrate online model update to respond to incidental errors in practical deployment. 3) Our thermal-aware scheduler can maximize throughput and achieve thermal balancing under given temperature constraints by combining task migration and Stop&Go techniques in a cost-effective and cache-aware fashion. In contrast, [97] only considers thermal constraints. Neither [96] nor [97] care about throughput. [92] has to rely on frequent task migration to balance heat across all cores. And none of them consider quantitative cost analysis regarding the impact of core/cache topology in multicore architecture.

Chapter 3

Latency and Throughput-Aware Scheduling

3.1 Introduction

In this chapter, we propose LATA, a LAtency and Throughput-Aware packet processing system for multicore architectures. It adopts hybrid parallelism with parallel pipeline core topology in fine-grained task level to achieve low latency and high throughput. We accomplish the above goal through the following three steps. First, we design a list-based pipeline scheduling algorithm from the task graph. Second, we apply a deterministic search-based refinement process to reduce latency and improve throughput through local adjustment. Third, we devise a cache-aware resource mapping scheme to generate a practical mapping onto a real machine.

To the best of our knowledge, LATA is the first of its kind to consider both latency and throughput in packet processing systems. We implement LATA on an Intel machine with two Quad-Core Xeon E5335 processors and conduct extensive experiments to show its better performance over other systems such as Parallel [20], Greedy [95], Random [91] and Bipar [98]. Based on six real packet processing applications chosen from NetBench [68] and PacketBench [79], LATA exhibits an average of 36.5% reduction of latency across all applications without substantially degrading the throughput. It shows a maximum of 62.2% reduction of latency for URL application over Random with comparable throughput performance.

The rest of this chapter is organized as follows. Section 3.2 introduces LATA system design. Section 3.3 presents LATA scheduling, refinement and mapping algorithms. Section 3.4 describes the experiment framework and the performance evaluation is shown in Section 3.5.

3.2 LATA System Design

Figure 3.1 shows LATA's system design flowchart. Given an network application, we first generate its corresponding task graph with both computation and communication information. Then, we proceed in a three-step procedure to schedule and map the task graph according to our novel design. Last, we deploy the program onto a real multicore machine to obtain its performance result.

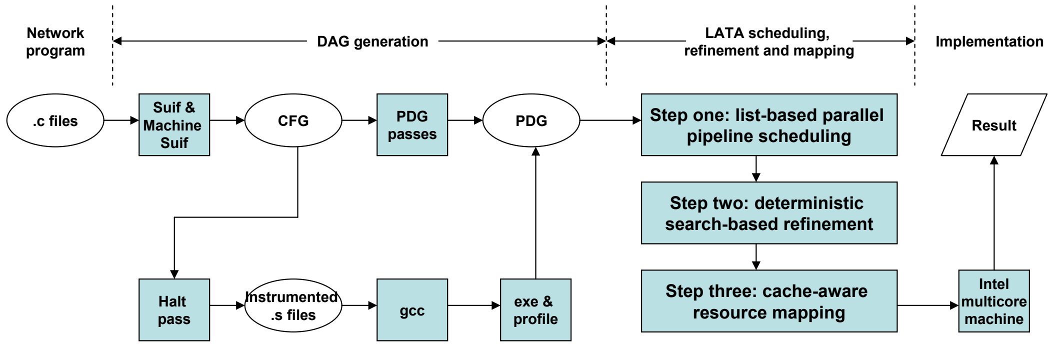
Network program
DAG generation
LATA scheduling, refinement and mapping
Implementation
.c files
Suif & Machine Suif
CFG
PDG passes
PDG
Step one: list-based parallel pipeline scheduling
Step two: deterministic search-based refinement
Step three: cache-aware resource mapping
Result
Halt pass
Instrumented .s files
gcc
exe & profile
Intel multicore machine

Figure 3.1: LATA system design flowchart.

3.2.1 Program Representation

We use program dependence graph (PDG) to represent a program as shown in Figure 3.2(a). PDG can also be called task graph, which is a weighted directed acyclic graph (DAG) defined by tuple $G{=}(V, E, C, T)$, where $V{=}\{n_i,i{=}1{:}v\}$ is the set of nodes and $v{=}|V|$, $E{=}\{e_{i,j}{=}{<}n_i, n_j{>}\}$ is the set of communication edges and $e{=}|E|$. Each node represents a task and each edge represents a communication from one task to the other. C is the set of edge communication times and T is the set of node computation times. $c_{i,j}$ is the communication time on edge $e_{i,j}$ and t_i is the computation time on node n_i [38].

We assume the DAG has a single starting point denoted by head node n_{head} and a single ending point denoted by end node n_{end}. For any DAG, we can always add a head node and an end node with zero computation time and appropriate edges with zero communication time. As an illustration in Figure 3.2(a), each node represents a basic block or a first-level loop in the program, which constitutes the tasks to be scheduled. Although the computation time for each node is easy to obtain (e.g., by inserting timers in the code), the communication time for multicore architectures is hard to measure due to the memory hierarchy. We address this issue in the next section.

3.2.2 Communication Measurement

We can not accurately calculate the communication time between two cores in a multicore architecture like Figure 3.8 unless we know the exact location of the cores. In LATA design, we use the average communication cost based on data cache access time, as given in Equations 3.1 and 3.2. $Comm_{avg}$ means the average communication cost to transfer a unit data set, which can be approximated by system memory latencies (L1, L2 and main memory access time) and program data cache performances (L1 and L2 cache hit rate). $DataSize$ refers to the transferred data set size between two communicating tasks.

$$Comm = Comm_{avg} \times DataSize \tag{3.1}$$

$$\begin{aligned} Comm_{avg} = T_{L1} \times Hit_{L1} + T_{L2} \times (1 - Hit_{L1}) \times Hit_{L2} \\ + T_{MEM} \times (1 - Hit_{L1}) \times (1 - Hit_{L2}) \end{aligned} \tag{3.2}$$

3.2.3 Problem Statement

We define *latency* as the schedule length of a program and *throughput* as the system throughput. The problem statement is: *given the latency constraint L_0, schedule a packet processing program in parallel pipeline core topology so as to maximize the throughput Th.*

The aim is to rearrange the tasks shown in Figure 3.2(a) into the parallel-pipeline task graph shown in Figure 3.2(b), so that the total execution time $T_1 + T_2 + T_3 + T_4$ is minimized while maintaining the throughput as high as possible. As we know, the throughput can be calculated by the inverse of the longest stage time $\frac{1}{T_{max}}$ in pipelining. Thus, we form our objective function in Equation 3.3, where L is the scheduled latency.

$$Maximize \;\; Th = \frac{1}{T_{max}} \;\; (\text{s.t.} \;\; L \leq L_0) \tag{3.3}$$

3.2.4 DAG Generation

As shown in Figure 3.1, LATA's system design consists of DAG generation, LATA scheduling, refinement and mapping, and finally implementation and evaluation. We briefly explain the DAG generation in this section and defer other parts to the following sections. To generate the DAG, we first convert the original C program into the SUIF control flow graph (CFG) [16]. For the ease of dependency analysis, we include all the functions of an application into one single file. After that, we write a Machine SUIF pass [13] to extract the PDG following Ferrante's algorithm based on both control and data flow information [34]. Finally, by using the Halt library in Machine SUIF to instrument source code, we profile the program in the task level for both the computation time and communication time.

To measure the computation time, we feed the program with continuous traffic traces to obtain the average execution time and frequency for each task. To measure the communication time, we first use LMbench [12] to get the L1, L2 and main memory access latencies. Then, we use SUIF/machine SUIF compilers to profile the variable liveness set at the entry of each basic block to measure the transferred data set size. Finally, we measure the program L1 and L2 data cache hit rate by PAPI [15]. After collecting all these information, we calculate the communication time following Equation 3.2.

3.3 LATA Scheduling, Refinement and Mapping

3.3.1 List-based Pipeline Scheduling Algorithm

LATA constructs the parallel pipeline topology based on traditional list scheduling algorithm, which is effective in both performance and complexity [59]. Given a DAG, we define node priority based on the computation top level assuming the same unit computation time for each node. According to [82], the computation top level of n_i is the length of the longest path ending in n_i, excluding t_i and all communication time. The purpose of assuming unit computation time is to find out all task-level parallelism, since nodes in the same level are independent and hence can be scheduled in parallel. The head node n_{head} belongs to level-1. The level of a certain node depends on the highest level among its predecessors. If its highest level predecessor belongs to level i, then that node belongs to level i+1.

We define *ready nodes* as those nodes whose predecessors have already been scheduled. Therefore, a *ready node* can be safely scheduled next. LATA starts off by putting the head node into the list, and then iteratively attaches *ready nodes* in the task graph to the last nodes in the list. This step guarantees that nodes in the list are sorted according to their priorities. After the list is constructed, LATA schedules nodes with the same priority into the same pipeline stage in parallel. Each parallel node takes up one processor and we

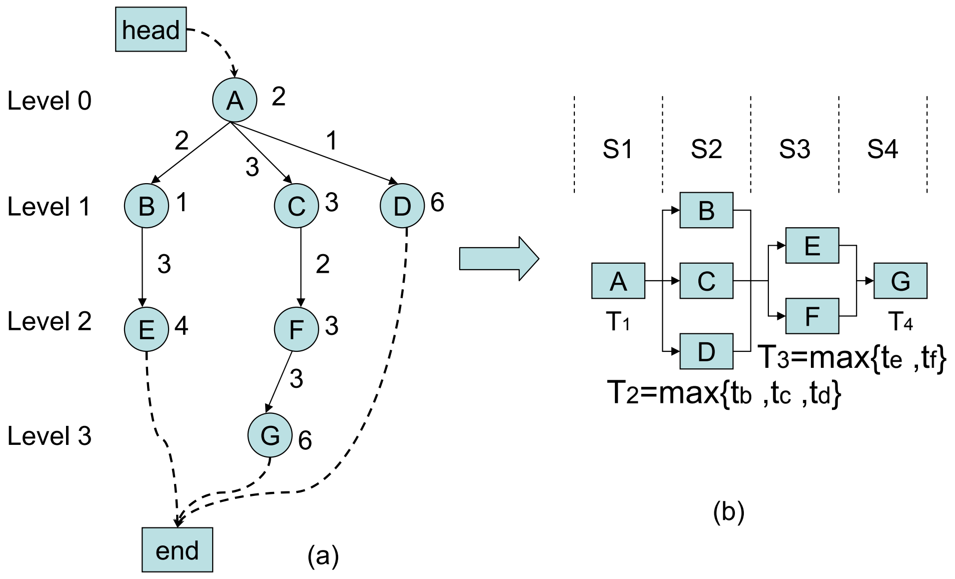

Figure 3.2: Parallel pipeline scheduling from a DAG.

finally obtain a parallel pipeline topology. In this way, latency can be reduced by hiding the computation time of less expensive tasks. Figure 3.2(b) shows the parallel pipeline scheduling from Figure 3.2(a). We ignore head node and end node in the scheduling because they are virtual nodes.

In such a parallel pipeline topology with S stages, we denote a sequential section as a stage with only sequential tasks, such as S_1 and S_4. Similarly, a parallel section refers to a stage with parallel tasks, such as S_2 and S_3. We define *communication critical path* (CCP) as the communication time between two stages, where $CCP_i{=}max\{c_{i,j}\}(n_i{\in}V_i$ and $n_j{\in}V_{i+1})$. The complexity of this step comes from 1) priority assignment, which is $O(V{+}E)$ according to [82], 2) a precedence order among the tasks and 3) CCP calculation. Mergesort with a complexity of $O(VlogV)$ can be used to order the nodes. The complexity associated with calculating CCP is simply $O(S{\cdot}E)$. In conclusion, the complexity is $O(VlogV{+}S{\cdot}E)$.

3.3.2 Search-based Refinement Process

This step focuses on iteratively finding a better scheduling topology by local adjustment of tasks in two phases. The first phase aims at reducing latency and the second phase aims at improving throughput. Although optimizing task scheduling problem is NP-complete in general [33], our heuristic adopts greedy algorithm and works well in practice with low time complexity.

Latency Reduction

Latency can be reduced by reducing either computation time or communication time. Because computation dominates the overall execution time for most packet processing applications running on multicore architectures, we prioritize computation reduction in designing LATA. Hence, LATA first applies *latency hiding* to reduce computation time. Then, *CCP elimination* and *CCP reduction* are used to reduce communication time.

Computation reduction: We define a *critical node* as the node in a pipeline stage which dominates the computation time. Then, *Latency hiding* can be defined as a technique that places a *critical node* from one stage to one of its adjacent stages without violating dependencies, so that its computation time is shadowed by the other *critical node* in the new stage. Backward hiding (BaH) refers to placing a *critical node* into its precedent stage. Forward hiding (FoH) refers to placing a *critical node* into its following stage.

Figure 3.3 shows *latency hiding*, where the node length reflects the computation time. For all the figures shown in this section, bold lines between two stages represent CCPs. Figure 3.3(a) is the same as Figure 3.2(b). In Figure 3.3(b), we place E into its precedent

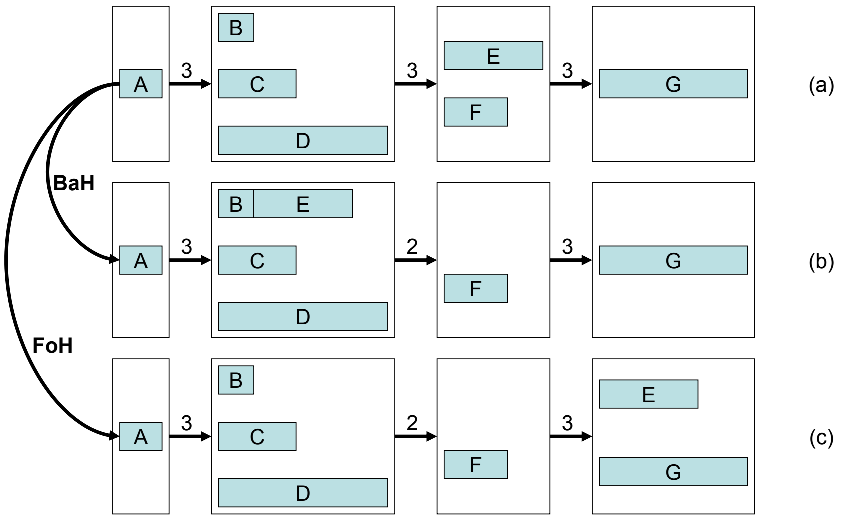

Figure 3.3: *Latency hiding* on node E.

stage with B, where the computation time of E is shadowed by D. In Figure 3.3(c), E is placed into its following stage and E's computation time is shadowed by G.

For each *critical node*, we test whether we can place it into one of its two adjacent stages without violating dependencies or increasing the latency. To break the tie in some cases, we favor the stage with more latency reduction. If both stages happen to reduce the same amount of latency, we favor BaH over FoH. This heuristic increases chances of more potential latency reduction in future iterations. The complexity is $O(S{\cdot}E^2)$ for each iteration, because we have to update the CCP and latency for each attempt, which results in $O(E^2)$. The total complexity is $O(V{\cdot}S{\cdot}E^2)$ after $O(V)$ iterations in this step.

Communication reduction: There are two techniques in communication reduction, namely *CCP elimination* and *CCP reduction*. *CCP elimination* is to eliminate communication time by combining two adjacent stages into one. If every node has only one predecessor in a certain stage, we can attach nodes in that stage to their predecessors in the precedent stage.

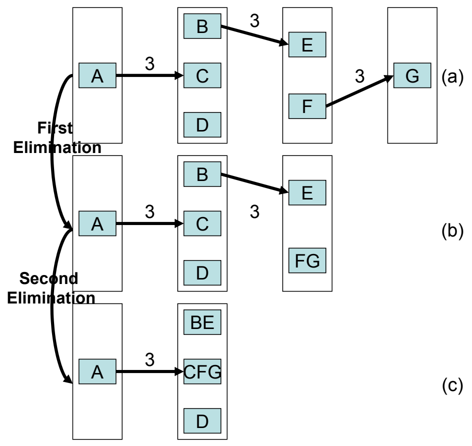

Figure 3.4: *CCP elimination.*

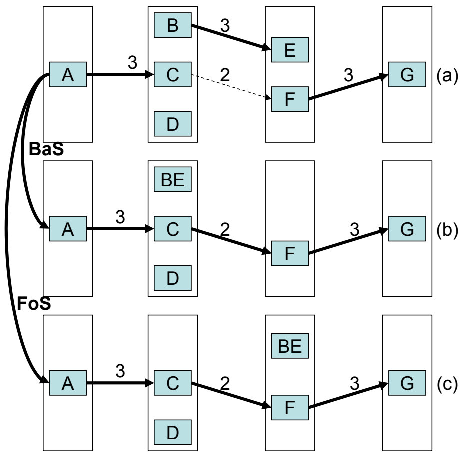

Figure 3.5: *CCP reduction.*

As shown in Figure 3.4, the first elimination combines the last two stages together. G is attached after F, since F is the only predecessor of G in Figure 3.2(a). The second elimination shown in the figure combines the last two stages again. This time, we attach E to B and FG to C. From Figure 3.4(c) we see that two CCPs haven been eliminated from the original pipeline scheduling, which results in the latency reduction by 6.

CCP reduction is to reduce the CCP weight by switching a node associated with the current CCP to one of its adjacent stages. Figure 3.5 shows two reduction techniques BaS and FoS. Backward switch (BaS) refers to switching a node backward to its precedent stage. Forward switch (FoS) refers to switching a node forward to its following stage. For each CCP, we consider the two nodes associated with it. Both BaS and FoS are tested on the task in the two nodes. If any latency reduction can be obtained, we take that action.

To break the tie in some cases, we favor the one with more CCP reduction. In case of equal CCP reduction, we choose BaS rather than FoS due to the same reason as in *latency hiding*.

For example, in Figure 3.5(b), E is switched backward to B, so communication time between B and E is eliminated, and communication time between C and F becomes the new CCP with less weight. In addition, Figure 3.5(c) shows the case where B is switched forward to E. Similarly, we see a decreased CCP between the two stages.

As we decrease the latency, it is possible that T_{max} will increase, which is unfavorable for the throughput according to Equation 3.3. So, for each CCP, whether we apply *CCP reduction* or *CCP elimination* is decided by Q, a beneficial ratio defined by $Q{=}\frac{\Delta L}{\Delta Th}$. We start off by selecting the biggest CCP. Then both techniques are tested to calculate the ratio Q. We take action on the one whose resulting Q is larger, which guarantees minimal throughput sacrifice. This process is iteratively executed until the latency constraint is achieved. The complexity for both techniques is $O(E{\cdot}V)$ in each iteration. Therefore, the total complexity is $O(E^2{\cdot}V)$ after $O(E)$ iterations.

Throughput Improvement

So far, we have reduced the latency by various techniques. In this section, we focus on improving throughput without violating the latency constraint. According to Equation 3.3, we can improve throughput by reducing T_{max} through *decomposition.* During the previous latency reduction process, chances are that many nodes are comprised of several tasks. If a node with T_{max} consists of more than one task, it can be decomposed into two separate nodes to reduce the bottleneck stage time T_{max}.

We define *decomposition* as to decompose one node with multiple tasks into two separate nodes without violating the dependencies. There are four decomposition techniques depending on how the two decomposed nodes are located as shown in Figure 3.6, where thick boxes indicate bottleneck nodes.

- SeD (Sequential Decomposition): Decompose two tasks from one processor into two adjacent tasks in different processors in a sequential section.
- PaD (Parallel Decomposition): Decompose two tasks from one processor into two parallel tasks in different processors in a parallel section.
- BaD (Backward Decomposition): Decompose two tasks from one processor into one task in the current section and the other in the precedent section.
- FoD (Forward Decomposition): Decompose two tasks from one processor into one task in the current section and the other in the following section.

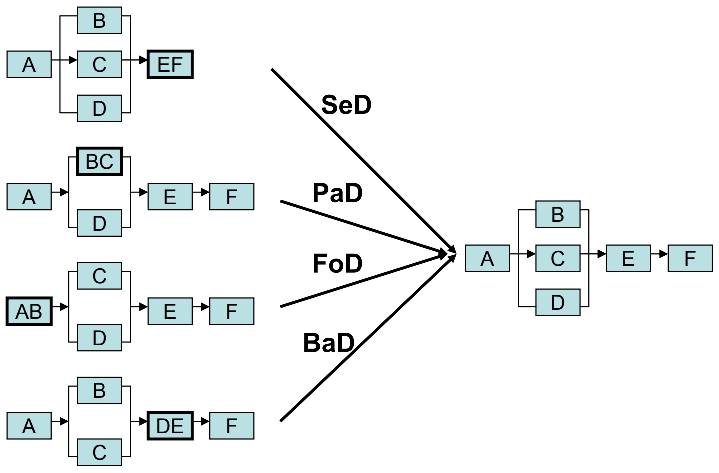

Figure 3.6: Illustration of four decomposition techniques.

We proceed the refinement process by iteratively applying the *decomposition* on current T_{max} node until no more throughput gain can be made. During each iteration, we first locate the node where T_{max} comes from. Then, for each task within that node, we attempt to apply the four decomposition techniques. After recording all possible decomposition points and corresponding techniques where positive results appear, we choose the task where the reduction of T_{max} is maximized as the potential decomposition point. If the latency constraint is not violated, we take that action.

The complexity for decomposition is $O(V^2)$ for each iteration and there are $O(V)$ iterations. Hence, the overall complexity is $O(V^3)$.

3.3.3 Cache-Aware Resource Mapping

Pre-mapping

The first step, pre-mapping, assigns a pre-defined number of virtual processors (8 in LATA) to scheduled nodes, considering both computation time balancing and communication time minimization. First, we check all parallel sections to see if we can combine two independent parallel nodes into one without increasing the latency nor reducing the throughput. After this, if we still end up with more scheduled nodes than real nodes, we iteratively bipartition the pipeline into two parts with the cutting point being the minimal CCP. This guarantees a minimal communication overhead [98]. For each bipartition step, we assign virtual processors in proportion to the workload in each portion. With respect to workload, we refer to the total computation time of all the tasks in that stage. At the same time, we avoid assigning more than one virtual processor to a single task.

This recursive algorithm terminates when 1) there is only one virtual processor left unmapped or 2) there is only one pipeline stage left with extra virtual processors. In the first case, we assign all the remaining tasks into that virtual processor. In the second case, we assign all the remaining virtual processors into that stage, with each virtual processor taking a fair share of workload by round-robin.

Real Mapping

The second step, real mapping, addresses specific task-to-core mapping. Figure 3.7(a) shows the tree structure of the processing units (PUs) on Xeon chip. From bottom up, a group of two cores shares the same last level cache (L2). Two of these groups (4 cores) share the same socket (S1 or S2). Two of these sockets (8 cores) share the same chip. Obviously, the communication cost between cores is asymmetric as illustrated by the different thickness of the curves in Figure 3.7(a). As a result, we can take advantage of the tree hierarchy to implement a cache-aware resource mapping.

First, we extract all the communication edges out of the scheduled topology and sort those edges in decreasing order. Figure 3.7(b) shows a sample scheduling topology after pre-mapping, where all the arrows represent the communication time. The thicker the arrow, the more time-consuming the communication. Second, we start off by picking the most time-consuming edge and then assign the two associated nodes to nodes with minimal communication cost. In our example, A and C are picked first and are assigned to C0 and C2, respectively. Third, we iteratively apply the same greedy algorithm until all the nodes are mapped to real cores as shown in Figure 3.7(c). During each iteration, we just pick the

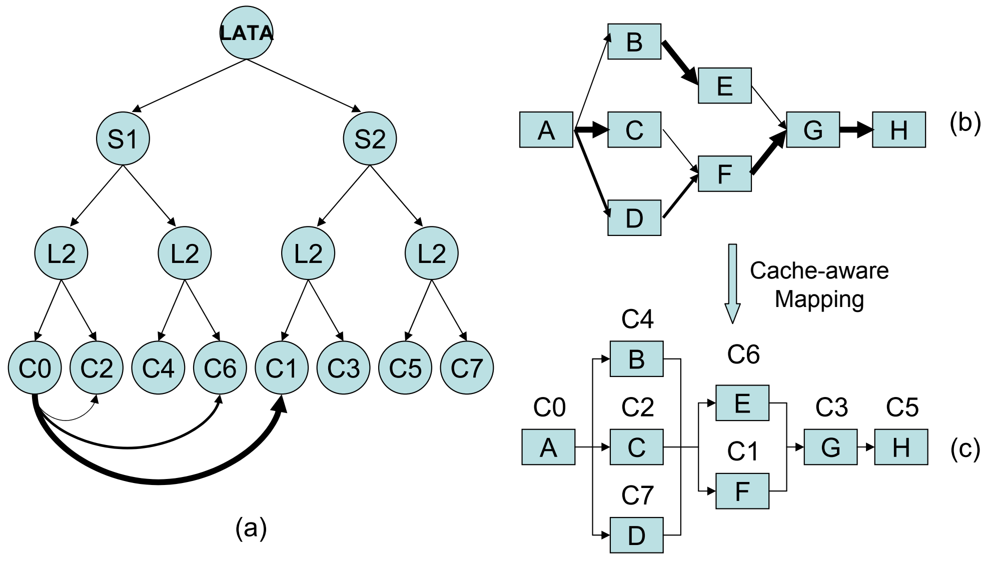

Figure 3.7: Illustration of cache-aware mapping.

thickest edge out of all the remaining edges and assign the unmapped nodes with the cores that incur the least communication cost among all the unmapped cores.

In cases when the real system provides more cores than the original scheduled topology, we simply apply real mapping first and then put extra cores to the bottleneck stage for packet-level parallelism.

The complexity in this section is dominated by pre-mapping process. Due to the bipartitioning, there are $O(logS)$ iterations. For each iteration, the complexity is simply $O(VlogV)$, which is from the sorting algorithm. Thus, the total complexity is $O(VlogV \cdot logS)$ in the algorithm. Considering the first two steps in LATA system design, we conclude that LATA has a total time complexity of $O(V^3+V \cdot E^2 \cdot S)$. Since LATA is designed off-line, this complexity is acceptable for packet processing systems.

3.4 Experiment Framework

We implement and evaluate LATA along with four other systems (Parallel [20], Greedy [95], Random [91] and Bipar [98]) to show its performance advantage in both latency and throughput. Latency is measured by the average execution time of one packet in microseconds (usec) and throughput is measured by million packets per second (mpps).

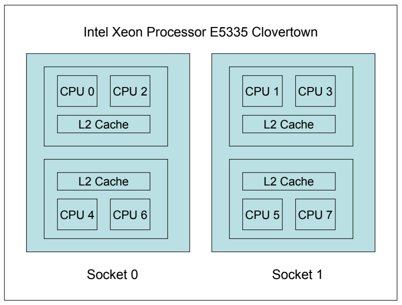

Figure 3.8: Layout of two Quad-Core Intel Xeon E5335 processors.

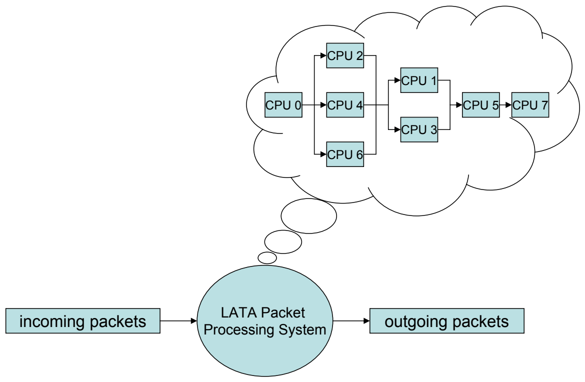

Figure 3.9: LATA with parallel pipeline topology.

We build our LATA packet processing system on a multicore server as shown in Figure 3.8. The target platform, an Intel Clovertown machine, consists of two sockets. Each socket has a Quad-Core Xeon E5335 processor with two cores sharing a 4MB L2 cache. The L1, L2 and main memory access latencies turn out to be 3, 14 and 217 CPU cycles, respectively, as measured by LMbench [12]. Figure 3.9 illustrates the overall system design. LATA assumes a single incoming and outgoing queue. The central part consists of an 8-core machine organized into a parallel pipeline core topology to exploit both spatial and temporal parallelism. The hardware configuration is set by default as an 8-core machine with instruction cache up to $4k$ instructions. The instruction cache size is an important parameter in partitioning programs and is used when we compare LATA with other Network Processor (NP) systems, whose processing engine has limited memory [8]. Our system is running Linux-2.6.18 OS and we use Pthread libraries to synchronize different tasks.

Six applications are chosen from NetBench [68] and PacketBench [79], including five IP-level programs (Flow, IPv4-trie, Route, DRR and IPchains) and one application-level program (URL). Their functionalities and code sizes (the number of instructions) are listed in Table 3.1. It is worth noticing that our selection of applications is based on the following three metrics: 1) code size must be large enough; 2) applications must be representative; and 3) applications must expose relatively more parallelization potential. The packet trace is from NetBench with $10,000$ packets. The routing table used for IPv4-trie is MAE-WEST [79] and the routing table size for DRR, IPchains and Route is set to 128 by default. We scale URL's results by a factor of 0.01 due to figure space limitations. For LATA, we assume the latency constraint is 75% of the sequential execution time for each application.

Table 3.1: Six packet processing applications.

Application	Functionality	Code Size
URL	URL-based switching	1428
Flow	Flow classification	3190
IPv4-trie	IPv4 routing based on trie	4596
Route	IPv4 routing based on radix	6600
DRR	Deficit-round robin scheduling	7633
IPchains	Firewall based on IP source	14735

We classify the four systems into two groups according to the form of parallelism. In the first group, LATA is compared with Parallel system (spatial parallelism), where every processor independently executes different packets in parallel as in [20]. No memory constraint is considered in this group. We also implement a list scheduling algorithm (List) called HLFET (Highest Levels First with Estimated Times) [17] as a reference for the best achievable latency. In the second group, we compare LATA with three NP systems based on pipelining (temporal parallelism) as in Greedy [95] and parallel pipelining (hybrid parallelism) as in Random [91] and Bipar [98]. These systems have limited memory constraints for each processor.

3.5 Performance Evaluation

3.5.1 Comparison with Parallel System

Figures 3.10 and 3.11 show the latency and throughput for six applications by LATA, Parallel and List. We observe that Parallel suffers from high latency due to its sequential execution of tasks. Compared with Parallel, LATA reduces the latency by an average of 34.2%. Particulary, for URL, LATA achieves the maximal latency reduction of 62.2%. In addition, LATA's throughput is close to that of Parallel in spite of the 75% latency constraint. This is because LATA is capable of optimizing its parallel pipeline core topology to produce good throughput. With respect to List, which is designed to produce the lowest latency, LATA actually matches its latency performance in most cases by aggressively exploiting task-level parallelism. Furthermore, LATA outperforms List in throughput by an average of 41.0% and a maximum of 56.7% for Route.

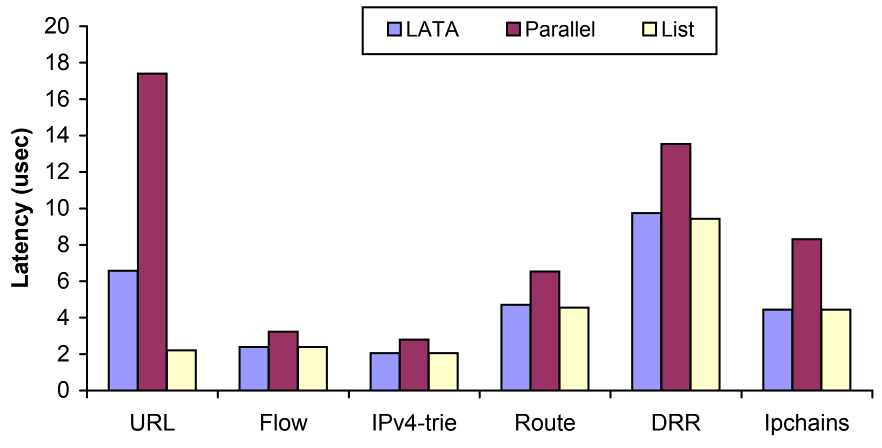

Figure 3.10: Latency of six applications by LATA, Parallel and List.

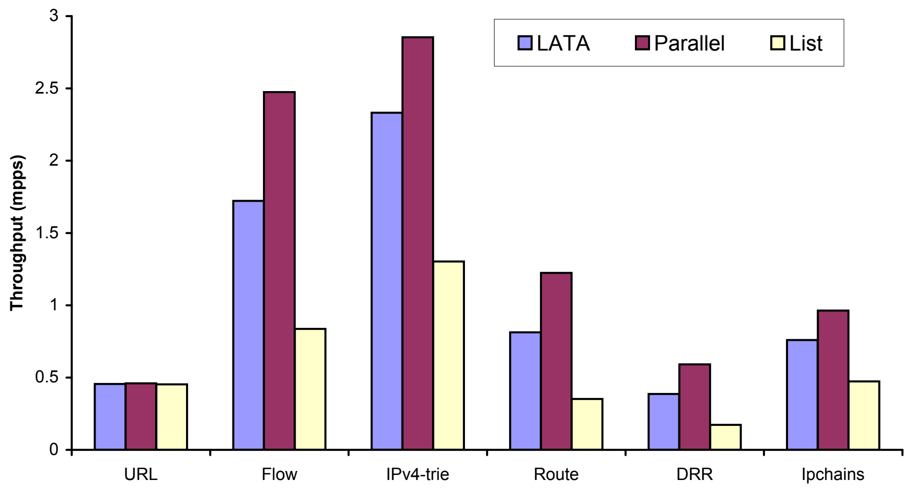

Figure 3.11: Throughput of six applications by LATA, Parallel and List.

3.5.2 Comparison with Three NP Systems

Figures 3.12 and 3.13 exhibit the latency and throughput for the three NP systems. Except LATA, all other systems adopt packet-level parallelism, which suffer from high latency. The slightly lower latency by Bipar and Greedy over Random comes from less communication overhead due to shorter pipeline length. LATA, on the other hand, exposes a substantial latency decrease by an average of 37.3% and maximum of 62.2% for URL compared with Random. Considering the throughput, we observe that LATA catches up with other systems in 5 out of 6 applications in spite of the 75% latency constraint, except the Flow application. However, the average of 30.4% throughput loss in Flow is compensated by 26.0% performance gain in latency reduction. This tradeoff once again proves LATA's uniqueness in satisfying the stringent latency constraint while attaining a comparable throughput.

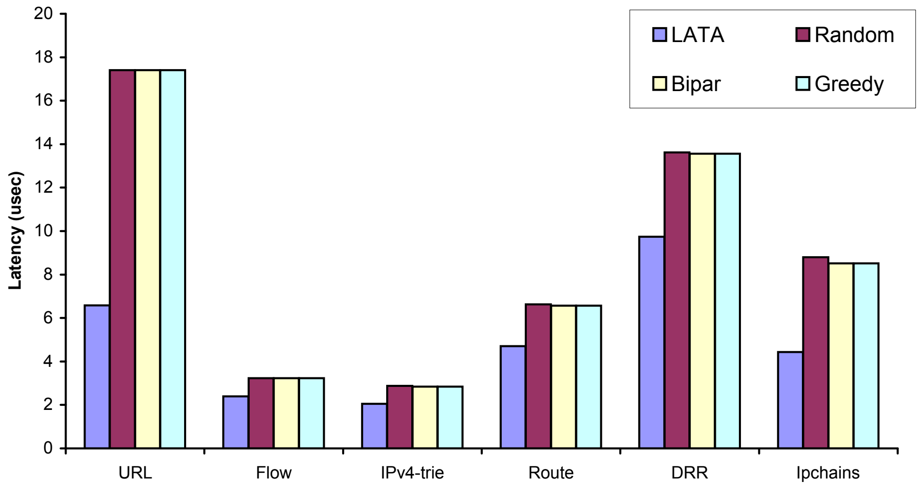

Figure 3.12: Latency of six apps by LATA, Greedy, Random and Bipar.

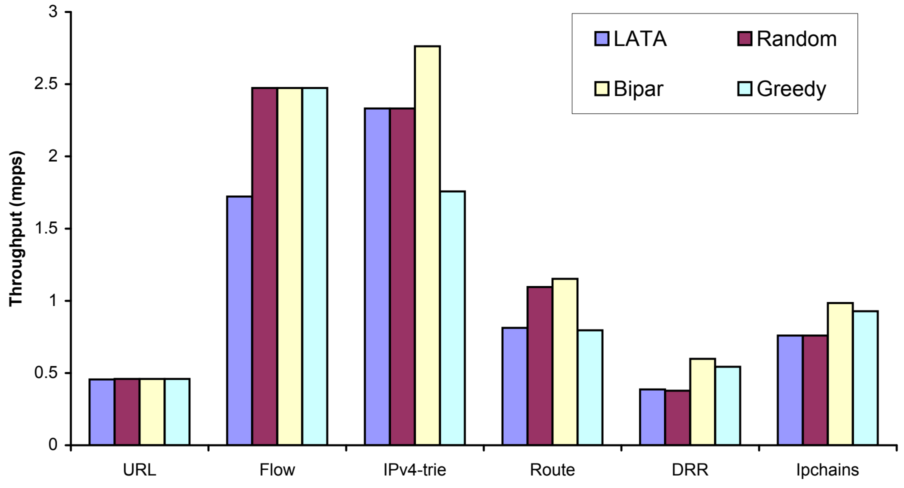

Figure 3.13: Throughput of six apps by LATA, Greedy, Random and Bipar.

3.5.3 Latency Constraint Effect

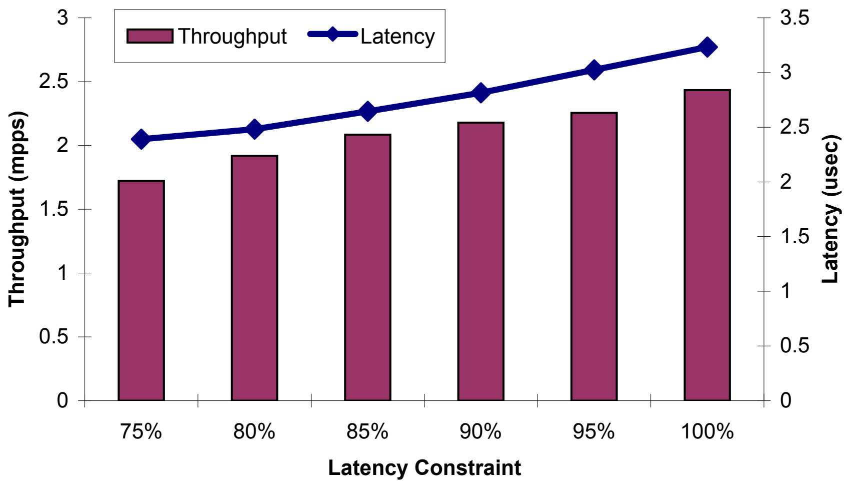

Figure 3.14: Latency and throughput of Flow application by LATA.

In this section, we show how the latency constraint affects the throughput of LATA by alleviating the latency constraint from 75% to 100%. We choose the Flow application as an example because its throughput by LATA is the worst in Figure 3.13. From Figure 3.14, we observe that as the latency constraint becomes less stringent, the throughput improves accordingly. In fact, when there is no constraint (at the point of 100%), LATA produces the same throughput as other systems do. This is because LATA spares nodes from parallel sections to help reduce the bottleneck stage time by applying *decomposition*. Originally, those nodes are used to satisfy latency constraints, causing many tasks from sequential sections to be fed into few nodes, which deteriorates the throughput with large T_{max}. This figure also shows the latency performance at each point with an increasing trend, which follows the changing latency constraint. In a word, not only can LATA achieve low latency without substantial throughput loss when the latency constraint is stringent, but also it can attain high throughput when the latency constraint is light.

3.5.4 Scalability Performance of LATA

We evaluate LATA's scalability by varying the number of cores. Figure 3.15 demonstrates the latency and throughput for Route. As the number of cores increases, we observe a decreasing trend of latency, which reflects the fact that LATA exploits task-level parallelism to reduce program execution time. When the core resource is not plenty (less than 3), no task-level parallelism can be exploited. When the number of cores increases from 3 to 6, task-level parallelism gradually takes effect and an obvious time decrease can be observed. As the core resource continues to increase (more than 6), we notice that the latency has reached the lower bound.

In addition, the increasing bars show that the throughput improves with more cores. From this figure, we can make two interesting observations. First, there is a slight throughput decrease when the number of cores increases from 2 to 3. This seemingly contradictory result can be explained by the fact that LATA prioritizes latency reduction when the latency constraint has not been satisfied. In this case, the extra core is used to reduce latency rather than improve throughput. We can clearly see the latency reduction during that period from the latency curve. Second, while the latency becomes saturated after 6 cores, the throughput continues to improve. This is because that extra cores can reduce bottleneck stage workload, which results in better throughput for the whole system.

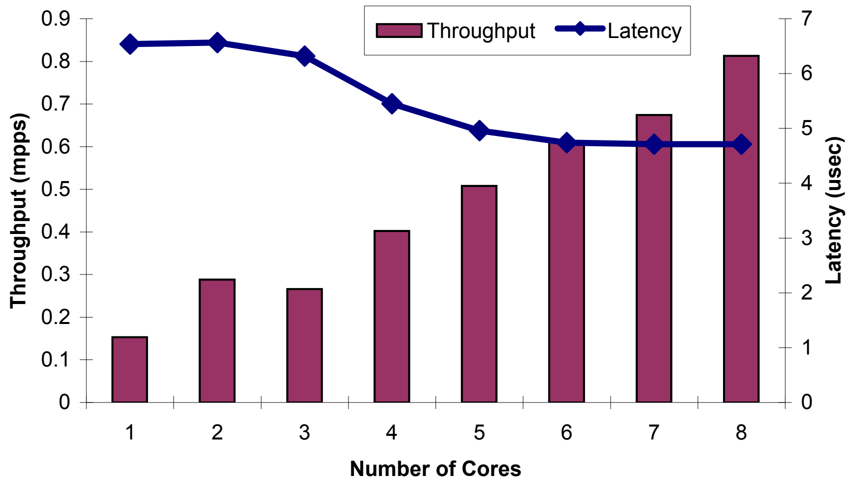

Figure 3.15: Latency and throughput of Route application by LATA.

3.5.5 Instruction Cache Size Performance

Lastly, we analyze the effect of the instruction cache size for IPv4-trie. Figure 3.16 shows the throughput when the cache size varies. As the cache size increases, we observe an increasing trend of throughput. Bipar produces the best throughput in most cases due to its minimal communication cost and balanced workload assignment. Greedy, which performs the worst, suffers from imbalanced task assignment, especially when the cache size is $4k$. LATA and Random sit between Bipar and Greedy. However, LATA has the least cache requirement compared to other systems. When the cache size is as small as $1k$, LATA produces the best throughput. Its throughput slowly grows as the cache size increases from $2k$ to $4k$. After that, LATA's performance catches the best. Since the code size is less than $5k$ for IPv4-trie, all systems produce the same best throughput at $5k$ point. The corresponding latency performance is similar to that of Figure 3.12 and hence, we omit it due to space limitations.

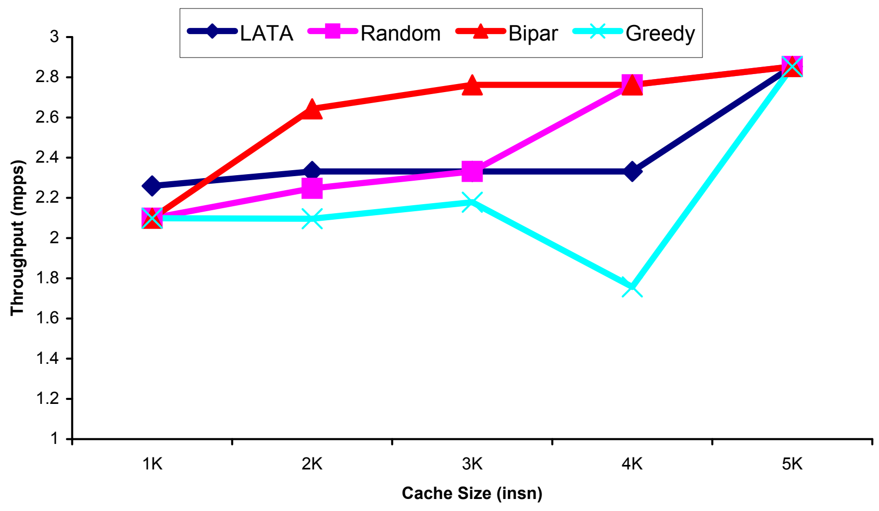

Figure 3.16: Throughput of IPv4-trie by LATA, Greedy, Random and Bipar.

Chapter 4

Optimizing Throughput and Latency Under Power Budget

4.1 Introduction

In the previous chapter, we introduce the novel parallel-pipeline scheduling on task-level for network applications that can attain high throughput under given latency constraints [53]. In this chapter, we address the power budget issue for this scheduling paradigm for network packet processing. We aim at optimizing both throughput and latency under given power budget by appropriately applying per-core DVFS.

We propose a three-step solution to achieve our goal. The algorithm works as follows. In the first step, we reduce power by lowering the frequency on parallel nodes without compromising throughput or latency. This goal can only be achieved for parallel-pipeline topology. If the resulting power consumption still exceeds the power budget, we go to step two. In the second step, we reduce the power with throughput unchanged and minimal latency increase by optimally adjusting the frequency on each core. If both step one and step two can not satisfy the power constraint, we go to step three. In the third step, we reduce the power with minimal throughput and latency performance loss adopting similar approach as in step two. Step three and step two are recursively executed until the power budget is finally met.

It is also important to note that this algorithm is generally applicable to any type of multicore packet processing systems ranging from general-purpose processors to network processors and programmable logic devices as long as per-core DVFS is available. In addition, the scheduling granularity can also vary from fine-grain (e.g., basic block) to course-grain (e.g., loop, function, task) in practice. We implement our algorithm as well as five other conventional algorithms for six real packet processing applications chosen from NetBench [68] and PacketBench [79] on an AMD machine with two Quad-Core Opteron 2350 processors [1]. The five chosen algorithms are Clock Gating (CG) [64] in PM category, and S-SPM [40], PDP-SPM [44], G-SPM [70] and P-SPM [70] in DVFS category. Compared to existing algorithms given the same power budget, our algorithm exhibits substantially better throughput and latency by an average of 64.6% and 25.2%, respectively.

The rest of this chapter is organized as follows. Section 4.2 introduces preliminaries, including the application model, the power model and the problem statement. Section 4.3 presents the optimization model and the three-step power-aware scheduling algorithm in detail. Section 4.4 describes the experimental framework and shows the performance evaluation.

4.2 Preliminaries

4.2.1 Application Model

We define a task graph as a weighted DAG by tuple G=(V, E, C, T), where V=$\{n_i$, i=1:$v\}$ is the set of nodes and v=$|V|$, E=$\{e_{i,j}$=$<n_i, n_j>\}$ is the set of communication edges and e=$|E|$. C is the set of edge communication times and T is the set of node computation times. $c_{i,j}$ is the communication time on edge $e_{i,j}$ and t_i is the computation time on node n_i. Figure 4.1 (a) gives a DAG example.

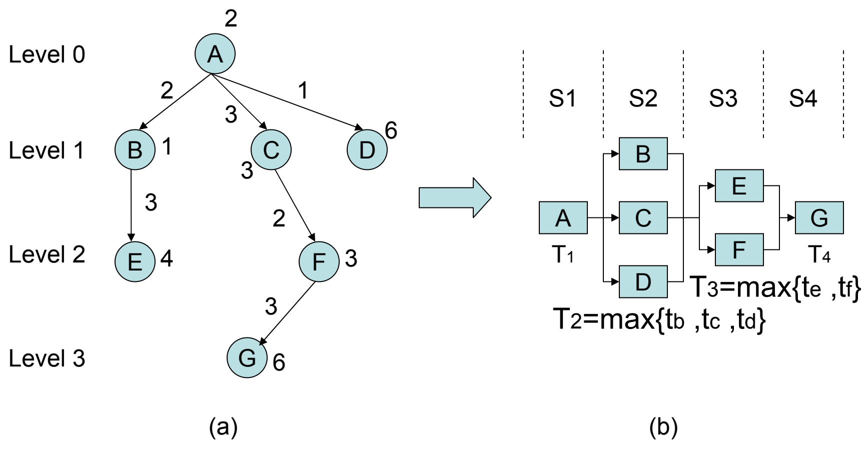

Figure 4.1: A parallel-pipeline scheduling from DAG.

We assume the application is scheduled into a parallel-pipeline topology from the DAG based on a static scheduling policy. Suppose there are N nodes running on N processors with S pipeline stages and M_i parallel nodes in stage S_i. The stage time T_i is the maximal node computation time in S_i. Figure 4.1 (b) illustrates an example of such scheduling. In addition, we can label each node in the parallel pipeline scheduling as node

$N_{i,j}$, where i refers to the stage number and j refers to the node order within a certain stage starting from the top.

We then define the two objective metrics in this chapter: *throughput* and *latency.* In pipeline topology, *throughput* is calculated by the inverse of the longest stage time $\frac{1}{T_{max}}$, where $T_{max} = Max\{T_i\}$, and *latency* is computed as the sum of stage time $T_i, i = 1, 2, ..., S$. We ignore communication time in this chapter without losing validation because it can be considered constant in DVFS scheme [70].

4.2.2 Power Model

Consider that task T consists of C clock cycles on processor P, which runs at voltage V and frequency f. We assume that C does not change with different V and f. For a given voltage V, processor P has an average power consumption Pow. It is known that processor power consumption is dominated by dynamic power dissipation given by: $Pow = K_a \cdot f \cdot V^2$, where K_a is a task/processor dependent factor determined by the switched capacitance.

The energy consumed by executing task T on processor P is computed as: $E = C \cdot \frac{Pow}{f}$. We can rewrite it as: $E = C \cdot E_{f,V} = C \cdot K_a \cdot V^2$, where $E_{f,V}$ is the average cycle energy. From this we can see that lowering the voltage would yield a drastic decrease in energy consumption. The frequency f is almost linearly related to the voltage: $f = K_b \cdot \frac{(V-V_T)^2}{V}$, where V_T is the threshold voltage and K_b is a constant. For a sufficiently small threshold voltage, the frequency is approximated to $K_b \cdot V$.

4.2.3 Problem Statement

Assume the initial parallel-pipeline scheduling with the highest frequency produces the best throughput and latency, which defines the upper bound for throughput and the lower bound for latency. Under such assumption, we give the problem statement as follows: *Given the parallel-pipeline scheduling and power budget, how to optimize the per-core frequency to maximize the throughput and minimize the latency for multicore architectures.*

The problem we want to optimize is: given a set of N cores $P_{1...N}$ that can each run at Q different frequency levels $F_{1...Q}$, find the best selection of frequency levels for all the cores that maximizes the throughput and minimize the latency, subject to the constraint that the total power is less than or equal to POW_{budget}.

We start with the objective functions for throughput and latency in Equation 4.1 and 4.2.

$$Maximize\ \ Th = \frac{1}{T_{max}} \tag{4.1}$$

$$Minimize\ \ L = T_1 + T_2 + ... + T_S \tag{4.2}$$

The execution time of each node t_i can be calculated as $t_i = \frac{C_i}{f_i}$, where C_i is the clock cycles of task n_i and f_i is the frequency on that core. $f_{1..N}$ are the set of frequency levels we are trying to find. Thus, throughput Th and latency L can be rewritten in Equation 4.3 and 4.4.

$$\begin{aligned} Th &= \frac{1}{T_{max}} = \frac{1}{Max\{\frac{C_i}{f_i}\}} = Min\{\frac{f_i}{C_i}\} && (4.3) \\ L &= T_1 + T_2 + ... + T_S \\ &= (\frac{C_1}{f_1} + \frac{C_2}{f_2} + ... + \frac{C_S}{f_S}) && (4.4) \end{aligned}$$

Next, we define the constraint, which specifies that the total power is less than or equal to POW_{budget}. According to [87], we can linearly approximate the power equation as $p_i = b_i f_i + c_i$, where b_i and c_i can be obtained by the linear approximation of the power dependence on frequency [87]. The constraint equation can then be written in Equation 4.5, where all the $c_{1..N}$ constraints are folded into c:

$$b_1 f_1 + b_2 f_2 + ... + b_N f_N + c \leq POW_{budget} \tag{4.5}$$

However, combining Equation 4.3 and 4.4 with Equation 4.1 and 4.2, we can see that these objective functions are not linearly solvable by the conventional Linear Programming (LP) as in [87]. The two reasons are: 1) The total throughput in parallel-pipeline scheduling is not just a linear summation of the partial throughput from all processors. Instead, it only depends on the inverse of the longest stage time. 2) The latency from each stage is inversely proportional to the frequency, which is also non-linear. As a result, we form a new optimization model to this problem in Section 4.3 and propose a novel algorithm to address that model.

4.3 Power-Aware Scheduling Algorithm

In this section, we address the power-aware scheduling algorithm that is capable of optimizing both throughput and latency for parallel-pipeline topology. We first introduce the optimization model, followed by the three-step recursive algorithm in detail. Then, we address the practical issues with discrete frequency levels.

4.3.1 Optimization Model

As mentioned in Section 4.2, we assume the initial parallel-pipeline scheduling with the highest frequency sets the upper bound for throughput and the lower bound for latency. Therefore, if the initial power consumption is already less than or equal to the power budget, the initial scheduling itself is acceptable.

Otherwise, we can reduce the frequency on each core to satisfy the power constraint. Hence, we express the optimization problem as follows: *given the initial throughput Th_0 and latency L_0 with the highest frequency and power budget, minimize the throughput and latency performance loss by optimally adjusting the frequency on each core.*

Equation 4.6 and 4.7 show the two new objective functions. They are inherently equivalent to the one introduced in Section 4.2 (Equation 4.1 and 4.2) but expressed from the complementary direction. We prioritize throughput to latency in our model because: 1) Throughput is still the most important metric for current network packet processing systems. 2) Latency is only required to meet the deadline requirement instead of minimization for most systems.

$$Minimize\ \ \Delta Th = Th_0 - Th \tag{4.6}$$

$$Minimize\ \ \Delta L = L - L_0 \tag{4.7}$$

4.3.2 A Three-Step Recursive Algorithm

Step One: In the first step, we reduce the power without compromising throughput or latency by keeping the pipeline stage time $T_i, i = 1, 2, ..., S$ unchanged. We define a critical node as the node in a pipeline stage that dominates the computation time. Therefore, the computation time of a critical node is equal to the pipeline stage time ($t_i = T_i$). For each stage S_i, we increase the computation time of non-critical nodes in that stage to the length of T_i. Since all stage times remain the same, the throughput and the latency will also keep unchanged during this step.

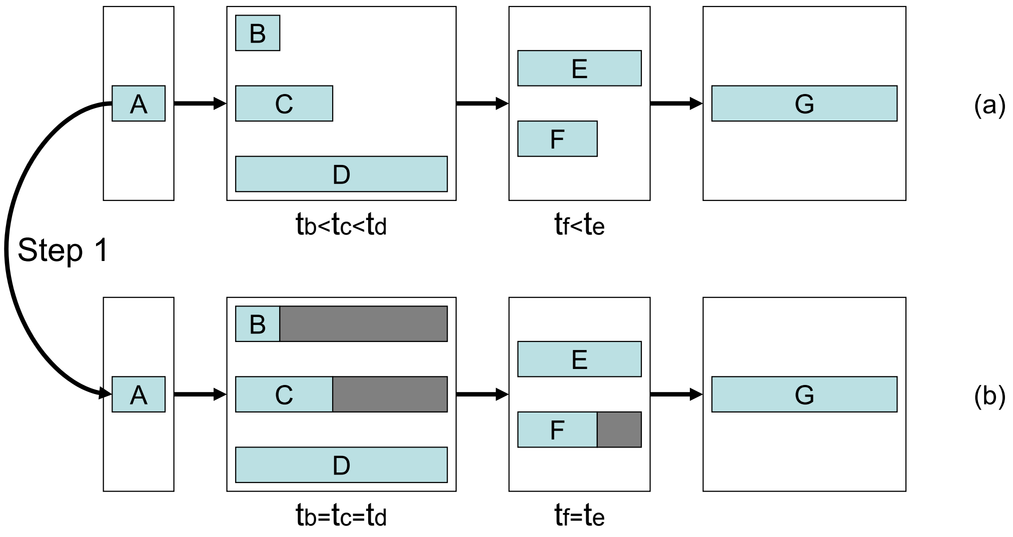

Figure 4.2: Illustration of the first step of the algorithm.

Figure 4.2 illustrates the first step, where the length of a node represents its computation time and grey area represents the extension of computation time. Figure 4.2 (a) is the same as Figure 4.1 (b). In Figure 4.2 (b), we can see that node B and node C are extended to the length of node D in that stage. Similarly, node F is extended to match node E in the next stage. Because increasing computation time and lowering frequency essentially refer to the same meaning [87], we use them interchangeably. As a result, power consumption is reduced by lowering the frequency on nodes B, C and F.

To quantify the power savings in the first step, we derive the formula in Equation 4.8 to calculate ΔP. For each node $N_{i,j}$, $\Delta t_{N_{i,j}}$ represents the difference of computation time.

$$\begin{aligned}
\Delta P &= \Delta P_1 + \Delta P_2 + ... + \Delta P_N \\
&= \sum_{i=1}^{S}\sum_{j=1}^{M_i} \Delta P_{N_{i,j}} \\
&= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot (f - f_{new})) \\
&= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot (\frac{C_{N_{i,j}}}{t_{N_{i,j}}} - \frac{C_{N_{i,j}}}{T_i})) \\
&= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot C_{N_{i,j}} \cdot \frac{\Delta t_{N_{i,j}}}{T_i \cdot t_{N_{i,j}}}) \qquad (4.8)
\end{aligned}$$

If the resulting power consumption after step one is still larger than power budget, we proceed to step two.

Step Two: In the second step, we reduce the power with throughput unchanged and minimal latency increase. This is achieved by keeping the longest stage time T_{max}

unchanged while we increase the stage time of other stages. We denote the stage with T_{max} as the bottleneck stage in the pipeline. Thus, all other stages are non-bottleneck stages.

We define ΔT as the shortest time period by which we can increase the latency. To minimize the latency increase, we iteratively increase the latency by ΔT until the power budget is satisfied or all the stages reach T_{max}. If the former comes true, the algorithm returns and the resulting scheduling guarantees the minimal latency increase, which will be proved shortly. Otherwise, if the latter comes true, we proceed to step three.

In each iteration, we optimally choose a non-bottleneck stage to increase its time from T_i to $T_i + \Delta T$. The candidate stage is chosen by comparing the potential power savings from all non-bottleneck stages. The stage with the largest power reduction will be selected. Because ΔT is the shortest time period that can be increased, and the corresponding power reduction is the largest during each iteration, The algorithm therefore guarantees optimality.

$$\begin{aligned}
\Delta P_i &= \Delta P_1 + \Delta P_2 + ... + \Delta P_{M_i} \\
&= \sum_{i=1}^{M_i} (b_i \cdot (f - f_{new})) \\
&= \sum_{i=1}^{M_i} (b_i \cdot (\frac{C_i}{T_i} - \frac{C_i}{(T_i + \Delta T)})) \\
&= \sum_{i=1}^{M_i} (b_i \cdot C_i \cdot \frac{\Delta T}{T_i \cdot (T_i + \Delta T)}) \qquad (4.9)
\end{aligned}$$

Intuitively, a stage with more parallel nodes will be a good candidate because more power savings will be available in that stage. In fact, besides the degree of parallelism, other parameters also matter. For a given latency increase ΔT, the potential power savings of stage S_i can be obtained from Equation 4.9.

Figure 4.3 illustrates the process of this step. Figure 4.3 (a) comes from the end of step one as shown in Figure 4.2 (b). In Figure 4.3 (b), we extend nodes A, E and F to the length of 6, respectively, to further reduce the power consumption. Now we end up with a scheduling which consists of equal-length pipeline stages.

In fact, Figure 4.3 shows the maximum power savings by step two, which results in maximal latency increase. Chances are that the actual latency would be lower than what we have seen here if the power budget is less stringent, in which case the algorithm would return earlier before every stage reaches T_{max}. According to Equation 4.8, we can also obtain the exact power savings in this step.

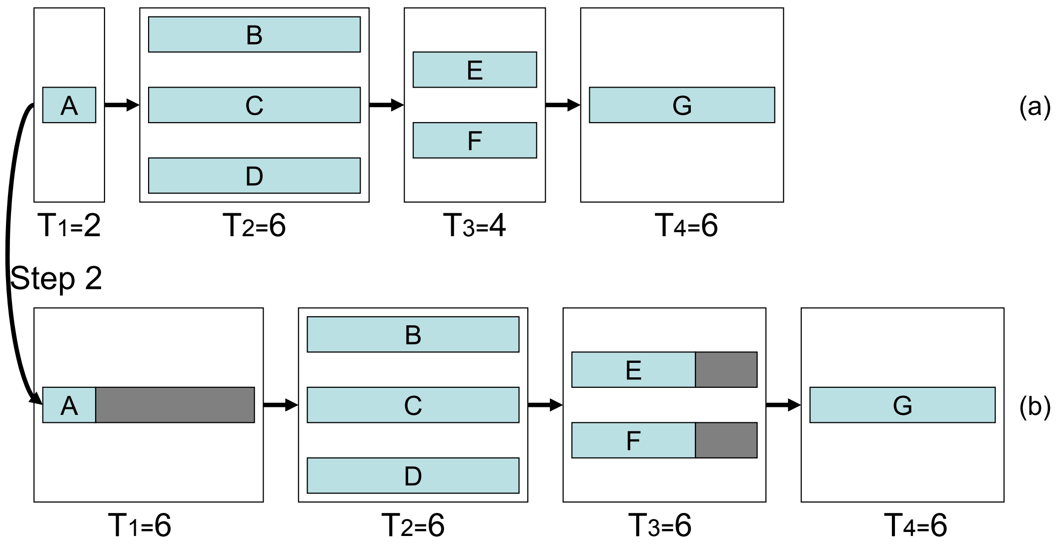

Figure 4.3: Illustration of the second step of the algorithm.

Step Three: In the third step, we reduce the power by minimizing both the throughput and the latency performance loss. Remember that after step two, every stage has the same stage time T_{max}. Following the same rule of choosing a candidate stage in step two, we optimally choose a stage to further increase its stage time by ΔT. Since the original T_{max} is increased, the throughput is compromised accordingly. However, our algorithm is able to guarantee a minimal performance loss in this scenario.

To optimally choose the candidate stage, we follow the same formula in Equation 4.9. The only difference is that we need to substitute T_i with T_{max} in the equation. The proof of optimality is in line with that in step two, where the minimal time period increment guarantees that when we satisfy the power budget constraint, the performance loss is minimal.

Figure 4.4 demonstrates step three. Figure 4.4 (a) is the result of step two as shown in Figure 4.3 (b). Suppose the candidate stage at the moment is stage two. We then increase the stage time from 6 to $6 + \Delta T$ for that stage. Notice that all other stages remain unchanged as shown in Figure 4.4 (b).

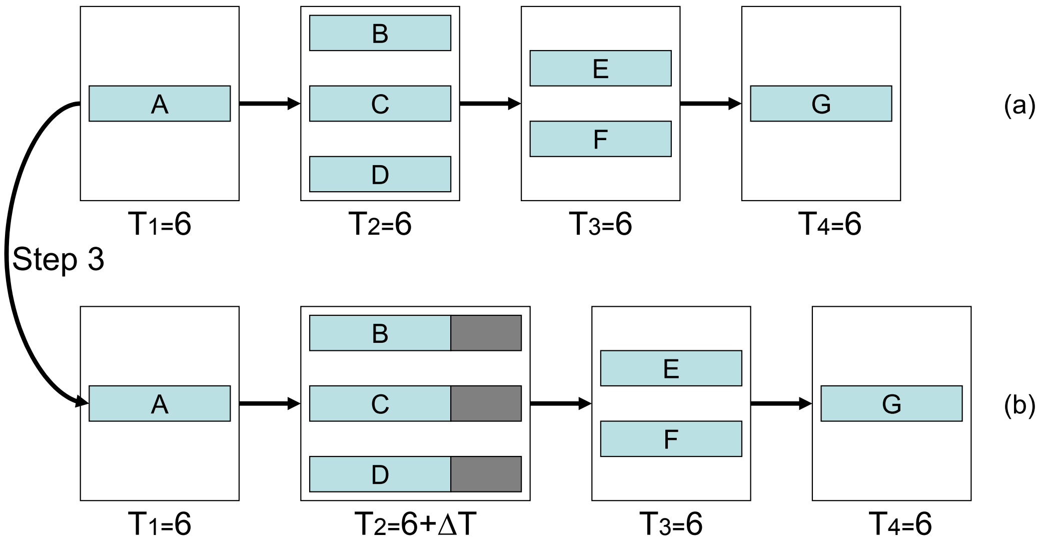

Figure 4.4: Illustration of the third step of the algorithm.

After increasing the original T_{max} to $T_{max} + \Delta T$, we go back to step two with the updated T_{max} if further power reduction is needed. The algorithm then recursively executes step two and step three until the power budget is finally met as shown in Figure 4.5. Algorithm 1 gives the pseudocode for the entire algorithm.

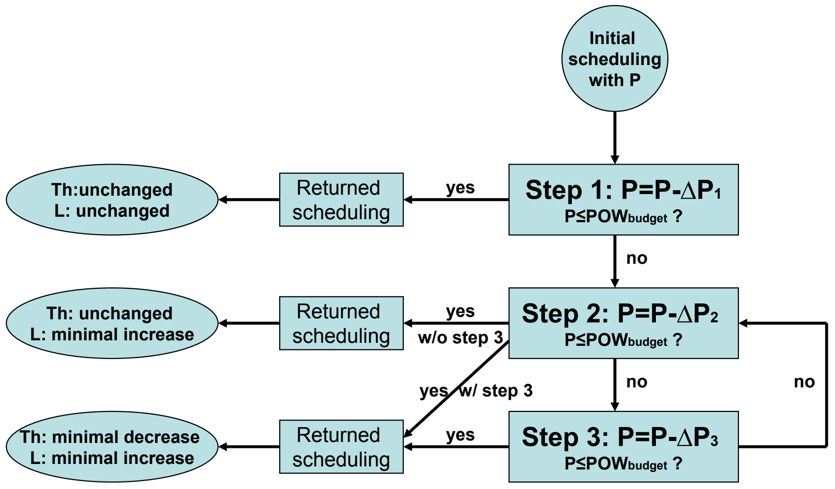

Figure 4.5: The power-aware parallel-pipeline scheduling algorithm.

With respect to the complexity, we conclude that step one has a complexity of $O(N)$, step two has a complexity of $O(\frac{T_{max}-T_{min}}{\Delta T} \cdot S^2 \cdot N)$ and step three has a complexity of $O(m \cdot \frac{T_{max}-T_{min}}{\Delta T} \cdot S^2 \cdot N)$. $\frac{T_{max}-T_{min}}{\Delta T}$ is the maximal number of iterations in step two and m is the maximal recursive times in step three. Because both ΔT and m depend on the number of discrete frequency levels in practice and are thus constant, the total complexity for our algorithm is $O(S^2 \cdot N)$. Therefore, our algorithm will terminate if 1) the power constraint is met or 2) the maximal number of updates is exceeded in either step two or step three. There will be no oscillations occurring in the updates within the while loop as shown in Algorithm 1, which guarantees the convergence of our algorithm.

Algorithm 1

```
1: if $POW \leq POW_{budget}$ then return
2: for each stage $S_i$ do    /* Step 1 */
3:     for each parallel task $n_j$ do
4:         $t_j \leftarrow T_i$
5: if $POW \leq POW_{budget}$ then return
6: while $T_{max}$ unchanged do    /* Step 2 */
7:     for each stage $S_i$ except $T_{max}$ stage do
8:         calculate $\Delta P_i$ according to Equation 9
9:     choose stage $S_i$ with $Max\{\Delta P_i\}$
10:    for each parallel task $n_j$ in $S_i$ do
11:        $t_j \leftarrow t_j + \Delta T$
12:    update $POW$
13:    if $POW \leq POW_{budget}$ then return
14: while $POW > POW_{budget}$ do    /* Step 3 */
15:    for each stage $S_i$ do
16:        calculate $\Delta P_i$ according to Equation 9
17:    choose stage $S_i$ with $Max\{\Delta P_i\}$
18:    for each parallel task $n_j$ in $S_i$ do
19:        $t_j \leftarrow t_j + \Delta T$
20:    update $POW$ and $T_{max}$
21:    if $POW \leq POW_{budget}$ then return else goto 6
```

4.3.3 Practical Issues with Discrete Frequency Levels

So far, we only address the ideal case where the frequency levels are continuous. However, in practice, those values are discrete, which requires some minor corrections in our algorithm.

The essential problem is about the shortest time period ΔT. In practice, the value of ΔT is determined by Equation 4.10, assuming f_1 and f_2 are two contiguous frequency levels and their difference is Δf. As a result, ΔT depends on the clock cycles C_i running on that processor and the current frequency on that processor.

$$\Delta T_i = \frac{C_i}{f_1} - \frac{C_i}{f_2} = C_i \cdot \frac{\Delta f}{f_1 \cdot f_2} \tag{4.10}$$

Thus, the following two changes are necessary in practice. First, in step one, we increase the computation time of non-critical nodes for each stage as much as possible without violating the initial stage time. We do not require that all non-critical nodes be increased to the same length of the critical node in that stage as in ideal case. Second, in step two and step three, we use the practical ΔT value obtained from Equation 4.10 when calculating the potential power savings for each stage.

4.4 Experiments and Evaluation

4.4.1 Experimental Framework

Figure 4.6 shows our experimental framework and flowchart, which consists of two steps. In the first step, we generate the program dependency graph (PDG) with profile information by SUIF/Machine SUIF compilers [13] [16], partition and map the application on an AMD machine with two Quad-Core Opteron 2350 processors [1].

More specifically, the original C program is first converted to the control flow graph (CFG). For the ease of dependency analysis, we include all the functions of an application into one single file. After that, we write a Machine SUIF pass to extract the PDG based on both control and data flow information [34]. Finally, by using the Halt library in Machine SUIF, we profile the program in the basic block level with continuous traffic traces to obtain the average execution time and execution frequency. At last, we schedule the program onto the real machine based on the application model presented in Section 4.2.

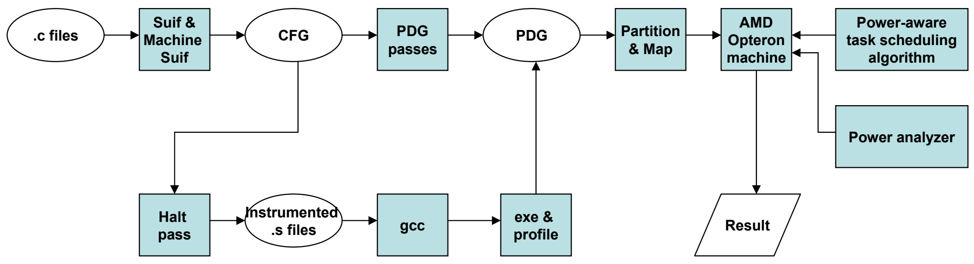

Figure 4.6: Experiment framework and flowchart.

In the second step, we apply our power-aware task scheduling algorithm and five other existing algorithms on the initial parallel pipeline scheduling. The predefined frequency levels of the Opteron 2350 processor are shown in Table 4.1. Meanwhile, the default power for each corresponding frequency is also listed in that table (default power refers to the power consumption when the system is not running our experimental applications). We use the EXTECH power analyzer (model 380801 [5]) to get the whole system power.

Table 4.1: Frequency(GHz) and power(W) configuration.

Frequency Level	1.0	1.2	1.4	1.7	2.0
Default Power	133.0	134.5	137.0	141.2	142.5

The hardware configuration is set by default as an 8-core machine with the instruction cache size up to $4k$ instructions. Six network applications are chosen from NetBench [68] and PacketBench [79]. Their functionalities and code sizes are listed in Table 3.1. For the code size, we measure them in terms of the number of instructions. The packet trace is from NetBench itself, which contains $10,000$ packets. The routing table used for IPv4-trie is MAE-WEST [79] and the routing table size for DRR, IPchains and Route is set to 128 by default. The input file for URL contains 100 lines of rules.

4.4.2 Performance Evaluation

We compare our algorithm with five other conventional algorithms to show its performance advantage in optimizing throughput and latency given the same power budget. Latency refers to the average execution time of one packet in microseconds (usec). Throughput is measured by million packets per second (mpps). Power consumption is measured by watts (w) and we use the net power consumed exclusively by our experimental applications as the metric.

In the PM category, we choose Clock Gating (CG) [64] since it also addresses the energy reduction issue in packet processing for network processors. As regard to DVFS, we choose four different static power management schemes for comparison, namely S-SPM [40], PDP-SPM [44], G-SPM [70] and P-SPM [70]. The reason why we compare with them is that their power-aware schemes are all built on top of task scheduling, which are inline with our algorithm. We briefly introduce them as follows:

- CG (Clock Gating): reduces power consumption by turning off processors.
- S-SPM (Simple SPM): distributes global static slack proportionally to the length of the schedule.
- G-SPM (Greedy SPM): allocates global static slack to the first task on each processor.
- P-SPM (Parallel SPM): distributes global static slack according to the degree of parallelism.
- PDP-SPM (Proportional Distribution and Parallelism SPM): distributes global static slack according to the degree of parallelism and exploits the local slack.

4.4.3 Power Reduction in Step One

Table 4.2: Power and latency after step one (S1) and two (S2).

	URL	Flow	IPv4-trie	Route	DRR	Ipchains	Avg
Power reduction after S1	0.0%	23.1%	11.2%	9.0%	19.8%	0.0%	10.5%
Power reduction after S2	9.8%	34.5%	28.0%	23.8%	31.7%	16.5%	24.1%
Latency increase after S2	18.5%	28.9%	29.9%	19.0%	29.9%	20.9%	24.5%

Figure 4.7 and Table 4.2 show the power reduction after step one for six applications compared with the initial power consumption. Four applications have lowered the

power by an average of 10.5% and a maximum of 23.1% in Flow. These power savings come from the reduced frequency for non-critical parallel tasks as shown in Figure 4.2. URL and IPchains consume the same power because their scheduling can not benefit from step one. Notice that during this process, both throughput and latency keep unchanged, which means the power savings come at no cost.

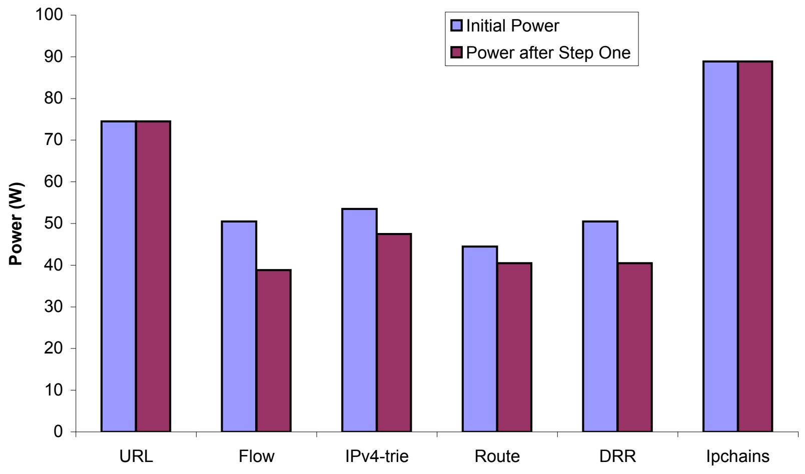

Figure 4.7: Power of six applications after step one.

4.4.4 Power and Latency Performance in Step Two

Figure 4.8, Figure 4.9 and Table 4.2 exhibit the power and latency after step two for six applications compared with the initial results. From Figure 4.8 we observe that all six applications have enjoyed power savings by an average of 24.1% through the frequency adjustment for tasks in non-bottleneck stages corresponding to Figure 4.3. More specifically, Flow achieves the maximum of 34.5% power reduction in this step due to its vastly differing stage times.

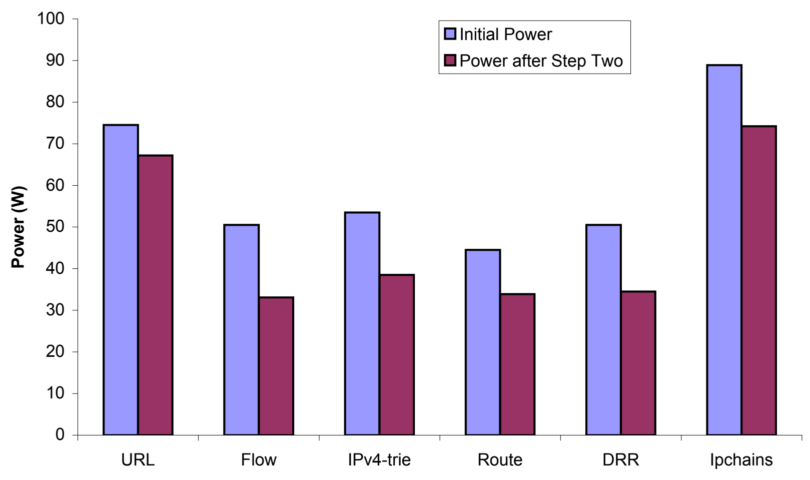

Figure 4.8: Power of six applications after step two.

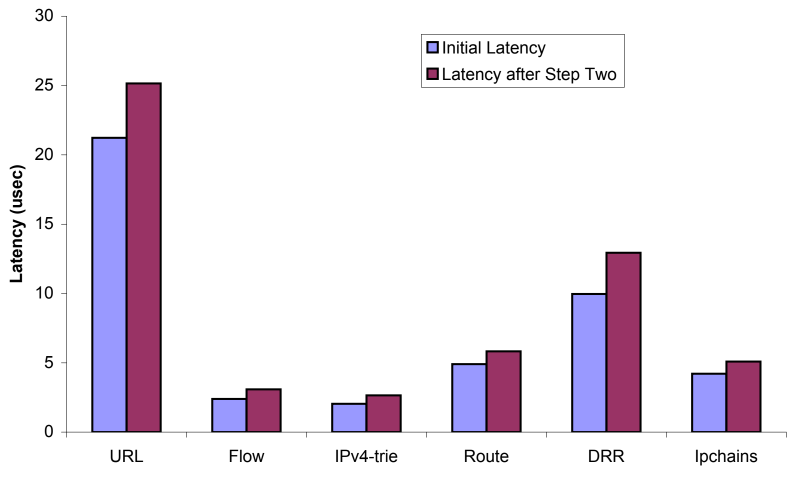

Figure 4.9: Latency of six applications after step two.

From Figure 4.9 we can see that the latency increase ranges from 18.5% in URL to 29.9% in DRR and IPv4-trie with an average of 24.5%. Although we trade latency with power on the same percentage scale, our algorithm guarantees the minimal latency increase while maintaining the throughput unchanged in this step. Moreover, because we demonstrate the maximal possible power reduction and latency increase in step two, chances are that actual latency performance would be even better if the real power budget is less than what we have achieved here.

4.4.5 Throughput and Latency Comparison in Step Three

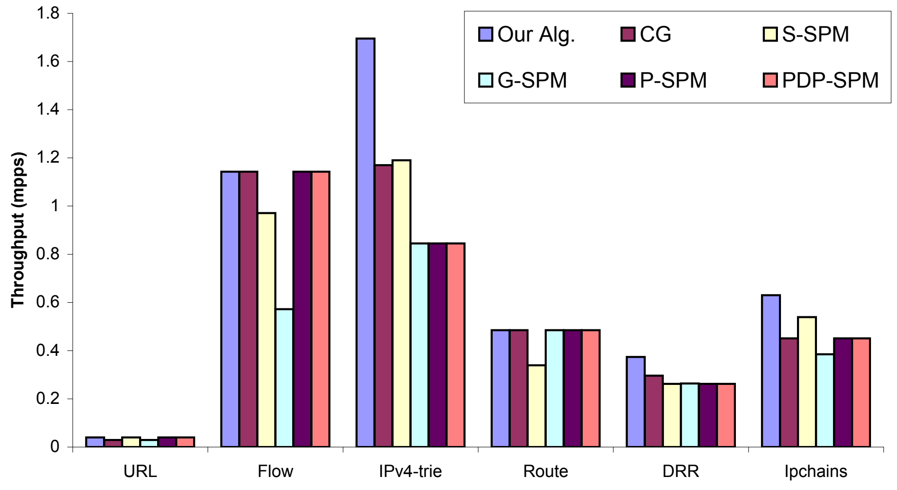

Figure 4.10: Throughput when power budget is 75% of the initial value.

We set the power budget to be 75% of the initial power consumption, so that all three steps in our algorithm will be required to satisfy the power budget constraint. The resulting throughput and latency of six different algorithms are shown in Figure 4.10 and Figure 4.11. With respect to throughput, our algorithm outperforms all others uniformly, with an average improvement of 64.6% compared to the lowest throughput for each application. The maximum increase appears in IPv4-trie where we observe a 100.6% improvement. The other five algorithms exhibit fluctuating performance for different applications as shown in Figure 4.10. This is because each of them has its own shortcoming. For CG, the reduction of active cores results in increased longest stage time, which adversely affects the throughput. For other SPM algorithms, they do not differentiate the bottleneck stage in parallel-pipeline scheduling. Therefore, they can not produce optimal throughput. Our algorithm is capable of achieving better throughput by optimally adjusting the frequency on each core during each step.

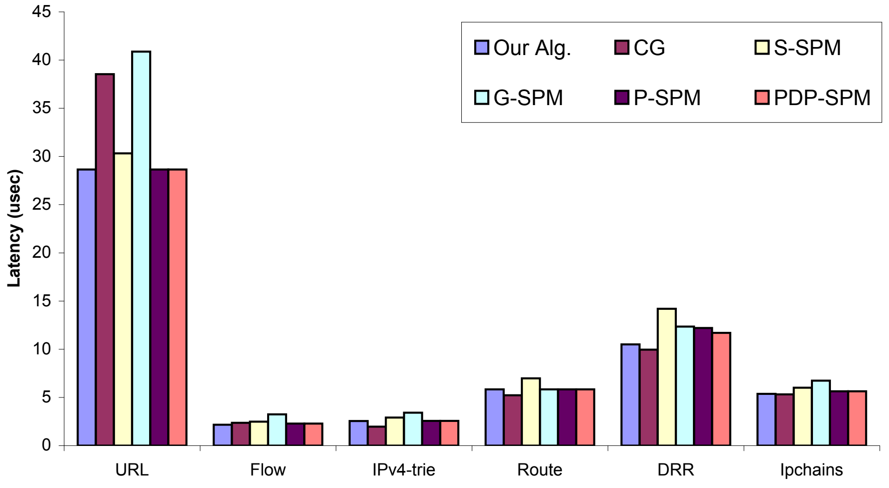

Figure 4.11: Latency when power budget is 75% of the initial value.

In terms of latency, we also observe the advantage of our algorithm from Figure 4.11. Compared with the highest latency for each application, our algorithm results an average of 25.2% latency reduction and the maximum reduction of 33.2% in Flow. PDP-SPM and P-SPM are the best among other SPM algorithms to produce low latency due to their consideration of parallelism. However, they still suffer from longer latency compared to our algorithm in most cases. Our strength lies in the fact that we iteratively apply step three and step two to guarantee the minimal latency increase, whereas PDP-SPM and P-SPM only greedily reduce frequency in parallel stages. On the other hand, we notice that

CG performs better in terms of latency for four applications than our algorithm because it maintains the highest frequency all the time. However, its better latency performance actually comes at the cost of substantial throughput deterioration.

4.4.6 Power Budget Sensitivity Performance

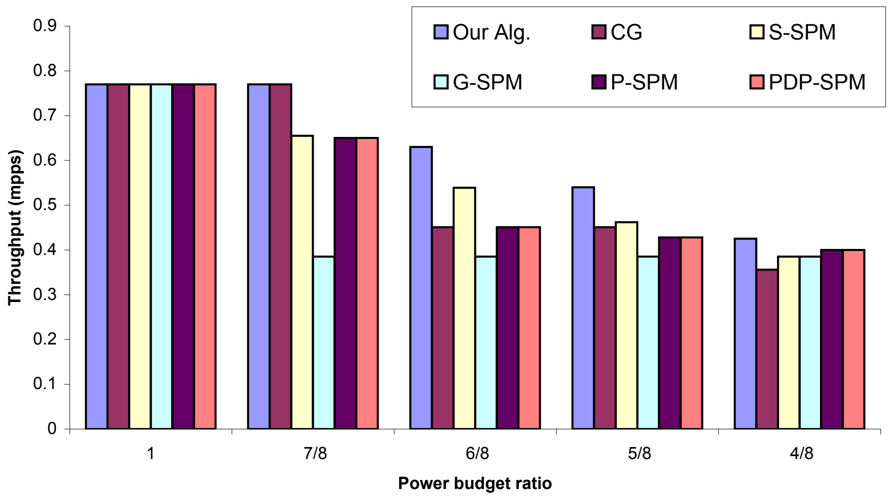

Figure 4.12: Throughput for IPchains when power budget varies.

Lastly, we analyze the effect of varying power budget on throughput and latency. We study a representative application IPchains for six algorithms by changing the power budget ratio from 1 to 4/8 of the initial power consumption. From Figure 4.12 we can see that our algorithm always produces the best throughput. As the power budget decreases, G-SPM suffers most because it always reduces frequencies from the first stage, which happens to be the bottleneck stage in this application. The other four algorithms also follow the same decreasing trend. Although their throughput plummets with different speeds, all of them fall behind our algorithm. On an average, our algorithm exhibits 55.8% improvement on throughput compared to the worst algorithm for each application. The maximum improvement appears to be 100.0% over G-SPM when the power budget ratio is 7/8.

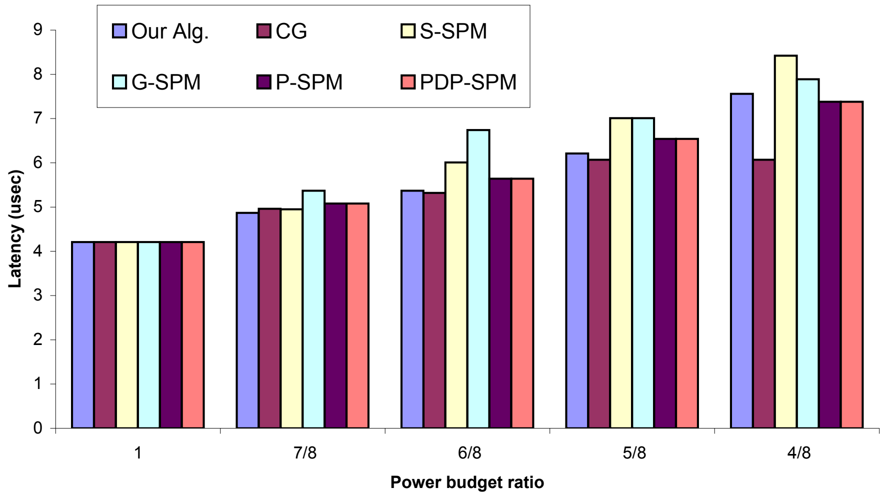

Figure 4.13: Latency for IPchains when power budget varies.

Figure 4.13 illustrates the increasing trend in latency for six algorithms when the power budget decreases. Compared to the highest latency for each application, our algorithm shows an average of 13.0% reduction of latency with a maximum of 20.3% over G-SPM when the power budget ratio is 6/8. We observe that P-SPM and PDP-SPM perform better than S-SPM and G-SPM in general, because both P-SPM and PDP-SPM take into consideration the degree of parallelism. However, these two still attain higher latency due to the lack of specialized optimization for parallel-pipeline topology. CG, on the other hand, has slightly lower latency in some scenarios, for it never reduces frequency. However, as shown in Figure 4.12, CG's throughput deteriorates substantially in those scenarios. Therefore, our algorithm once again proves its advantage in both throughput and latency due to its optimal adjustment of frequency on each core.

Chapter 5

Power-Optimal Scheduling Under Traffic Variation

5.1 Introduction

In the previous two chapters, we address the task-level scheduling for network applications and its associated power-aware optimization scheme. From this chapter on, we focus on packet-level scheduling for network applications. In particular, we take network traffic variation into account in this chapter as power saving potentials. Accordingly, this chapter describes a power-efficient multicore system for network applications which dynamically adjusts system operating level and per-core frequency configuration based on incoming traffic rate. Our on-line algorithm optimizes a novel power model that considers both dynamic and static power. The dynamic per-core frequency configuration is achieved through a combination of per-core DVFS, power gating, and power migration.

We first derive a formula to translate traffic arrival rate to required cumulative core frequency. Then, based on our power model, we derive the optimal system operating level to maintain sufficient system throughput for the current traffic while using minimal dynamic power. Lastly, because each core may be configured at a different operating frequency, we migrate active cores in the system periodically to achieve thermal balancing and reduce peak core temperature. To the best of our knowledge, we are the first to target power optimization considering both dynamic and static power for network applications running on multicore servers.

To verify our design, we implement our approach on a multicore server system with varying traffic loads, running six real network applications from NetBench [68]. Our approach reduces power consumption by an average of 41.0% compared to running with full capacity without any reduction in throughput. Our approach also outperformed three other approaches: chip-wide DVFS [81], power gating [65], and a hybrid combination of chip-wide DVFS and power gating [60].

In summary, this chapter presents the following contributions:

- Proposes a traffic-aware and power-efficient multicore system for network applications by translating incoming traffic rate to optimal per-core frequency configuration.
- Establishes a new power model considering both dynamic and static power and optimizes the power model to manage processor power under varying network traffic.
- Applies per-core DVFS, power gating, and power migration techniques to minimize both dynamic and static power consumption for multicore servers.

- Implements our approach on a multicore server for real network applications and shows substantial improvement in power savings over existing approaches.

The rest of this chapter is organized as follows: Section 5.2 presents our system design which includes the traffic-aware power optimization scheme in a three-step approach. Section 5.3 presents our implementation and performance evaluation.

5.2 Traffic-aware power optimization

5.2.1 System Design

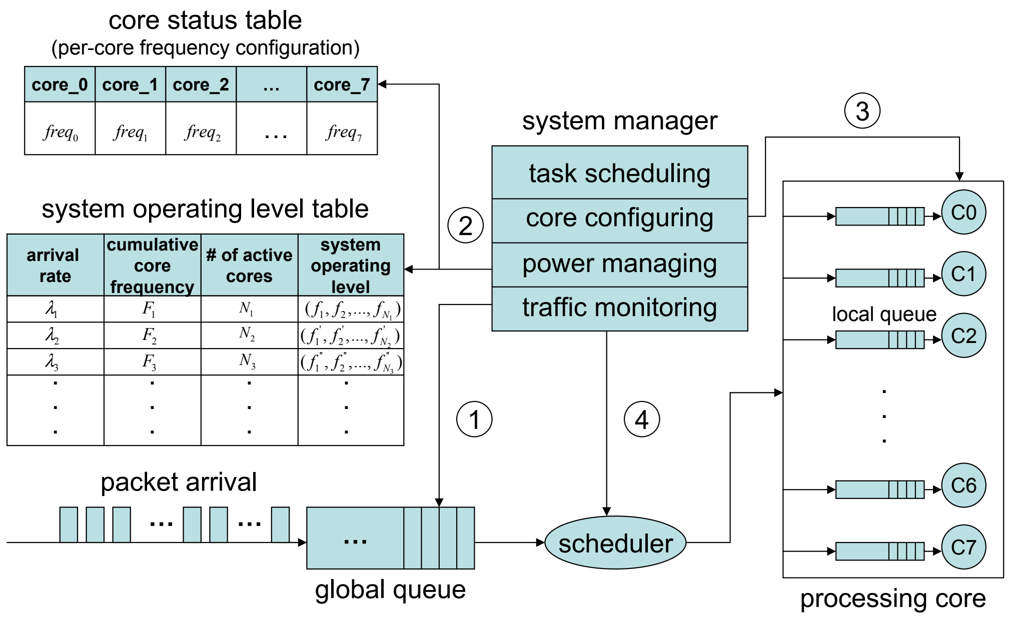

Figure 5.1: Overview of the traffic-aware and power-efficient system.

The typical application supported by this work runs on a multicore server and processes a stream of network requests. Figure 5.1 shows the system overview, where incoming packets from the network are first stored in a global FIFO queue and then scheduled to proper cores for packet processing. The core component is system manager, which consists of four functional modules: traffic monitoring, power managing, core configuring, and task scheduling. We briefly describe the function of each module based on the system manager's operational order.

1. **Traffic monitoring module** tracks packet inter-arrival times to obtain the packet arrival rate and detect the rate change point whenever the traffic rate varies. We monitor the traffic and detect the rate change point by using the sampling technique based on maximum likelihood ratio [77, 81], which is particularly useful for server/router deployment where traffic changes can be predicted only based on prior information.

2. **Power managing module** manages two runtime tables, a system operating level table, which caches optimal system operating level for a given traffic arrival rate, and a core status table, which tracks the actual per-core frequency configuration. The system operating level is represented by tuple $(f_1, f_2, ..., f_N)$ throughout this chapter[1], where $f_1 \geq f_2 \geq ... \geq f_N$. This tuple indicates that N cores are active running and the ith core has the frequency f_i $(1 \leq i \leq N)$. Given a certain arrival rate, the power managing module appropriately derives the optimal system operating level based on our dynamic power optimization scheme. In addition, it also initializes the core status table at each traffic rate change point, and periodically updates the core status table to enable power migration for active cores between two consecutive traffic rate change points based on our static power optimization scheme.

[1]The system operating level does not physically specify which frequency goes to which particular core. It only virtually contains an array of core frequencies optimized for a given traffic rate.

3. **Core configuring module** adjusts the frequency level of each core based on the information from the core status table managed by the power managing module. At each configuration point, it applies power gating to cores labeled as inactive as soon as their local queues become empty, and applies per-core DVFS for cores labeled as active and adjusts their frequencies according to their respective configurations chosen from one of the five frequency levels, namely 1GHz, 1.2GHz, 1.4GHz, 1.7GHz and 2GHz. The core configuring module is critical in the system because it is where the three applied power techniques are actually enabled.

4. **Task scheduling module** appropriately schedules packets in the global queue to active cores in our system. As there is no inter-packet dependency for network applications, every packet can be independently dispatched to any active core. When the per-core frequency configuration updates, the scheduler stops sending packets to power-gated cores. For active cores with different core frequencies, the scheduler distributes the workload, which is determined by the number of packets, per-packet size, and application type, in proportion to the core frequency. This approach achieves weighted load balancing across cores under varying traffic rate and avoids loss of system throughput due to the change of system operating level and update of per-core frequency configuration.

As we focus on the power managing module in this chapter, we propose a three-step approach as shown in Figure 5.2 to solve the power optimization problem.

5.2.2 Step 1: System Service Model

The system service model translates the traffic arrival rate to required cumulative core frequency in the multicore system. As traffic rate varies, an ideal cumulative core frequency should be just sufficient to satisfy the demand without over-provisioning. To quantitatively establish the relationship between arrival rate and cumulative core frequency, we use service rate, which is the aggregated service capability from all cores, as an intermediate. Thus, our system service model involves two parts: 1) relationship between arrival rate and required service rate and 2) relationship between service rate and required cumulative core frequency.

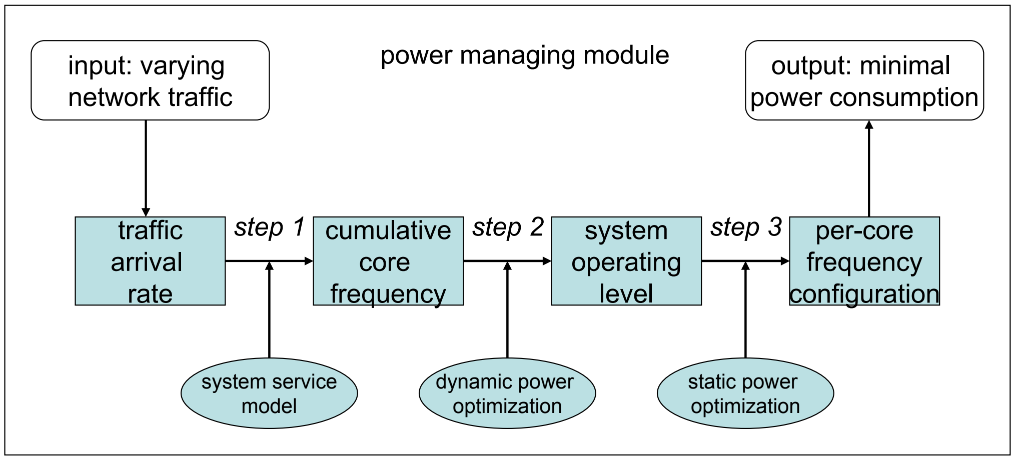
power managing module
input: varying network traffic
output: minimal power consumption
traffic arrival rate
step 1
cumulative core frequency
step 2
system operating level
step 3
per-core frequency configuration
system service model
dynamic power optimization
static power optimization

Figure 5.2: A three-step power optimization scheme.

In the first part, we let service rate equal arrival rate whenever the network traffic varies. This simple but effective design choice is based on the following observations. First, to guarantee a stabilized system without packet overflow, service rate should be no less than arrival rate. Second, to avoid over-provisioning and achieve power efficiency, service rate should be no greater than arrival rate. Considering both constraints, we let service rate be the same as arrival rate.

In the second part, [45,81] have shown that the service rate is linearly proportional to the CPU frequency for a single-core system. However, because we target multicore architectures where different cores may run at different frequencies, it is necessary to re-think and justify the relationship between the service rate and cumulative core frequency. We, therefore, conduct two empirical trace-driven studies with 6 chosen network applications from NetBench on our multicore machine to help establish the relationship.

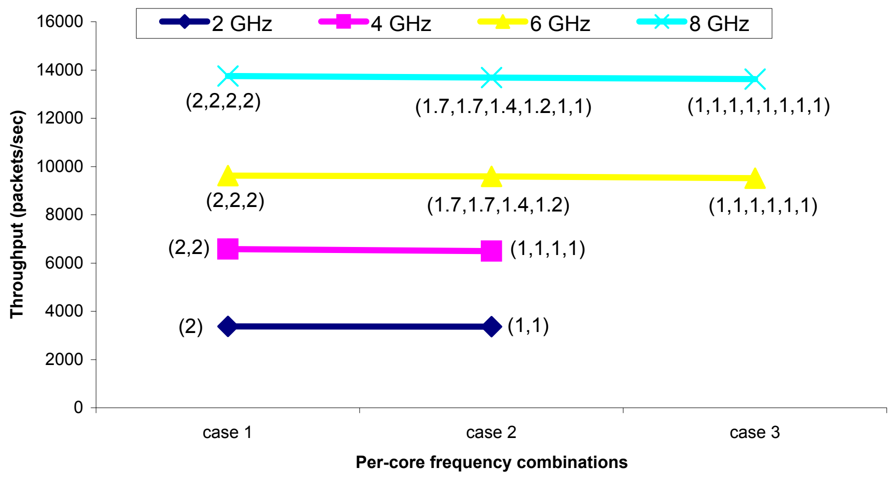

Figure 5.3: Throughput versus different per-core frequency combinations.

First, we examine the effect of various per-core frequency combinations versus throughput given the same cumulative core frequency. Figure 5.3 shows the results of the URL application when we vary the cumulative core frequency from 2GHz to 8GHz. For each cumulative core frequency, we change the per-core frequency combinations. From this figure, we observe that the throughput (service rate) only depends on cumulative core frequency, regardless of per-core frequency combinations. In fact, this observation is obvious for network applications with packet-level parallelism running on multicore servers using weighted load-balanced task scheduling. When incoming packets are independently processed on multiple cores with different service rates due to different frequencies, we can equivalently treat this multicore server as a single-core system with the aggregated service rate equal to the sum of per-core service rate, regardless of per-core frequency combinations.

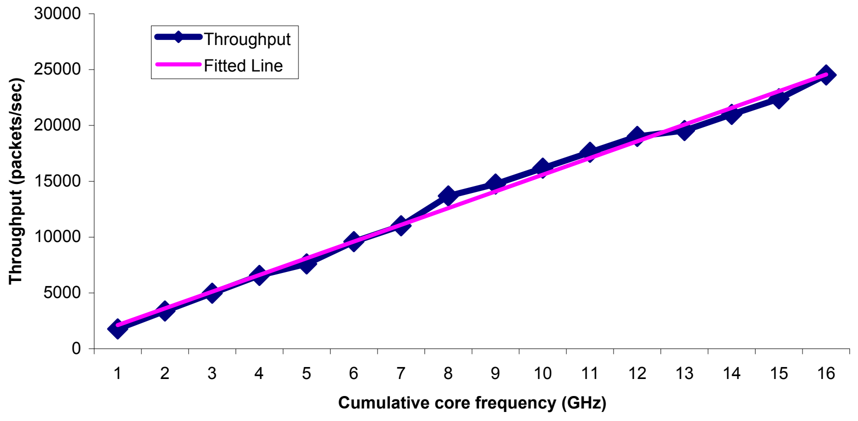

Figure 5.4: Throughput versus cumulative core frequency.

Second, we build the relationship between the service rate and cumulative core frequency in our system. We vary the cumulative core frequency from the minimum (1GHz) to the maximum (16GHz) and record the system throughput. The results show that for our multicore server, the throughput (service rate) is also linearly proportional to the cumulative core frequency. Figure 5.4 illustrates both the experiment result and the fitted line for the URL application (Other applications have the similar results with different parameters and coefficients). Therefore, our system service model for the URL application is given by the linear function in Equation 5.1, where X represents cumulative core frequency and Y represents the service rate, or the arrival rate in our case.

$$Y = 1496 \cdot X + 628 \tag{5.1}$$

5.2.3 Step 2: Dynamic Power Optimization

The dynamic power optimization scheme takes cumulative core frequency as input and produces the optimal system operating level as output. More specifically, it answers the following three questions: Q1) what is the theoretically optimal number of active cores from our power model? Q2) what is the actual number of active cores considering the integer constraint? and Q3) what is the frequency assignment for active cores considering the discrete frequency levels?

Power Model

Consider a network application running on a core at voltage v and frequency f. The dynamic power consumption is given by: $P_{dynamic} = K_a \cdot f \cdot v^2$, where K_a is a task/core dependent factor determined by the switched capacitance. Besides, the frequency f is almost linearly related to the voltage $v : f = K_b \cdot (v - v_t)^2/v$, where v_t is the threshold voltage and K_b is a constant. For a sufficiently small threshold voltage, the frequency is approximated to $K_b \cdot v$. Therefore, we assume the dynamic power consumption is cubic to the frequency as shown in Equation 5.2, where $K = K_a/K_b^2$.

$$P_{dynamic} = K \cdot f^3 \tag{5.2}$$

With respect to static power, we assume that for power-gated cores, they consume zero static power. For active cores, static power consumption is exponential to core temperature [85]. However, as this step focuses on dynamic power optimization, we ignore the temperature effect and assume the static power of each active core is constant, P_s. Detailed discussion for static power consumption is given later because it is related to thermal balancing and power migration. Thus, the total power consumption of an active core is given by Equation 5.3:

$$P_{core} = P_{static} + P_{dynamic} = P_s + K \cdot f^3 \tag{5.3}$$

In a multicore system with N active cores, suppose f_i is the frequency on core i and $P(f_1, f_2, ..., f_N)$ is the total system power consumption as a function of system operating level denoted as $(f_1, f_2, ..., f_N)$. We have the following:

$$P(f_1, f_2, ..., f_N) = N \cdot P_s + K \cdot (f_1^3 + f_2^3 + ... + f_N^3) \tag{5.4}$$

Now we formulate the objective function for our power optimization problem as shown in Equation 5.5, assuming F is the required cumulative core frequency.

$$Minimize \quad P(f_1, f_2, ..., f_N); \tag{5.5}$$

$$s.t. \quad f_1 + f_2 + ... + f_N \geq F$$

In the following, we focus on answering Q1 in a quantitative approach under the assumption that the core frequency is continuous. Later on, to answer Q3, we will relax this constraint in practical scenario with discrete frequency levels.

Suppose we have x active cores to handle cumulative core frequency F. As the dynamic power is proportional to the cube of core frequency, we know that when every active core is running at the same frequency of F/x, the total dynamic power consumption reaches minimum. Thus, from Equation 5.4, we can derive the total power consumption as follows.

$$\begin{aligned} P = P(f_1, f_2, ..., f_x) &= x \cdot P_s + K \cdot (f_1^3 + f_2^3 + ... + f_x^3) \\ &\geq x \cdot P_s + x \cdot K \cdot (F/x)^3 = x \cdot P_s + \frac{K \cdot F^3}{x^2} \end{aligned} \tag{5.6}$$

This function ($P = x \cdot P_s + \frac{K \cdot F^3}{x^2}$) is a unimodal function and has a global minimum as illustrated in an example in Figure 5.5. This curve is drawn for the URL application when we set $P_s = 5.8$, $K = 1.6$ and $F = 3$. More details about the parameters can be found in Section 5.3. It shows that starting from a single active core ($x = 1$), increasing the number of active cores will reduce the total power consumption while satisfying the cumulative core frequency requirement, until the number of active cores increases past a certain threshold value. We call this value x^*, which is the optimal number of active cores that strikes a good balance between static and dynamic power. In fact, from the classic algebra inequality as shown in Equation 5.7, we can easily solve the problem in Equation 5.8.

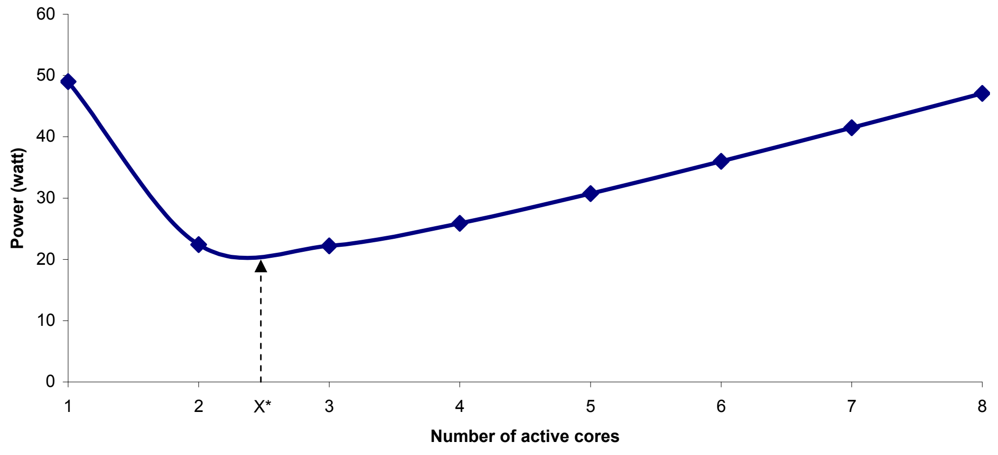

Figure 5.5: Power consumption as the number of active cores varies.

$$\frac{a+b+c}{3} \geq \sqrt[3]{a \cdot b \cdot c} \tag{5.7}$$

when a=b=c, left side reaches minimum.

$$\begin{aligned} P &= x \cdot P_s + \frac{K \cdot F^3}{x^2} = \frac{x \cdot P_s}{2} + \frac{x \cdot P_s}{2} + \frac{K \cdot F^3}{x^2} \\ &\geq 3 \cdot \sqrt[3]{\frac{P_s^2 \cdot K \cdot F^3}{4}} \end{aligned} \tag{5.8}$$

In addition, the minimal power consumption is achieved if and only if Equation 5.9 is satisfied.

$$\frac{x \cdot P_s}{2} = \frac{K \cdot F^3}{x^2} \Rightarrow x^* = \sqrt[3]{\frac{2K \cdot F^3}{P_s}} \tag{5.9}$$

Integer Constraint

In answering Q1, we allow the fractional number of cores and do not consider the boundary conditions in the previous section. However, in practice, the number of active cores must be an integer between 1 and C_{max}, the max number of cores in the system. Now, we address Q2 in consideration of this constraint. Suppose x^* from Equation 5.9 is the theoretically-derived optimal number of active cores. Firstly, if $x^* < 1$, we choose 1 active core, and if $x^* > C_{max}$, we choose C_{max} active cores.

Secondly, we discuss the scenario when $1 < x^* < C_{max}$. From Equation 5.6 we know that the power function is monotonically decreasing when $x < x^*$ and monotonically increasing when $x > x^*$. Therefore, the nearest integers of x^*, $floor(x^*)$ and $ceiling(x^*)$, will be candidates for the practical number of active cores with minimal power consumption. For example, in Figure 5.5, when x^* has a value of 2.5, we can either use 2 active cores with higher per-core frequency or 3 active cores with lower per-core frequency to satisfy the throughput demand.

To choose between $floor(x^*)$ and $ceiling(x^*)$, we can simply calculate their respective power consumption based on Equation 5.4 and select the one with lower value.

If $floor(x^*)$ is chosen, Equation 5.10 has to be satisfied.

$$floor(x^*) \cdot P_s + \frac{K \cdot F^3}{floor(x^*)^2} \leq ceiling(x^*) \cdot P_s + \frac{K \cdot F^3}{ceiling(x^*)^2} \tag{5.10}$$

Let $floor(x^*) = X$ and $ceiling(x^*) = X + 1$, we have:

$$X \cdot P_s + \frac{K \cdot F^3}{X^2} \leq (X+1) \cdot P_s + \frac{K \cdot F^3}{(X+1)^2} \tag{5.11}$$

By solving Equation 5.11, we conclude that if $F \leq \sqrt[3]{\frac{P_s}{K} \cdot \frac{X^2 \cdot (X+1)^2}{2X+1}}$, we choose $floor(x^*)$ as the best integer number of active cores. Otherwise, we choose $ceiling(x^*)$.

Frequency Assignment

After obtaining the optimal number of active cores, we address Q3, the frequency assignment problem to appropriately assign frequency to each active core considering the discrete frequency levels. We propose two rules to guide the frequency assignment.

- *Rule* 1*: Always provide the minimal cumulative core frequency that satisfies the traffic demand.*
- *Rule* 2*: For a given cumulative core frequency, the per-core frequency combination with the least standard deviation consumes the least power.*

To demonstrate the two rules, we carry out two empirical studies with the same settings as in Section 5.2.2. In the first study, we vary the cumulative core frequency from 2GHz to 8GHz. For a given cumulative core frequency, we vary the per-core frequency combinations and record the net power consumption (load power minus idle power). Figure 5.6 shows the results for the URL application. From this figure, we observe that power consumption varies substantially, as much as 114% when comparing $(2, 2)$ to $(1, 1, 1, 1)$, with different per-core frequency combinations, which indicates that a proper frequency assignment is very necessary for multicore servers supporting per-core DVFS and power gating.

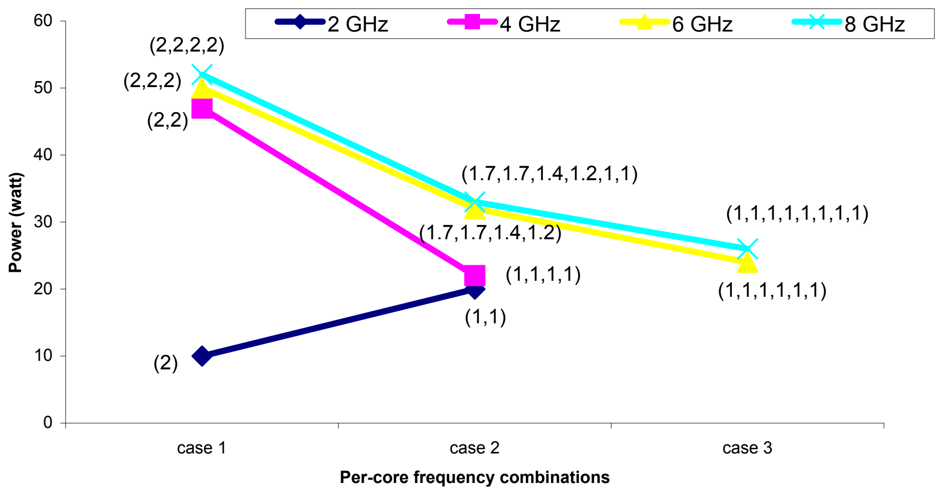

Figure 5.6: Power versus different per-core frequency combinations.

In the second study, we take two cores and change the frequency in all possible combinations and record the power consumption. Figure 5.7 shows the 3D plot for the results with the URL application, where each black point represents a per-core frequency combination and its corresponding power consumption. In addition, based on Figure 5.7, we also plot Figure 5.8 illustrating the power consumption versus cumulative core frequency. When a certain cumulative core frequency corresponds to multiple power consumptions, we take the minimal one. From these two figures, we notice: 1) If we only consider the minimal power consumption for a given cumulative core frequency as shown in Figure 5.8, we see higher cumulative core frequency corresponds to higher power consumption. 2) For the same cumulative core frequency, the more evenly-distributed per-core frequency combination results in less power consumption as shown in Figure 5.7. For example, point

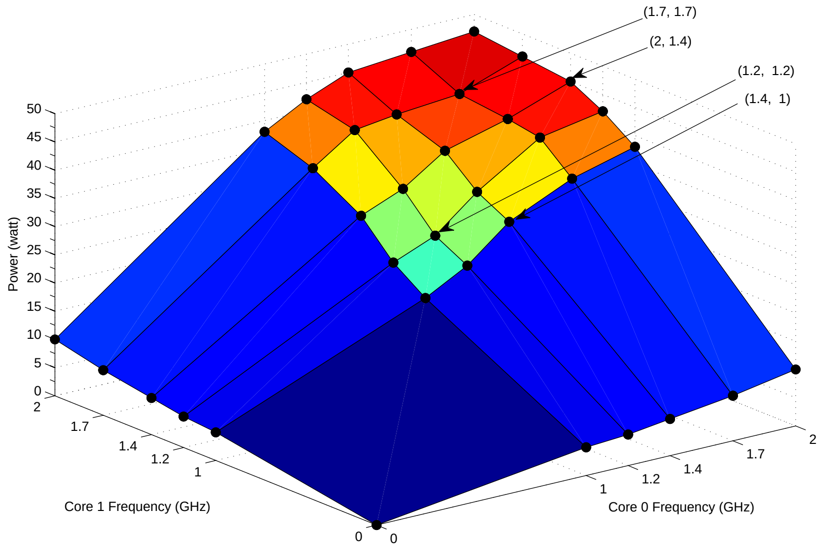

Figure 5.7: Power versus two-core frequency combinations.

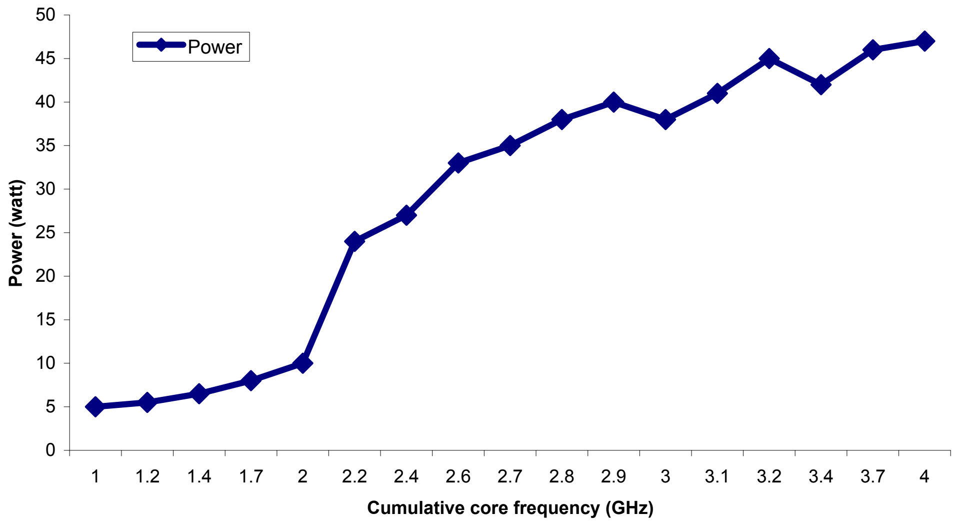

Figure 5.8: Power versus cumulative core frequency with two cores.

$(1.7, 1.7)$ is lower than point $(2, 1.4)$ and point $(1.2, 1.2)$ is lower than point $(1.4, 1)$, although they have the same cumulative core frequency in both cases. In summary, this exhaustive study empirically validates our two rules.

5.2.4 Step 3: Static Power Optimization

The static power optimization scheme takes system operating level as input, which virtually contains an array of core frequencies optimized for a given traffic rate, and produces as output the actual per-core frequency configuration that is dynamically updated for power migration. This step focuses on the power migration design for active cores to achieve thermal balancing and reduce peak core temperature that effectively reduces static power consumption.

Design Overview

While keeping the system operating level constant, we appropriately vary the physical location for active cores so that thermal balancing is achieved across all cores. If the time interval between two migration points is small enough, we can minimize peak core temperature and effectively reduce static power consumption. Figure 5.9 illustrates the overview of our power migration scheme with varying network traffic rate. At each traffic change point (T_i), we apply the dynamic power optimization scheme to obtain the optimal system operating level. Because the number of active cores may be less than the total number of cores, and per-core frequency is heterogeneous, we periodically redistribute the power dissipation at each migration point (t_i) among all cores. In Figure 5.9, color squares represent active cores with different frequency levels (the darker the color, the higher the frequency), whereas white squares represent power-gated cores. In this example, the migration process happens among core pairs (C1, C2), (C6, C4), and (C5, C3), where the highest frequency cores C1, C6 and C5 are swapped with the lowest frequency cores C2, C4 and C3.

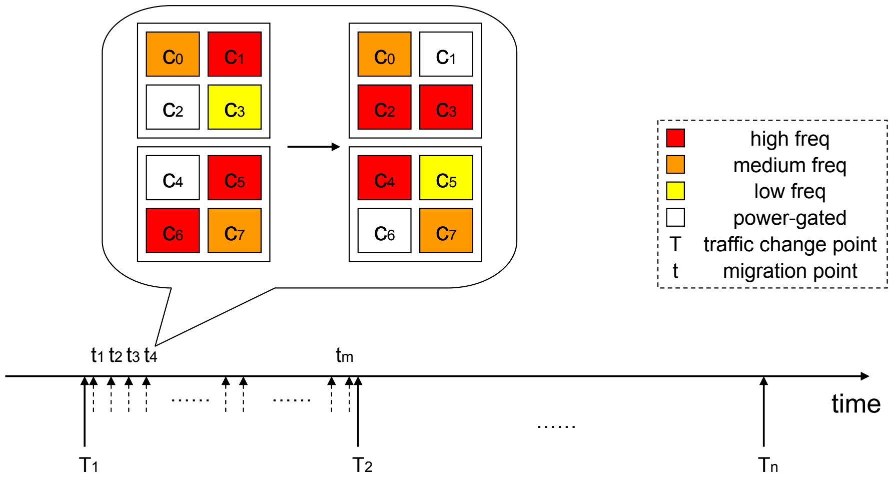

Figure 5.9: Illustration of power migration for active cores.

As mentioned earlier, the power managing module periodically updates the core status table to guide power migration in the system. For network applications that feature in periodic packet processing, power migration only involves the swapping of operating frequency among cores (through per-core DVFS and/or power gating). The task scheduling module will redistribute future workload based on updated core status table accordingly. Therefore, there is no actual data/state transfer between two cores, avoiding substantial migration overhead.

Migration Policy

The policy of our power migration consists of both long-term update and short-term update of core status table. The long-term update refers to the initialization of core status table at each traffic change point, which is on the order of minutes based on network traffic studies in [51, 65, 81]. The short-term update refers to the periodic update of core status table at each migration point between two consecutive traffic change points. Considering core thermal behavior, packet processing time and system reconfiguration overhead, we find an update frequency of 1 second to be a good value for short-term update in our system.

Long-term update: The long-term update should be based on previous history in the core status table for thermal balancing. At the traffic change point, because the system operating level will change in terms of the number of active cores and core frequency, we want every core to have even power dissipation over a period of time. For example, to evenly distribute the power consumption from the highest frequency core in the previous configuration, we should initially make it power-gated or assign the lowest frequency to it. Therefore, our long-term update can be described as follows:

General policy: Given the current system operating level $(f_1, f_2, ..., f_N)$*, we first sort the per-core frequency configuration in the previous core status table according to the frequency level from the lowest to highest. Then, we assign frequency* f_1 *to the first core in that list and frequency* f_2 *to the second core in that list and so on. For cores that are not assigned a frequency level, we leave them to be power-gated.*

We have one exception for the above-mentioned long-term update. As we target servers with multicore processors, we should use as few processors as possible while satisfying traffic demand. Thus, when all active cores can fit into one processor, we should always use only one processor. Considering the general policy, we add the following exception rule:

Exception: If all active cores can fit into one processor, we choose the processor which contains the core that is assigned the frequency f_1.

Short-term update: The short-term update aims to achieve thermal balancing across all cores in the system through power migration. However, we can not adopt a simple round-robin or random migration policy because system operating level changes over time. We argue that the migration policy has to be temperature-aware to guarantee thermal balancing during two consecutive short-term updates. Because core frequency is directly related to core temperature and per-core frequency is easy to obtain, we propose a frequency-aware migration policy to guide the short-term update as follows:

General policy: We sort the per-core frequency configuration in the current core status table according to the frequency level from the lowest to highest. Then, we swap the frequency between the first core in the list and the last core, and between the second core and the last but one core, and so on.

This strategy lets the power dissipation be evenly distributed across all cores during two consecutive short-term updates; thus overall thermal balancing will be achieved as expected. In the exceptional case where only one processor is used, we apply the following rule:

Exception: If all active cores can fit into one processor, we switch the processor at every migration point and copy the same per-core frequency configuration within a processor from one to the other. However, at every other migration point, we update the per-core frequency configuration within a processor following the short-term update general policy.

This exception rule ensures we will keep the frequency assignment to one processor when it is possible. Using the regular short-term policy without this exception will likely split the frequency assignment across multiple processors.

5.3 Experimental Evaluation

5.3.1 Experiment Setup

We implement our scheme along with three other schemes on an AMD server with two Quad-Core Opteron 2350 processors, running Linux-2.6.35 kernel. For power measurement, we use a power analyzer (model EXTECH 380801 [5]) to obtain the real-time whole system power. We use the net power consumed exclusively by network applications as the metric for fair comparison. Net power is obtained by subtracting idle power (measured when no application is executing) from load power (measured when network application is executing on partial or all cores).

In our experiment, per-core DVFS is achieved by setting the core frequency to one of the five predefined frequency levels: 1GHz, 1.2GHz, 1.4GHz, 1.7GHz and 2GHz. We rely on the Linux kernel CPUfreq subsystem to implement the frequency scaling. Power gating is achieved by removing cores from active working set based on kernel's built-in

CPU “hotplug” support, which mimics precisely the behavior of power gating [62]. Task scheduling module achieves power migration by dynamically scheduling incoming packets to active cores (as discussed in Section 5.2.1).

Table 5.1: Six network applications from NetBench.

Name	Functionality	Category
CRC	CRC-32 checksum calculation	Micro level
TL	Radix-tree table lookup routine	Micro level
Route	IPv4 routing based on radix	IP level
DRR	Deficit-round robin scheduling	IP level
URL	URL-based switching	Application level
MD5	Message digest algorithm	Application level

We parallelize six network applications from NetBench [68] (as listed in Table 5.1) and execute them in a multi-threaded fashion with packet-level parallelism. To guarantee each active core is running a thread, we enforce thread-to-core binding by setting thread affinity. We select two applications from each category (i.e., Micro-level, IP-level and Application-level). The packet trace is from NetBench with $10,000$ packets, which are repeatedly processed in our experiment. The packet size ranges from 40 bytes to 1500 bytes with an average of 723 bytes. The routing table size for TL, Route and DRR is 128, and we use the small_input file for URL.

Table 5.2 shows application-specific parameters. We profile each application and obtain their system service model, where X represents the cumulative core frequency and Y represents the service rate (packets/sec), equivalent to the arrival rate. To quantify the workload for weighted load balancing scheduling, we also obtain the per-packet latency for each application when running on a single core with 2GHz frequency. We observe that the packet processing time is either constant or linear with respect to the packet size. We derive the dynamic power parameter K for each application based on Equation 5.2 and Equation 5.3 by substituting known frequency and measured power consumption. In addition, to calculate the static power P_s, we refer to AMD Opteron 2350 specification, and use $V_{dd} \in (1.06V, 1.35V)$ and $I_{leak} \in (4.2A, 5.3A)$ as the $65nm$ technology parameters [62]. Hence, we take the average of 5.8W as the input for our power model.

Table 5.2: Application-specific parameters.

App.	System Service Model	Latency (μs)	K
CRC	$Y = 81109 \cdot X + 26662$	0.008·size+0.3	1.5
TL	$Y = 571389 \cdot X + 328743$	0.8	1.4
Route	$Y = 253707 \cdot X + 87975$	1.8	1.4
DRR	$Y = 74965 \cdot X + 54945$	5.5	1.4
URL	$Y = 1496 \cdot X + 628$	0.131·size+73.2	1.6
MD5	$Y = 76016 \cdot X + 35024$	0.005·size+3.2	1.8

To achieve traffic variation, we experiment with both synthetic and real-world workloads. For synthetic workload, we set the required cumulative core frequency (F) for incoming traffic to be one of the following five cases (as shown in Table 5.3). For real-world workload, we take the 24-hour traffic as shown in Figure 1.1. We consider the total volume as the arrival traffic for packet processing (i.e., traffic rate varies between 320K packets/sec and 720K packets/sec), and sample 24 different average traffic rates at each hour to obtain the required cumulative core frequency. Without loss of generality, the cumulative core frequency is then scaled according to our system capacity from 1GHz to 16GHz. For both workloads, we change the traffic rate every minute and set the power migration frequency to be 1 second.

Table 5.3: Synthetic workload for different traffic rate.

Traffic Rate	extra low	low	medium	high	extra high
F	1GHz	4GHz	8GHz	12GHz	16GHz

In the experiment, we first compare our scheme to a native system without power management. Since the native system always runs with full capacity, our scheme can substantially outperform it in terms of power savings without throughput reduction. In addition, we compare our scheme with three other power-aware schemes, i.e., PG [65], which turns off cores when traffic is light using power gating, C-DVFS [81], which assumes a unified frequency adjustment across all cores using chip-wide DVFS, and C-Hybrid [60], which combines both chip-wide DVFS and power gating. Again, compared to these schemes, our scheme can achieve more power savings.

5.3.2 Power Savings

Figure 5.10 shows power savings percentage for our scheme under different synthetic workloads compared to a native system. We observe that our scheme can achieve power savings in four out of five rates ranging from 18.0% for the CRC application in high traffic rate, to as high as 90.0% for the URL application in extra low traffic rate. The only exception is the extra high traffic rate, where all the cores must be running at the maximal 2GHz and no power-saving potentials can be explored. Overall, our scheme reduces an average of 41.0% power consumption for the six applications and their five different workloads.

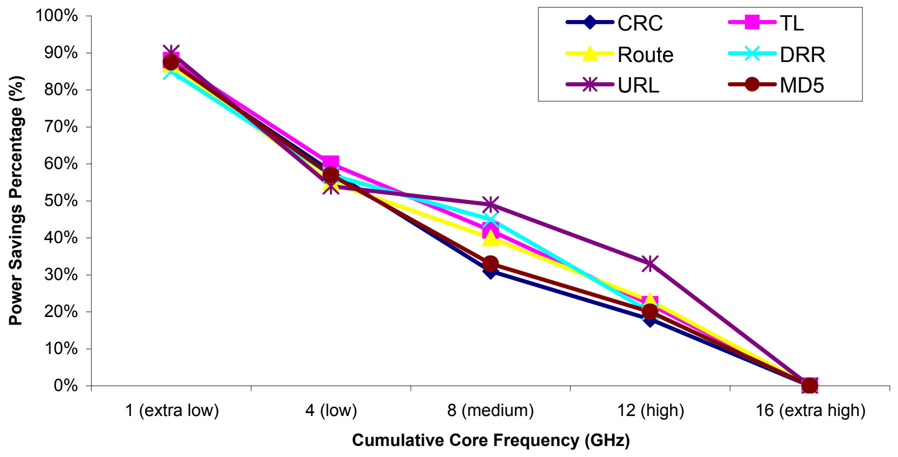

Figure 5.10: Power savings percentage under different workloads.

In addition, we find our scheme especially useful when the traffic is light, e.g., in medium, low and extra low cases. This is because under light load we have more potential to apply per-core DVFS, power gating and power migration to achieve power savings.

Figure 5.11 shows the average power consumption for different applications comparing our scheme with three other schemes for the five different synthetic workloads. We observe that our scheme performs the best across all applications with an average of 35.2% power savings over C-DVFS, 24.3% over PG and 10.5% over C-Hybrid.

C-DVFS performs the worst due to significant over-provisioning and excessive static power consumption, as it always keeps all the cores actively running. PG improves upon C-DVFS by turning off unnecessary cores to mitigate over-provisioning and save static power. However, without frequency scaling, it still suffers from excessive power consumption during extra low traffic. C-Hybrid outperforms both C-DVFS and PG due to its more flexible power management scheme using both chip-wide DVFS and power gating. But, C-Hybrid fails to achieve the best power savings because it does not consider static power or support more advanced per-core DVFS and power migration. Our scheme outwins all other schemes by providing the optimal system operating level and dynamically changing the per-core frequency configuration.

In addition, to emphasize the importance of power migration, we also experiment with our scheme without migration. Our scheme with power migration achieves an additional 2.5W reduction of power on average over our scheme without power migration. This highlights the advantage of including power migration in our power optimization scheme. In particular, when the traffic rate is extra low, low and medium, power migration can significantly reduce peak core temperature and hence effectively reduce static power consumption. We will present our study of processor thermal behavior in Section 5.3.5.

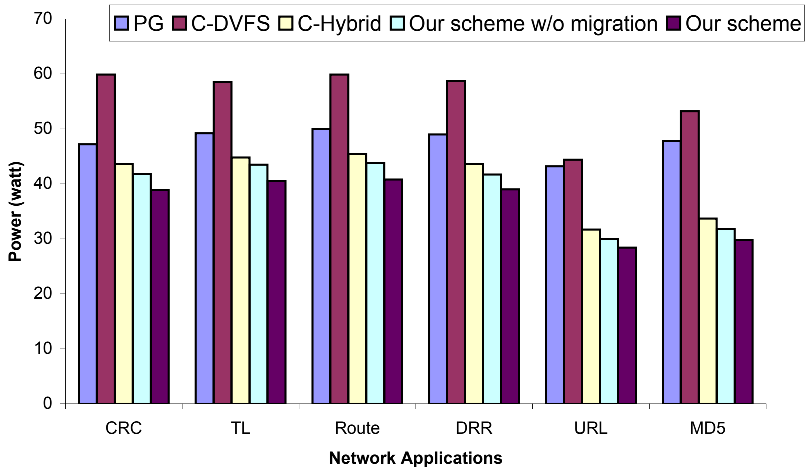

Figure 5.11: Power consumption comparison with three other schemes.

5.3.3 Energy Savings

Figure 5.12 shows normalized energy consumption compared to a native system for different schemes using the real-world workload as input (24-hour traffic as shown in

Figure 1.1). We observe that all four scheme are able to achieve energy savings, ranging from the least energy consumption of 0.45 for the URL application with our scheme, to the most energy consumption of 0.71 for the Route and DRR application with C-DVFS scheme. However, upon averaging out all six applications over the 24-hour period, we still find our scheme outperforms PG, C-DVFS and C-Hybrid by 22.0%, 19.1% and 8.4%, respectively. The poor performance of PG and C-DVFS is due to the following two reasons: 1) PG always lets the cores run at full speed without frequency scaling, and 2) C-DVFS always has all 8 cores actively running without power gating. Compared to PG and C-DVFS, C-Hybrid improves the energy performance by combining both chip-wide DVFS and power gating. However, because C-Hybrid is unable to provide the optimal system operating level and ignores static power, it often fails to achieve the best energy savings.

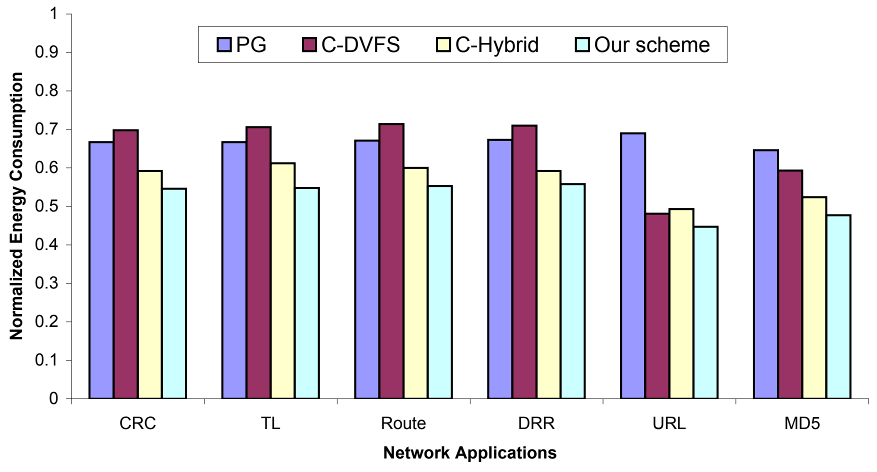

Figure 5.12: Normalized energy consumption with three other schemes.

Table 5.4 shows the snapshot of per-core frequency configuration and the provided cumulative core frequency at time 17:00 for different schemes. From this table, we can visualize the exact system operating level and frequency combinations at that moment. As the required cumulative core frequency is only 8.5GHz based on Figure 1.1, we see clearly that Native, PG and C-DVFS suffer from substantial over-provisioning by providing "more-than-needed" cumulative core frequency. Both C-Hybrid and our scheme provide "just-enough" cumulative core frequency. However, C-Hybrid consumes much higher dynamic power without a proper frequency assignment. Our scheme, on the other hand, maintains a good balance between dynamic and static power with the optimal number of active cores and appropriate per-core frequency configuration.

Table 5.4: Per-core frequency configuration snapshot at 17:00.

	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	F
Native	2	2	2	2	2	2	2	2	16
PG	2	2	2	2	2	0	0	0	10
C-DVFS	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	9.6
C-Hybrid	1.7	1.7	1.7	1.7	1.7	0	0	0	8.5
Our scheme	1.4	1.2	1.2	1.2	1.2	1.2	1.2	0	8.6

5.3.4 Reconfiguration Overhead

As our scheme involves periodic reconfiguration of core status, we also investigate the overhead associated with our scheme. First, we individually measure the overhead for DVFS and power gating. For DVFS, it takes 0.008 seconds to change the per-core frequency level. For power gating, it takes 0.11 seconds to turn off a core and 0.08 seconds to turn on a core. In addition, we notice that in power gating, turning off a core does not add overhead as that power-gated core will be inactive in the next second. Also, not every core changes status every second. Therefore, we measure the average per-core reconfiguration

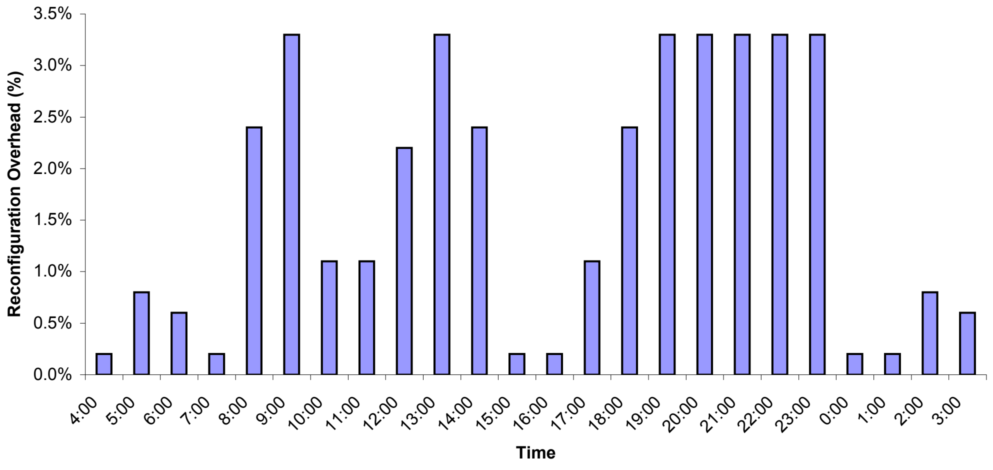

Figure 5.13: Reconfiguration overhead versus time in our scheme.

overhead over the 24-hour traffic periodic at each hour as shown in Figure 5.13. This figure shows the result for TL, Route and DRR, which have the same system operating level with the same K value. The other three applications, CRC, URL and MD5 have very similar performance. Every second, we count the invoked number of DVFS and power gating for all the cores and divide the aggregated total overhead by 8. From this figure, we observe that the overhead ranges between 0.2% and 3.3% with an average of 1.7%, which is negligible. It is also easy to see that during low traffic hours (i.e., 8:00-14:00 and 17:00-23:00), the overhead is higher due to more frequent power migrations.

5.3.5 Thermal Behavior

Finally, to demonstrate the effectiveness of power migration in reducing peak core temperature, we conduct in-depth study of processor thermal behavior when running network applications with different schemes. We use IPMItool utility to read processor thermal sensor and obtain the temperature for each processor every second. As there is no per-core thermal sensor available in our system, we approximately regard the processor temperature as the peak core temperature in that processor. Figure 5.14 shows the maximal temperature increase at extra low (XL), low (L) and medium (M) traffic rate, where power migration is playing a significant role. Since all the starting temperatures are the same, we can see our scheme has the minimal peak core temperature in all cases. In this figure, DVFS represents DVFS-based schemes, including both C-DVFS and C-Hybrid, as they have the same thermal behavior.

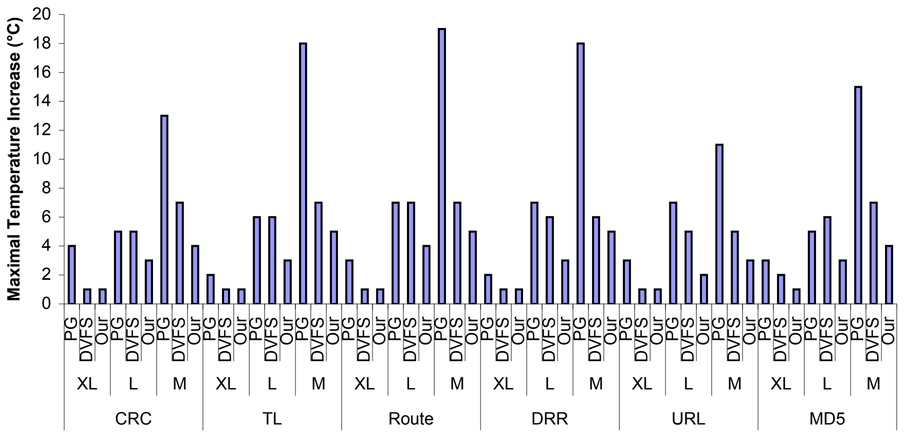

Figure 5.14: Temperature performance under different workloads.

More specifically, we see that PG causes the highest temperature increase (up to 19 degree for the Route application in medium traffic rate) because it lets all the cores run at the maximal frequency all the time. DVFS-based schemes, on the other hand, achieve better thermal behavior with frequency scaling, especially in extra low traffic rate. However, it still suffers from 1 degree to 3 degree higher peak core temperature compared to our scheme when the traffic rate is low or medium, as it always stresses the same active cores. We observe that our scheme on average reduces peak core temperature by 6 degree compared to PG in all traffic rates and by 2 degree compared to DVFS in low and medium traffic rate. This observation clearly shows that our scheme is able to achieve thermal balancing and keep a lower peak core temperature through power migration.

Chapter 6

Predictive Model-Based Thermal-aware Scheduling

6.1 Introduction

As can be seen from the previous chapter, power and temperature are closely related to each other. On one hand, higher power consumption generates more heat and increases temperature. On the other hand, higher temperature leads to higher static power consumption and exacerbates the problem. Thus, in this chapter, we focus on thermal-aware scheduling of network applications on multicore architecture.

we first establish a novel predictive thermal model for generic periodic tasks running on a single core. Our model is based on verification through both the HotSpot simulator [85] and a real Linux machine with six Netbench applications [68]. Simulation is used to verify the temperature rise and fall during a packet execution because the time is too small to be measured through an on-chip thermal sensor. Measurement is used for verification of the behavior over long term. Both simulation results and machine measurement show that our model can accurately predict the core temperature. Then, we propose an online model update to respond to incidental errors in practical use. Finally, by combining the thermal model and the online update, we design, implement and evaluate a predictive model-based thermal-aware scheduler for network applications on multicore architecture. To the best of our knowledge, we are the first to integrate thermal awareness into scheduler design for periodic tasks based on a predictive thermal model.

With regard to the online model update, we use on-chip thermal sensors to do model sanity check periodically. We compare the sensor temperature with model temperature and if the difference is larger than certain predefined threshold, we appropriately update model parameters “on-the-fly”. Lastly, our thermal-aware scheduler strives to maximize throughput and achieve thermal balancing across all cores without violating thermal constraint by combining task migration and Stop&Go techniques in a cost-effective and cache-aware fashion. It always places the current task to an idle core with the lowest temperature, which not only guarantees thermal balancing but also contributes to high throughput due to no waiting time. To avoid thermal emergency, the scheduler will make an appropriate decision based on cache-aware cost analysis, either migrating the task to another core with lower temperature (task migration) or letting the hot core cool down and resume it on the same core (Stop&Go).

We implement our thermal-aware scheduler along with four existing techniques (a load balancing scheduling scheme called Least Load First (LLF), LLF with thermal-awareness (LLF+), and two thermal-aware scheduling schemes, Greedy [92] and Coldest Core First (CCF) [97]) on an Intel server with two Quad-Core Xeon E5335 processors. Through extensive experiment, we observe that our scheduler achieves higher throughput, lower temperature and better thermal balancing with negligible scheduling overhead and no thermal constraint violation.

Our contribution can be summarized as follows.

- We establish a predictive thermal model for generic periodic tasks, which can characterize both temperature rise and fall and dynamically derive the core temperature quickly and accurately.
- We propose an online model update using on-chip thermal sensors, which can correct incidental errors by adjusting model parameters "on-the-fly".
- We design a predictive model-based thermal-aware scheduler for network applications on multicore architecture, which combines both task migration and Stop&Go techniques in a cost-effective and cache-aware fashion.
- We verify our model and evaluate the scheduler through extensive simulations and machine measurements using real benchmark applications.

The rest of this chapter is organized as follows. Section 6.2 introduces background information. In Section 6.3, we build the periodic task thermal model and describe the online model update. Then, we propose our predictive model-based thermal-aware scheduler in Section 6.4. We present experiment results and performance evaluation in Section 6.5.

6.2 Preliminaries

6.2.1 System Architecture

Figure 6.1 shows an overview of the targeted system architecture consisting of multiple cores with local queues of tasks. Incoming packets are first stored in a global FIFO queue. The execution time of a packet can be obtained based on its application type and packet size, but the packet inter-arrival time can vary. The scenario represents a periodic task, where the execution takes place when a packet arrives. The scheduler consists of two functional modules, namely *packet dispatcher* and *thermal manager*. In each scheduling cycle, the *packet dispatcher* fetches the next available packet from the global queue, makes the scheduling decision, and then dispatches the packet into the appropriate core. Each core runs a packet processing thread, which iteratively fetches a packet from its local queue and executes the task.

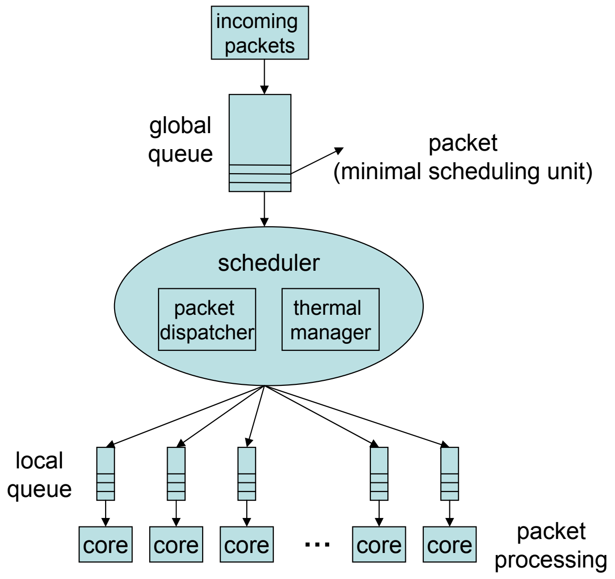

Figure 6.1: Overview of the system architecture.

The *thermal manager* monitors the thermal behavior of each core. We assume each core has two states: *busy* and *idle*. In the *busy* state, the core is executing a packet and the temperature will rise unless it is saturated; whereas in the *idle* state, the core is idle and the temperature will fall unless it has already reached the ambient temperature. The two states interleave with each other and cause the temperature variation. Based on our predictive thermal model, the *thermal manager* knows if a core will reach its critical temperature. If such thermal emergency happens, the *thermal manager* will pause the hot core and make an appropriate decision, either migrating the remaining task to another core with lower temperature (task migration) or letting the hot core cool down and resume its execution on the same core after certain amount of cool-down interval (Stop&Go).

6.2.2 Core/Cache Topology

As task migration involves data transfer between two cores and potential loss of cached states, the communication cost and cache performance are major concerns in overhead analysis and destination core selection. For multicore architecture, the inter-core communication cost is heterogenous due to hierarchical core/cache topology (e.g., cores sharing the last level cache have much less communication time compared to cores located on different sockets). As a result, the core/cache topology factor has to be carefully considered, since cache-awareness is critical in avoiding expensive communication overhead and unnecessary cache misses.

For example, Figure 6.2 shows a typical multicore architecture and its core/cache topology. The Intel Xeon E5335 has a tree-based hierarchy. From bottom up, a group of two cores share the same L2 cache. Two of these groups (4 cores) share the same core socket (S0 and S1). And the processor contains 2 such sockets that communicate through main memory. It is obvious to see that the communication cost between two cores is vastly different as illustrated by the arrow thickness ($C_{(C_0,C_2)}<C_{(C_0,C_6)}<C_{(C_0,C_1)}$).

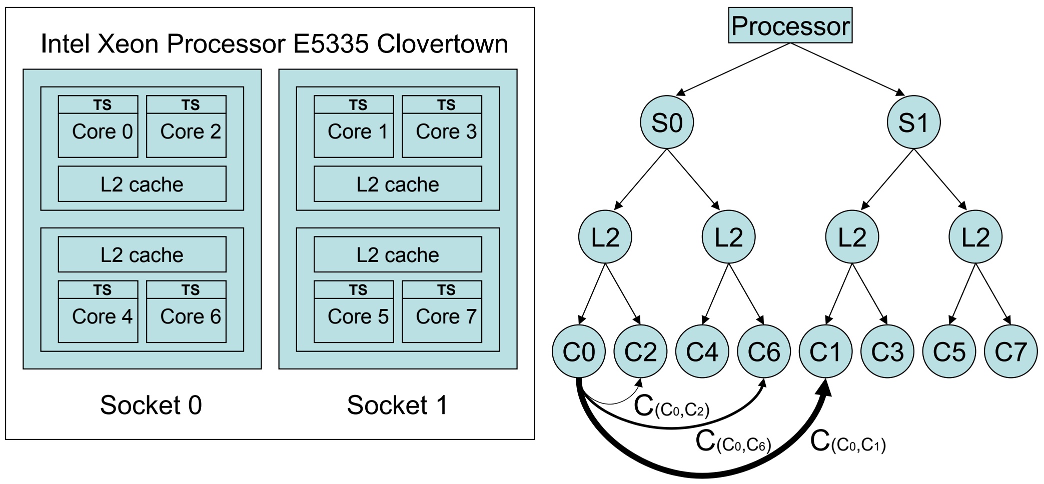

Figure 6.2: A tree-based core/cache topology on multicore architecture.

To achieve cache-awareness in task migration, we differentiate the migration overhead according to the communication time between two cores, which is further dependent on their inter-core relationship. For instance, in Figure 6.2, we can see three different inter-core relationships as indicated by the three arrows. We use U_{tt} to denote the time to transfer a unit data between two closest cores, and $C_{(C_{j_1},C_{j_2})}$ to represent the normalized communication cost between core C_{j_1} and core C_{j_2} for a unit data. If C_{j_1} and C_{j_2} are the two closest cores, $C_{(C_{j_1},C_{j_2})} = 1$. Otherwise, $C_{(C_{j_1},C_{j_2})}$ is the ratio defined as the real communication time between C_{j_1} and C_{j_2} divided by U_{tt}, which is always greater than 1.

6.2.3 Notations

We consider a multicore system with M homogeneous cores, $\{C_1, C_2, ..., C_M\}$; a set of N periodic tasks, $\{\tau_1, \tau_2, ..., \tau_N\}$; and K application types, $\{\alpha_1, \alpha_2, ..., \alpha_K\}$. The communication cost between two cores is heterogenous due to hierarchical core/cache topology. Each task $\tau_i (0 < i \leq N)$ has its own timing and temperature information. The notations regarding the system and timing/temperature-related properties of a given task $\tau_i (0 < i \leq N)$, application type $\alpha_k (0 < k \leq K)$ and core $C_j (0 < j \leq M)$ are defined as follows:

- Core state ($State_j^t$): the state of core C_j at time t, either *busy* or *idle*.
- Core temperature (T_j^t): the temperature of core C_j at time t. T_j^0 represents the initial value.
- Scheduled core ($Core_i$): the core where task τ_i is scheduled.
- Arrival time (A_i): the time when task τ_i arrives.
- Starting time (S_i): the time when task τ_i starts.
- Waiting time (W_i): the time difference between A_i and S_i. $S_i = A_i + W_i$.
- Execution time (E_i): the elapsed time to execute task τ_i.
- Overhead time (O_i): the wasted time of task τ_i during its execution due to migration or cooling.
- Finish time (F_i): the time when task τ_i finishes. $F_i = S_i + E_i + O_i$.

- Remaining time (R_i): the remaining execution time of task τ_i when the core reaches $T_{critical}$.
- Transferred data size (D_{size}): the size of transferred data in task migration.
- Unit transfer time (U_{tt}): the time to transfer a unit data between two closest cores.
- Normalized communication time ($C_{(C_{j_1},C_{j_2})}$): the normalized communication cost to transfer a unit data from core C_{j_1} to core C_{j_2}.
- Task migration overhead ($M_{(C_{j_1},C_{j_2})}$): the time overhead when task migrates from core C_{j_1} to core C_{j_2}.
- Steady-state temperature (T_{ss}^{k}): The temperature saturation point for application type α_k.
- Ambient temperature (T_{am}): The lowest core temperature when there is no task running.
- Critical temperature ($T_{critical}$): The given temperature constraint that should never be violated.
- Safe temperature (T_{safe}): The highest temperature that allows a core to finish its remaining task without reaching $T_{critical}$ again.
- Cooling time (L_i): The time overhead for the core to cool from $T_{critical}$ to T_{safe}.
- Trigger temperature ($T_{trigger}$): Thermal management is triggered when the core temperature rises to $T_{trigger}$.

6.2.4 Problem Statement

We consider two objectives when designing our thermal-aware scheduler. First, we aim at maximizing throughput measured by the number of packets processed per unit time. Second, we strive to achieve thermal balancing across all cores and guarantee that the given thermal constraint will be satisfied at any time. As discussed earlier, only a proactive approach based on predictive thermal model can potentially achieve these two objectives. Therefore, we define the problem statement as follows: *For network applications running on multicore architecture, how can one model the thermal behavior of periodic tasks and apply the predictive thermal model to appropriately schedule packets, so that system throughput is maximized, thermal balancing is achieved and temperature constraint is satisfied?*

6.3 Periodic Task Thermal Model

The periodic task execution is a continuous flow consisting of interleaved temperature rising and falling phases. The phase change corresponds to the change of CPU states between *busy* and *idle*. Figure 6.3 sketches the thermal profile of the core as the periodic task executes. The temperature increases when the task is executing and it decreases during idle periods. Figure 6.3 only shows the temperature variation for the first three tasks with their starting time and finish time.

Specifically, at time S_0, the core starts out at temperature T_0. In the interval [S_0,S_1], the core temperature increases to T_0' and cools down to T_1. In the interval [S_1,S_2], the temperature rises to T_1' when the second task finishes and then cools down to T_2. More generally, suppose at time S_k, the core is at temperature T_k. Then, the temperature rises to T'_k when the k^{th} task finishes and it cools down to T_{k+1} by time S_{k+1}. Ideally, if the *busy* time and *idle* time are constant, we can easily show that $T'_0 \leq T'_1 \leq T'_2 \leq \cdots \leq T'_k \leq \cdots$ and $T_0 \leq T_1 \leq T_2 \leq \cdots \leq T_k \leq \cdots$. We will prove that the two non-decreasing sequences of temperatures eventually converge in this case.

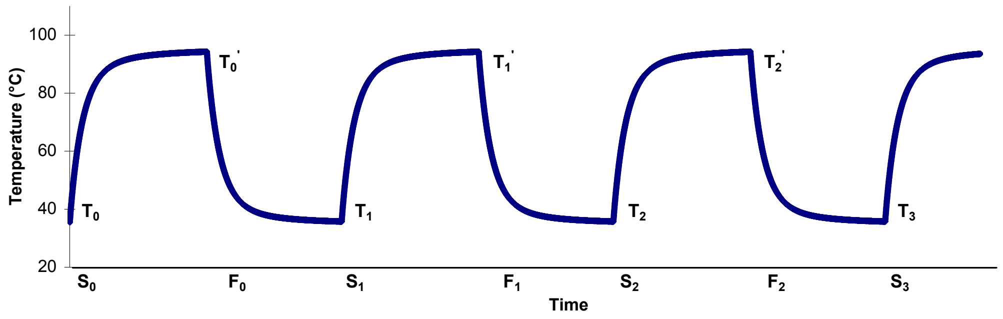

Figure 6.3: Temperature variation as the periodic task executes.

From Figure 6.3, we can apply a recursive approach to dynamically derive the current temperature from the most recent temperature history (e.g., the temperature when the previous task finishes) given the timing information. Hence, in this section, we first model the rising and falling edges separately. Then, we derive the recursive model for periodic tasks. Finally, we discuss model thermal properties and analyze its time complexity and accuracy.

6.3.1 Single Task Thermal Model

Temperature Rising

When a core is in *busy* state, its temperature will rise and will eventually stabilize at a thermal saturation point T_{ss}. At this point, the heat generation rate will equal to the heat dissipation rate, thus thermal equilibrium is achieved. The classic heat transfer equations model T_{ss} in a system with heat sources [52]. It has been proved that the temperature changes exponentially to T_{ss} starting from any initial temperature. In another word, the rate of temperature change is proportional to the difference between the current temperature and T_{ss}. In this chapter, we assume T_{ss} of a certain application type is known from offline profiling without thermal management. Let $T(t)$ represent the temperature at time t and let T_{init} be the temperature when a task starts execution ($T(0) = T_{init}$). According to [52], we have:

$$\frac{dT(t)}{dt} = \beta \times (T_{ss} - T(t)) \tag{6.1}$$

By solving Equation 6.1 with $T(0) = T_{init}$ and $T(\infty) = T_{ss}$, we can derive the following equation:

$$T(t) = T_{ss} - (T_{ss} - T_{init}) \times e^{-\beta t} \tag{6.2}$$

β is a processor/application-specific constant, whose value can be determined by fitting temperature rising curve using *least square method.* With Equation 6.2, we can predict the rising temperature after time t when a task gets started given its initial temperature.

Temperature Falling

When a core is in *idle* state, its temperature will fall and will finally reach ambient temperature T_{am}. At this point, the core enters thermal equilibrium state due to no more heat loss from itself to its surroundings. This cooling process can be described by Newton's law of cooling [23], which states that the temperature change rate of an object is proportional to the difference between its own temperature and the ambient temperature. Newton's law makes a statement about an instantaneous rate of change of the temperature, which can be translated into a differential equation similar to Equation 6.1. Let T_{init} be the temperature when a task just finishes execution ($T(0) = T_{init}$). Equation 6.3 represents the temperature change rate according to Newton's cooling law.

$$\frac{dT(t)}{dt} = -\rho \times (T(t) - T_{am}) \tag{6.3}$$

By solving Equation 6.3 with $T(0) = T_{init}$ and $T(\infty) = T_{am}$, we can derive the following equation:

$$T(t) = T_{am} + (T_{init} - T_{am}) \times e^{-\rho t} \tag{6.4}$$

ρ is a processor specific constant irrespective of running applications. We can use the same *least square method* as in temperature rising phase to obtain ρ. With Equation 6.4, we can predict the temperature after time t when a task finishes given its initial temperature.

6.3.2 Periodic Task Thermal Model

In order to represent generic periodic tasks, we allow both *busy* time and *idle* time to be arbitrary. We consider two scenarios in deriving the thermal model for periodic tasks. On one hand, if the core is in *idle* state at time t, the core temperature is only determined by the temperature when the previous task finishes and the elapsed time since then. It is because the most recent temperature history already reflects the effect of all past task executions and temperature variations. Thus, we only keep record of the temperature when the previous task finishes. Suppose the previous task is τ_{n-1} and its finish time is F_{n-1}. Then, based on Equation 6.4 we have the following:

$$T(t) = T_{am} + (T^{F_{n-1}} - T_{am}) \times e^{-\rho(t-F_{n-1})} \tag{6.5}$$

On the other hand, if the core is in *busy* state at time t, the core temperature is dependent on the temperature when the current task started and the elapsed time since then. Suppose the current task is τ_n and its starting time is S_n. Based on Equation 6.2 we have:

$$T(t) = T_{ss} - (T_{ss} - T^{S_n}) \times e^{-\beta(t-S_n)} \tag{6.6}$$

As we only keep record of $T^{F_{n-1}}$, the temperature when the previous task finished, we can use Equation 6.5 to derive T^{S_n} from $T^{F_{n-1}}$ as shown in Equation 6.7.

$$T^{S_n} = T_{am} + (T^{F_{n-1}} - T_{am}) \times e^{-\rho(S_n - F_{n-1})} \tag{6.7}$$

Now we can recursively derive the core temperature at any time from previous equations. We show how to get T^{F_n} from T^{S_n} as another example[1]:

$$T^{F_n} = T_{ss} - (T_{ss} - T^{S_n}) \times e^{-\beta(F_n - S_n)} \tag{6.8}$$

Based on previous equations we present our periodic task thermal model in Equation 6.9. Suppose the first task starts out at time 0 and at temperature T_{am}.

$$T(t) = \begin{cases} T_{am} + (T^{F_{n-1}} - T_{am}) \times e^{-\rho(t - F_{n-1})}, & \textit{idle} \\ T_{ss} - (T_{ss} - T^{S_n}) \times e^{-\beta(t - S_n)}, & \textit{busy} \end{cases} \tag{6.9}$$

$$\text{where } \begin{cases} T^{S_n} = T_{am} + (T^{F_{n-1}} - T_{am}) \times e^{-\rho(S_n - F_{n-1})} \\ T^{F_1} = T_{ss} - (T_{ss} - T_{am}) \times e^{-\beta F_1} \end{cases}$$

[1]If thermal constraint is enforced, T^{F_n} will be $T_{critical}$ in case of thermal emergency.

6.3.3 Model Thermal Properties

Upper/Lower Bound

For generic periodic tasks (i.e., arbitrary *busy* time and *idle* time), the temperature variation is upper-bounded by T_{ss} and lower-bounded by T_{am} at any time.

proof: Given any time t during periodic task execution, it is either in temperature rising phase or temperature falling phase. Equation 6.2 characterizes the temperature rising phase. Given this exponential equation, the temperature $T(t)$ will always be smaller than T_{ss}. Only when t is infinity will $T(t)$ be equal to the upper bound T_{ss} as per Equation 6.10.

$$T(t) = T_{ss} - (T_{ss} - T_{init}) \times e^{-\beta t} \leq T_{ss} \tag{6.10}$$

$$lim_{t\to\infty} T(t) = T_{ss}$$

Equation 6.4 models the thermal behavior of temperature falling phase. Similarly, whatever the starting temperature T_{init} is, the temperature $T(t)$ will always be larger than T_{am} while decreasing. In this case, only when t equals to infinity will $T(t)$ reach the lower bound T_{am} as shown in Equation 6.11.

$$T(t) = T_{am} + (T_{init} - T_{am}) \times e^{-\rho t} \geq T_{am} \tag{6.11}$$

$$lim_{t\to\infty} T(t) = T_{am}$$

Thermal Equilibrium

For ideal periodic tasks (i.e., constant *busy* time and *idle* time), their temperature variation will eventually converge to thermal equilibrium between T_{min} and T_{max}. In each period, the temperature increases from T_{min} to T_{max}, and then decreases from T_{max} to T_{min}.

proof: First, we know $T_{am} < T_{min} < T_{max} < T_{ss}$ from the previous proof. Second, from the definition of ideal periodic tasks, we know the task inter-arrival time is constant ($S_1 - S_0 = S_2 - S_1 = \cdots = S_{k+1} - S_k = \cdots$) and the execution time of each task is also constant ($E_0 = E_1 = E_2 = \cdots = E_k = \cdots$). Third, we assume the task execution starts from T_{am} without loss of generality. Relating to Figure 6.3, we have $T_0 = T_{am}$. It is also true that $T'_0 \leq T'_1 \leq T'_2 \leq \cdots \leq T'_k \leq \cdots$ and $T_0 \leq T_1 \leq T_2 \leq \cdots \leq T_k \leq \cdots$, since both *busy* time and *idle* time are invariants. Now we demonstrate the two non-decreasing sequences of temperatures converge to T_{max} and T_{min}, respectively.

We let Δ_k represent the temperature increase while the k^{th} task is executing ($\Delta_k = T'_k - T_k$), and Δ'_k represent the temperature decrease after the k^{th} task finishes ($\Delta'_k = T'_k - T_{k+1}$). Based on Equation 6.2, from the property of exponential equation[2], we know that given the same execution time, if the starting temperature is higher, the temperature increase will be smaller. Thus, from $T_0 \leq T_1 \leq T_2 \leq \cdots \leq T_k \leq \cdots$, we have the following: $\Delta_0 \geq \Delta_1 \geq \Delta_2 \geq \cdots \geq \Delta_k \geq \cdots$. Similarly, based on Equation 6.4, we know that given the same cooling time, if the starting temperature is higher, the temperature decrease will be bigger. Thus, from $T'_0 \leq T'_1 \leq T'_2 \leq \cdots \leq T'_k \leq \cdots$, we have the following:

[2]The slope of the curve decreases over time.

$\Delta'_0 \leq \Delta'_1 \leq \Delta'_2 \leq \cdots \leq \Delta'_k \leq \cdots$. As a result, the non-increasing sequence of Δ_k will eventually merge with the non-decreasing sequence of Δ'_k at the equilibrium point, since their initial values satisfy $\Delta_0 \geq \Delta'_0$ (because $T_0 \leq T_1$, we have $\Delta_0 = T'_0 - T_0 \geq T'_0 - T_1 = \Delta'_0$). From that point onward, both Δ_k and Δ'_k will be equal to $T_{max} - T_{min}$, and we have $lim_{k\to\infty} T_k = T_{min}$ and $lim_{k\to\infty} T'_k = T_{max}$ in thermal equilibrium state.

As a matter of fact, we can obtain the value of T_{min} and T_{max} by solving Equation 6.12 that characterizes the temperature variation in thermal equilibrium state. Suppose the *busy* time constant and *idle* time constant are t_{busy} and t_{idle}, respectively.

$$\begin{cases} T_{min} = T_{am} + (T_{max} - T_{am}) \times e^{-\rho t_{idle}}, & \textit{idle} \text{ state} \\ T_{max} = T_{ss} - (T_{ss} - T_{min}) \times e^{-\beta t_{busy}}, & \textit{busy} \text{ state} \end{cases} \tag{6.12}$$

$$\Rightarrow \begin{cases} T_{min} = \frac{T_{am}(1-P)+T_{ss}(1-Q)P}{1-PQ} \\ T_{max} = \frac{T_{ss}(1-Q)+T_{am}(1-P)Q}{1-PQ} \end{cases} \text{where} \begin{cases} P = e^{-\rho t_{idle}} \\ Q = e^{-\beta t_{busy}} \end{cases}$$

6.3.4 Time Complexity and Accuracy

The two criteria of an online thermal model are fast response time and accuracy. We expect to apply our model to network applications with periodic packet processing in the magnitude of microseconds. If the model takes very long to compute the temperature, it will simply fail to function. Our model, as presented in Equation 6.9, however, has very fast response time. It only takes timing information and one history temperature to derive the current temperature from an exponential equation. Experiments on a real Linux machine with 2GHz frequency show that our model takes only $0.1\mu s$ to compute the temperature.

The accuracy of our model is also guaranteed by comparing with both well-known simulator and real Linux machine. Because the thermal behavior of the packet (Figure 6.3) is valid at micro second level, it is impossible to get measurement results at that low granularity. The current temperature sensors measure the core temperatures only at one second interval. Thus, we have verified our thermal model by running six network applications chosen from NetBench on HotSpot simulator with different packet traces. HotSpot simulator [85] is best known for its accuracy and credibility in architectural studies. Simulation results show that our model can closely match the temperature curve drawn by HotSpot simulator. In addition,we also validate the model by comparing it with actual measurements in a Linux box to show that the mathematical model adequately represents the thermal characteristics at a large time interval. Detailed results regarding model verification can be found in Section 6.5.

6.3.5 Online Model Update

Although our mathematical model can adequately represent the thermal characteristics of periodic tasks, its robustness and soundness can still be compromised due to incidental errors.

First, the model is vulnerable to unpredictable dynamics in practical online use, such as changing ambient temperature, abnormal application behavior, hardware malfunction, interference from other threads or heat dissipation from other cores. All these random events/conditions can result in model failure. Second, the model is error-prone by its recursive nature as the error may propagate at each calculation. From Equation 6.9 we know that our thermal model is based on four constant parameters (i.e., processor/appli-cation-specific parameters T_{ss} and β, processor-specific parameters T_{am} and ρ), timing information (i.e., t, F_{n-1} and S_n) and the most recent temperature history $T^{F_{n-1}}$. For the four constant parameters, their values will not change for a given application and core. This is because 1) all the applications consume the constant stable power as can be seen in Figure 6.5 without much fluctuation; and 2) the heat dissipation of a core in *idle* state only depends on its hardware physical properties, which will not change over time. Thus, there is no need to update them periodically. In addition, timing information (which is converted from CPU cycles or obtained through system call) can be also considered trustworthy. Thus, only the value of $T^{F_{n-1}}$, which is derived each time from the model itself, may not always be accurate. Given the recursive nature of our thermal model, it is possible for a faulty $T^{F_{n-1}}$ with tiny little error to cause catastrophe in the long run.

We propose an online model update as shown in Figure 6.4 to make the model more robust and sound. This update strategy can effectively correct incidental errors by appropriately adjusting the model parameter $T^{F_{n-1}}$ "on-the-fly".

More specifically, our update strategy relies on on-chip temperature sensors to do model sanity check (i.e., automatic calibration), as modern CPUs equipped with temperature sensors are capable of periodically reporting their own internal temperature. Figure 6.4 illustrates the framework of our online model update based on per-core temperature sensor. For our system (i.e., an Intel server with two Quad-Core Xeon E5335 processors), the temperature sensor sampling interval is 1 second. Thus, every second we feed the real core temperature to an online model adjustor. In the meantime, our thermal model also sends its predicted model temperature to the adjustor every second. Then, the adjustor compares the two values and if the difference is larger than certain predefined threshold, it will appropriately compute a new $T^{F_{n-1}}$, the most recent temperature history, to replace the old value. By this update, all the future temperatures derived from the thermal model will be trustworthy based on the updated $T^{F_{n-1}}$.

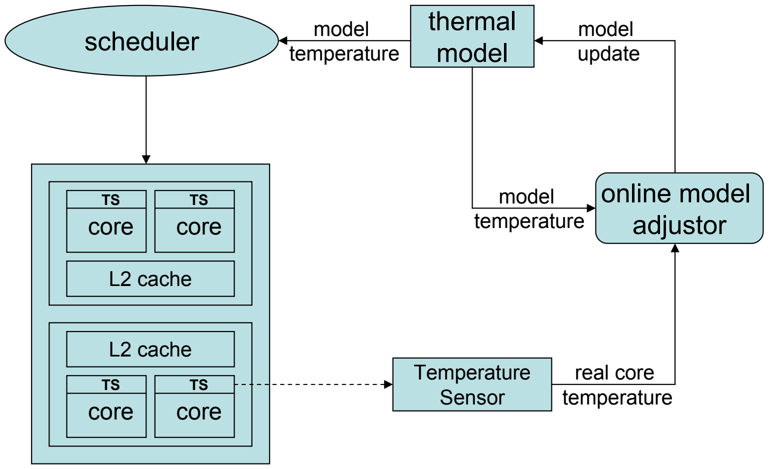

Figure 6.4: Online model update in practice.

$T^{F_{n-1}}$ is updated as follows. Suppose the update point is time t and the temperature derived from the model is $T(t)$. First, we use the real core temperature to replace $T(t)$ in Equation 6.9 using the same equation as the original $T(t)$ is derived depending on the core status at time t. Second, we regard $T^{F_{n-1}}$ as an unknown variable and solve that equation to obtain the updated $T^{F_{n-1}}$. Equation 6.13 shows the calculation of updated $T^{F_{n-1}}$ in both *idle* and *busy* states. In Section 6.5 we will show the implementation details of the proposed update strategy and highlight its advantage in improving model accuracy through real machine experiments.

$$T^{F_{n-1}} = \begin{cases} T_{am} + \frac{T(t)-T_{am}}{e^{-\rho(t-F_{n-1})}}, & \textit{idle} \text{ state} \\ T_{am} + \frac{T_{ss}-\frac{T_{ss}-T(t)}{e^{-\beta(t-S_n)}}-T_{am}}{e^{-\rho(S_n-F_{n-1})}}, & \textit{busy} \text{ state} \end{cases} \tag{6.13}$$

6.4 Predictive Thermal-aware Scheduler

Our thermal management algorithm differs from previous work in the following two ways: 1) algorithms that handle periodic tasks do not use time-based predictive thermal model. Instead, they either do not have thermal model (i.e., reactive approaches) [35, 88] or rely on power estimation from history data [92, 93]; 2) algorithms that use time-based predictive thermal model only handle single task execution, whose temperature will simply rise to saturation point and then stabilize. They can not be extended to handle periodic tasks that involve both temperature rise and fall because they do not model the falling phase [96, 97].

6.4.1 Thermal Management Techniques

We adopt two thermal management techniques, task migration and Stop&Go, to guarantee that the thermal constraint is never violated.

Task migration means migrating current task from an overheated core to a cooler core. By doing so, the hot core will switch to the *idle* state and its temperature will cool down. The new core will continue executing the remaining task without much delay. Hence, task migration can effectively avoid thermal emergency with little throughput degradation. There are two conditions that have to be satisfied in task migration. First, the core running the current task will reach $T_{critical}$ before it finishes. This information can be obtained from our thermal model. Second, there exists at least one idle core where the task can be migrated. To retrieve this information, we can refer to the core state ($State_j^t$). Other issues concerning task migration include its cost analysis (e.g., data transfer) and the selection of the best destination core. We will address them shortly as they are closely related to core/cache topology of the underlying architecture.

Stop&Go refers to temporarily suspending the running task, letting the hot core cool down for some time and then resuming its execution. In Stop&Go, when to suspend the task and when to resume its execution are two design choices.

On one hand, as the core thermal behavior is modeled by exponential equations, we know the temperature decreases faster when the initial temperature is higher in the falling phase. This property is also self-explanatory due to the fact that the temperature change rate is proportional to the difference between the current temperature and T_{am}. Hence, we should first heat up the core to $T_{critical}$, the highest possible temperature, and let it cool down to determine the first design choice. This strategy will result in larger temperature decrease given the same cool-down interval and contribute to better thermal and throughput performance.

If the cool-down interval is too small, the core may not have enough time to cool down. In this case, the core will most likely to reach $T_{critical}$ again before finishing the remaining task, triggering another Stop&Go with extra system overhead. On the contrary, if the cool-down interval is too long, the task waiting time is significant, which may result in substantial throughput degradation. In our design, the core will resume task execution as soon as its temperature falls down to T_{safe}, the highest temperature that allows a core to finish its remaining task without reaching $T_{critical}$ again. In this way, throughput is maximized due to minimal waiting time and system overhead is minimized because Stop&Go will not be triggered more than once for every task. We will address how to derive T_{safe} and cool-down interval in the next section.

The major differences between task migration and Stop&Go lie in the following four aspects: 1) In task migration, the new core will immediately start executing the remaining task. In Stop&Go, the task will not be resumed unless the temperature falls to T_{safe}. 2) The overhead cost in Stop&Go is caused by cool-down interval, whereas the overhead cost in task migration is caused by data transfer and potential loss of cached states. 3) Stop&Go does not care about the status of other cores; whereas task migration requires at least one idle core. 4) A task will experience only one Stop&Go. But it may experience several task migrations if the migrated core reaches $T_{critical}$ again before the task finishes.

6.4.2 Overhead Quantification

For Stop&Go, the overhead time is equal to the cool-down interval (L_i), which depends on how long it takes for the core's temperature to decrease from $T_{critical}$ to T_{safe}. Although $T_{critical}$ is a predefined constant (i.e., given temperature constraint), T_{safe} depends on application type and the size of the remaining task. Thus, we need to first obtain T_{safe}, and then derive L_i.

To compute T_{safe}, we rely on the thermal model to backtrace its value. More specifically, we first obtain the remaining execution time of the task, R_i, which is the total execution time minus the time already spent on executing the task. Then, we use Equation 6.9 in the case of *busy* state to derive T_{safe} by substituting $T(t)$ with $T_{critical}$, T_{S_n} with T_{safe}, and $(t - S_n)$ with R_i. Equation 6.14 shows the derivation of T_{safe}.

$$\begin{aligned} T_{critical} &= T_{ss} - (T_{ss} - T_{safe}) \times e^{-\beta R_i} \\ T_{safe} &= T_{ss} - \frac{T_{ss} - T_{critical}}{e^{-\beta R_i}} \end{aligned} \tag{6.14}$$

After both T_{safe} and $T_{critical}$ are available, we can easily derive L_i by solving Equation 6.15, which is based on Equation 6.9 in the case of *idle* state.

$$\begin{aligned} T_{safe} &= T_{am} + (T_{critical} - T_{am}) \times e^{-\rho L_i} \\ L_i &= \frac{\ln \frac{T_{safe} - T_{am}}{T_{critical} - T_{am}}}{-\rho} \end{aligned} \tag{6.15}$$

With respect to task migration, we consider two cases. In the first case, one migration is enough to safely finish the task. In the second case, the migrated core will again reach $T_{critical}$ before the task finishes. In such a case, another thermal technique will be triggered, either task migration or Stop&Go. In fact, we know which case to happen from our thermal model before task migration occurs. In addition, if case two is to occur, we also know how soon the migrated core will reach $T_{critical}$ again and what is the remaining execution time. For the overhead quantification for the second case, we assume Stop&Go will be triggered in case of thermal emergency[3]. Thus, the associated overhead in task migration consists of two parts. The first part is the migration overhead, and the second part is potential Stop&Go overhead on the destination core, L_i. To compute the first part, we need to obtain D_{size}, which is equal to the packet size in network applications. Then, based on $C_{(C_{j_1},C_{j_2})}$ and U_{tt}, we can obtain the overhead time. To compute the second part, we can simply follow Equation 6.15. Finally, Equation 6.16 shows the calculation of $M_{(C_{j_1},C_{j_2})}$.

$$M_{(C_{j_1},C_{j_2})} = \begin{cases} C_{(C_{j_1},C_{j_2})} \cdot U_{tt} \cdot D_{size}, & \text{case 1} \\ C_{(C_{j_1},C_{j_2})} \cdot U_{tt} \cdot D_{size} + L_i, & \text{case 2} \end{cases} \tag{6.16}$$

6.4.3 Scheduling Algorithm

Our scheduling scheme has three objectives: throughput maximization, thermal balancing, and thermal constraint agreement. Our algorithm is based on a predictive thermal model for periodic tasks and two thermal management techniques, namely, task migration and Stop&Go. When a task τ_i with application type α_k is to be scheduled, the scheduling algorithm is described below.

1. Scan all the idle cores and select one with the lowest temperature. Schedule τ_i on this core ($Core_i$). If all the cores are busy, wait for the first idle core. (line 1-4)

[3]Because it is impossible to know the future core state information at this moment, Stop&Go is the only choice for fair comparison.

2. If the steady state temperature of task τ_i is less than the critical temperature $T_{critical}$, we are done. (line 5)

3. If the estimated temperature when task τ_i completes, $T_{Core_i}^{S_i+E_i}$, is less than the critical temperature $T_{critical}$, we are done. (line 6-7)

4. Otherwise, when the core temperature rises to $T_{trigger}$, calculate and compare the overhead for Stop&Go on $Core_i$ (Equation 6.15) and task migration on all other idle cores (Equation 6.16). Then, choose the core with the least overhead and take appropriate action. (line 8-15)

5. If Stop&Go is triggered, wait until $Core_i$ cools down to T_{safe} and resume its execution on the same core. If task migration is triggered, migrate τ_i to the destination core and continue its execution on the new core. Then go back to step 3 with updated timing variables.[4] (line 16-22)

Algorithm 2 presents the pseudocode. Although we only present the algorithm for one periodic task, the algorithm should continuously run for every incoming task following the same routine.

Algorithm 2 *Input: task τ_i with application type α_k*

```
1: if all cores are busy then
2:     $Core_i$ ← first idle core
3: else
4:     $Core_i$ ← idle core with the lowest temperature
```

[4]Task migration may still cause the new core to overheat again.

```
5:  **if** $T_{ss}^{k} \leq T_{critical}$ **then return**
6:  calculate $T_{Core_i}^{S_i+E_i}$
7:  **if** $T_{Core_i}^{S_i+E_i} \leq T_{critical}$ **then return**
8:  $t \leftarrow$ current time
9:  when $T_{Core_i}^{t} \geq T_{trigger}$, execute the following:
10: **for** each $C_j$ **do**    /* $C_j \in \{Core_i + all\ idle\ cores\}$ */
11:     **if** $C_j = Core_i$ **then**
12:         compute $T_{safe}$ and $L_i$ for $C_j$
13:     **else**
14:         compute $M_{(Core_i, C_j)}$ for $C_j$
15: choose the $C_j$ with the least overhead, $Core_j$
16: **if** $Core_j = Core_i$ **then**
17:     trigger Stop&Go
18:     **return**
19: **else**
20:     trigger task migration to $Core_j$
21:     update timing variables
22:     **goto** line 6
```

6.5 Experiment and Evaluation

6.5.1 Experiment Setup

We have done extensive experiments in 1) model verification and 2) performance evaluation. For model verification, we use both the HotSpot simulator [85] and a real Linux machine to run six network applications chosen from NetBench [68]. For performance evaluation, we implement our scheduling scheme along with four alternatives on an Intel server with two Quad-Core Xeon E5335 processors to compare their performance. The four compared schemes are described below.

- Least Load First (LLF): Schedule the packet to the least-loaded core. There is no thermal management.
- Thermal-aware Least Load First (LLF+): Schedule the packet to the least-loaded core. Use Stop&Go in case of thermal emergency based on thermal sensor readings.
- Greedy: Schedule the packet to the first non-overheated idle core [92].
- Coldest Core First (CCF): Schedule the packet to the coldest idle core [97].

The last two schemes are thermal-aware, and simple task migration is adopted in case of thermal emergency based on our thermal models[5] (always migrate to the coldest core). However, if there is no available idle core for migration, Stop&Go is used instead and the cooling time is fixed to be 1 second.

[5]As we are the first to propose predictive model-based thermal management for periodic tasks, there are no more suitable schemes other than the two we select for comparison.

Table 6.1 lists the six applications. We select two applications from each category (i.e., Micro-level, IP-level and Application-level) to make them representative. The packet trace is from NetBench with 10000 packets. The packet size ranges from 40 bytes to 1500 bytes with an average of 723 bytes. For thermal behavior purpose, we repeatedly execute each packet for 1000 times and manually insert packet inter-arrival time in three different ways assuming the network bandwidth is 1Gbps per core: Fixed Gap (FG), Random Distribution (RD), and Simulated Distribution (SD), as shown in Table 6.2. The routing table size for Route and DRR is 128, and we use the small_input file for URL.

Table 6.1: Six network applications from NetBench.

Name	Functionality	Category
CRC	CRC-32 checksum calculation	Micro level
TL	Radix-tree table lookup routine	Micro level
Route	IPv4 routing based on radix	IP level
DRR	Deficit-round robin scheduling	IP level
URL	URL-based switching	Application level
MD5	Message digest algorithm	Application level

Table 6.2: Packet trace patterns.

Pattern	Inter-arrival description
FG	$0.75ms$
RD	randomly chosen from 0 to $1.5ms$
SD	*idle* time is one half of *busy* time

We use HotSpot simulator to verify the model because the real temperature sensors can not measure in less than a second interval. The periodic tasks of packet executions vary in microsecond level. The HotSpot simulator needs the following input: 1) functional blocks of the simulated core, 2) floor plan of the simulated core and 3) the power trace during execution. For the first two inputs, we take the Alpha EV6 processor as a representative example, which is the default setting in HotSpot simulator as used in [84]. It is worth noting that our interest lies in the model verification other than investigating a particular core architecture. Thus, using a different core architecture will at most produce different parameters for our thermal model but it will not affect our model verification.

For the third input, to obtain the *busy* state power trace needed by HotSpot for each functional block, we use Wattch [22] with default settings to profile each application without inter-arrival time. Figure 6.5 shows the average total power consumption over time for all six applications running on Wattch, where each time unit represents 500 CPU cycles. From this figure, we see that after a short period of warm-up phase, every application has constant stable power consumption. Thus, we use their constant stable power consumption as *busy* state power in HotSpot as shown in Table 6.3. For *idle* state power, we assume zero power consumption as in power-gated case. Using other non-zero value will slow the temperature drop, which will only adjust the model parameter ρ but will also not affect the model verification. In addition, as HotSpot outputs the thermal profile for each functional block, we take the highest temperature among all blocks to represent the core temperature.

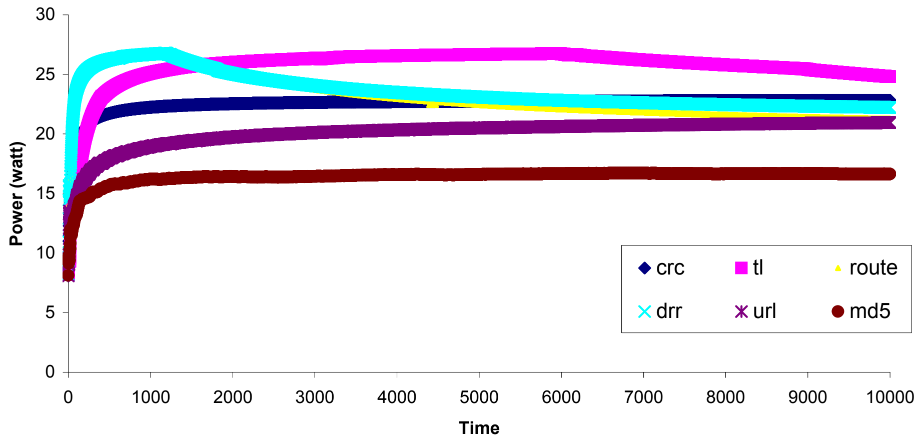

Figure 6.5: Power consumption versus time on Wattch.

For real machine measurement, we use one core of the Intel Xeon E5335 server with 2GHz core frequency to run benchmark applications. We bind the running thread on the fixed core and use lm-sensors [11] to read the core temperature every second. Lm-sensors is a Linux hardware monitoring tool, which can access Intel's Coretemp driver to get per-core temperature information. In addition, we also implement the online model adjustor to demonstrate the practicability and advantage of online model update. We let the update take place whenever there is 0.5 degree difference between model temperature and real temperature.

Table 6.3: Application power and time characteristics.

App.	Power(watt) (*busy* state)	Power(watt) (*idle* state)	$E_i(\mu s)$
CRC	22.8	0	0.008×pkt_size+0.3353
TL	24.8	0	0.8050
Route	21.3	0	1.7721
DRR	22.2	0	5.5501
URL	20.9	0	0.1369×pkt_size+75.499
MD5	16.6	0	0.0054×pkt_size+3.5355

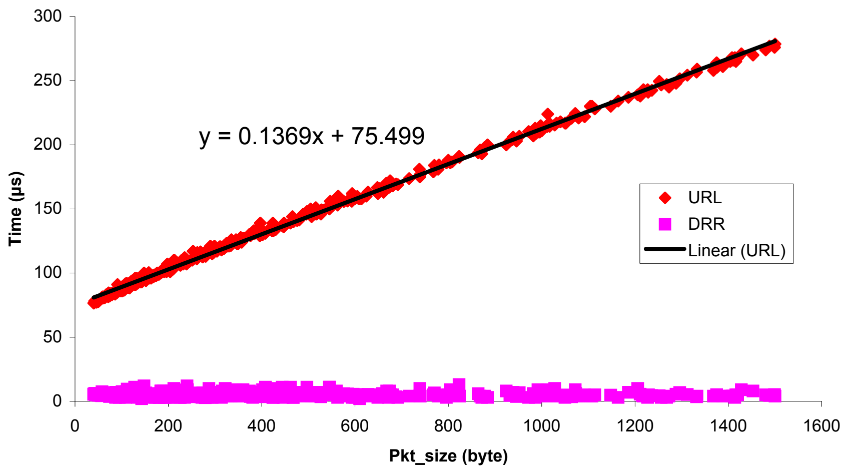

Figure 6.6: Execution time versus packet size for URL and DRR.

For performance evaluation, we implement our scheme with online model update and four others on an Intel server with two Quad-Core Xeon E5335 processors using the same settings as in real machine measurement. We create 8 parallel threads and bind each running thread to one core. We use sleep function calls in Linux to mimic the behavior of suspending a running task. All the model-related parameters, such as T_{am}, T_{ss}, β and ρ, are obtained offline from running benchmark applications on the real machine without enforcing thermal management as shown in Table 6.5. In particular, we also measure U_{tt} to be 7.3ns and $C_{(C_{j_1},C_{j_2})}$ to be 1, 1.7 and 3.1, respectively, depending on the inter-core distance between C_{j_1} and C_{j_2} as shown in Figure 6.2. The two thermal threshold parameters, $T_{critical}$ and $T_{trigger}$, are set to be 38 degree and 37.9 degree, respectively. In addition, to obtain the task execution time E_i for each packet, we profile the packet trace for each of the six applications and observe that E_i is either constant or linear to the packet size. The numerical results are listed in Table 6.3. For clarity we only show two typical applications (URL and DRR) in Figure 6.6 as an example of execution time versus packet size. As this chapter only focuses on single application execution, we repeatedly run the 10000-packet trace for multiple times for each application until its thermal behavior stabilizes, usually 1 to 2 minutes. During each time, we randomly enforce one of the three packet trace patterns as shown in Table 6.2. We start each application run only after the core temperature drops to its minimum, T_{am}, which we measure to be 35 degree from reading on-chip thermal sensor.

6.5.2 Model Verification

HotSpot Simulation

Table 6.4: Thermal parameters from HotSpot.

Application	T_{am}	T_{ss}	β	ρ
CRC	35 degree	116.7 degree	0.006051	0.006126
TL	35 degree	122.8 degree	0.006049	0.006126
Route	35 degree	120.7 degree	0.006058	0.006126
DRR	35 degree	117.3 degree	0.006054	0.006126
URL	35 degree	105.3 degree	0.006063	0.006126
MD5	35 degree	94.3 degree	0.006071	0.006126

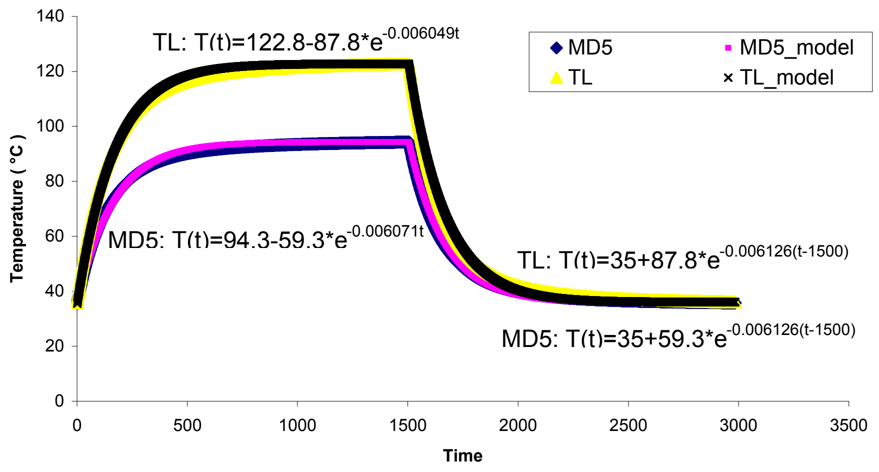

Figure 6.7: Model validation on HotSpot for single task.

In this section, we verify our model by running six applications on HotSpot and compare the HotSpot temperature with model temperature. Table 6.4 shows the model-related parameters derived from HotSpot simulator for all six applications. From this table we see that T_{am} and ρ are constant irrespective of applications, whereas T_{ss} and β are application specific, with T_{ss} ranging from 94.3 degree for MD5 to 122.8 degree for TL.

Figure 6.7 exhibits the model validation for single task in both temperature rising and falling phases. We show two applications with the highest T_{ss} (TL) and the lowest T_{ss} (MD5). Model equations are added next to their respective curves. For each application, we let it run until T_{ss} is reached, and then let the core cool down to T_{am}. Every time unit is equal to HotSpot sampling interval ($3.333\mu s$), which is used throughout this section. From this figure, we observe that the single task thermal model can accurately predict the temperature in both phases for real applications, with the average error being less than 1 degree.

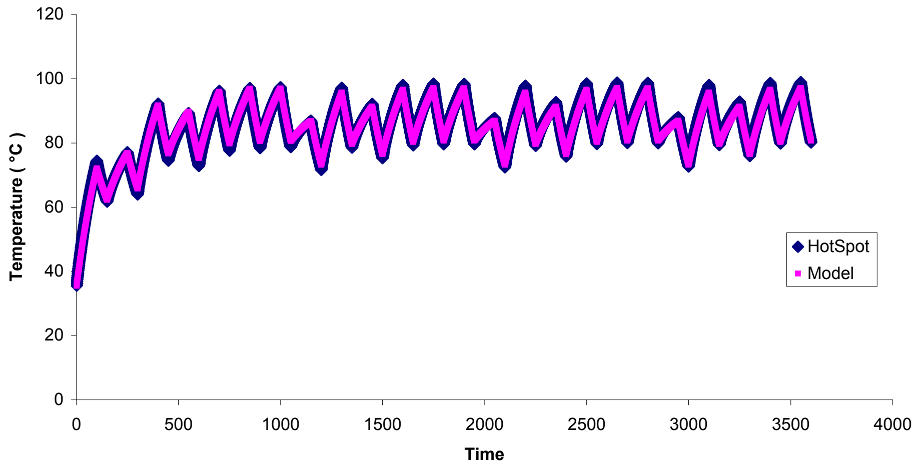

Figure 6.8: Model validation on HotSpot for periodic tasks.

Figure 6.8 shows the model verification for periodic tasks in a typical run with mixed workload and Simulated Distribution. For visualization purpose, we scale up the *busy* time and *idle* time in this way: we randomly pick one application and let it run for 100 time units and then cool down for 50 time units. we compare the HotSpot temperature and model temperature for the first 3600 time units, where 24 temperature rising phases and

24 temperature falling phases interleave with each other. From this figure, we see that the model temperature matches HotSpot temperature very well with only 0.8 degree difference on average.

Machine Measurement

Table 6.5: Thermal parameters from real machine.

Application	T_{am}	T_{ss}	β	ρ
CRC	35 degree	39 degree	1.4904	0.2517
TL	35 degree	44 degree	0.2547	0.2517
Route	35 degree	40 degree	0.4091	0.2517
DRR	35 degree	39 degree	1.4904	0.2517
URL	35 degree	39 degree	0.4318	0.2517
MD5	35 degree	42 degree	0.6213	0.2517

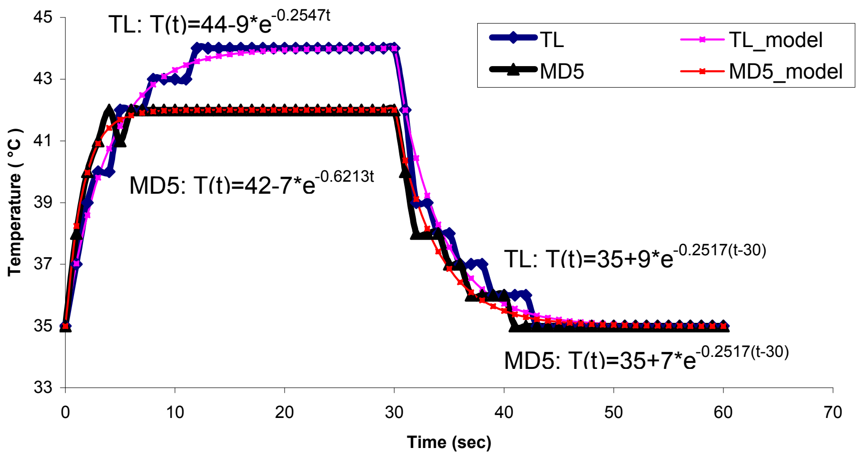

Figure 6.9: Model validation on real machine for single task.

In addition to HotSpot, we further conduct machine measurement to validate our model via running benchmark applications on a single core of the Intel Xeon E5335 server. Table 6.5 shows the model-related parameters obtained from real machine for each of the six applications. Compared to Table 6.4, we find out that T_{ss} on a real machine is far less

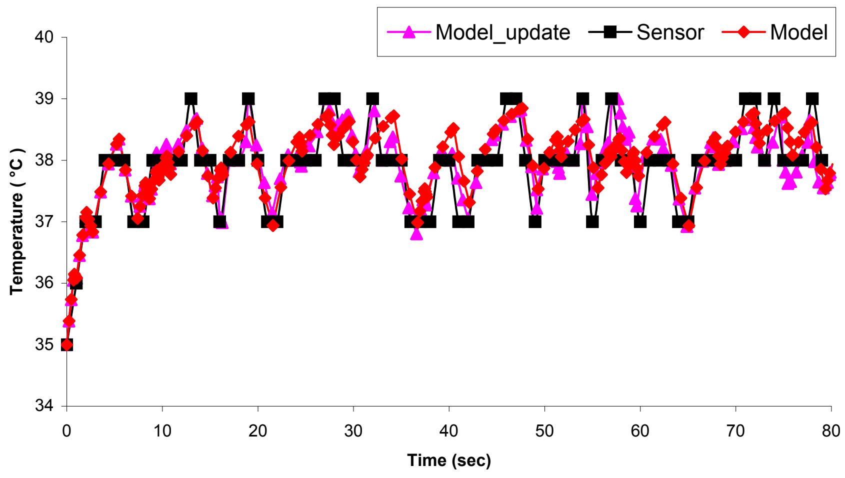

Figure 6.10: Model validation on real machine for periodic tasks.

than what we observe on HotSpot due to enforced cooling mechanism and different core architecture/technology. The range of T_{ss} on a real machine is from the lowest 39 degree for CRC, DRR and URL to the highest 44 degree for TL.

Figure 6.9 shows the temperature profile from both sensor reading and model prediction when running the same TL and MD5 as in Figure 6.7. We set the *busy* time and *idle* time to be 30 seconds each, so that T_{ss} will be reached during temperature rising phase and T_{am} will be reached during temperature falling phase. From this figure, we see that the mathematical model adequately represents the thermal characteristics for single task in both phases. Although the temperature sensor only returns discrete integer values, we still observe that the average difference is only 0.1 degree for MD5 and 0.2 degree for TL. In the worst case, the maximal differences for MD5 and TL are just 1.1 degree and 1.4 degree, respectively.

Figure 6.10 demonstrates model verification for periodic tasks and highlights the advantage of online model update. We run URL for 80 seconds in this experiment. For the sensor to capture the temperature variation, we repeat 10000 times for each packet and scale the packet interval according to Random Distribution. The sensor reads core temperature once a second and our model calculates temperature three times faster. Compared to sensor temperature, our model can nicely predict the temperature rising and falling phases. The real temperature values fall closely to the curve drawn from our model with the largest difference being 0.8 degree. However, this relatively large error, which is 20% of the 4 degree temperature fluctuation, is because we run this simulation without our automatic calibration.

When we add our online model update, we can improve the model accuracy as shown by Model_update curve. Because the sensor reading is rounded off to the integer, we set 0.5 degree as the threshold to apply update. In the experiment, we notice that there are 6 occurrences of underestimation and 12 occurrences of overestimation. After applying online update, the largest difference between model temperature and sensor temperature reduces to only 0.4 degree. With our online model update doing automatic calibration, we can hold the error to be within 0.5 degree even if there is larger temperature fluctuation.

6.5.3 Performance Evaluation

Temperature Behavior

In this section, we address the effectiveness of our scheduling scheme in achieving good thermal behavior of network applications by comparing with real-time core temperature. Figure 6.11 shows the temperature behavior on a randomly-selected core for different schemes for the first 25 seconds when executing MD5, whose T_{ss} is 42 degree and $T_{critical}$ is

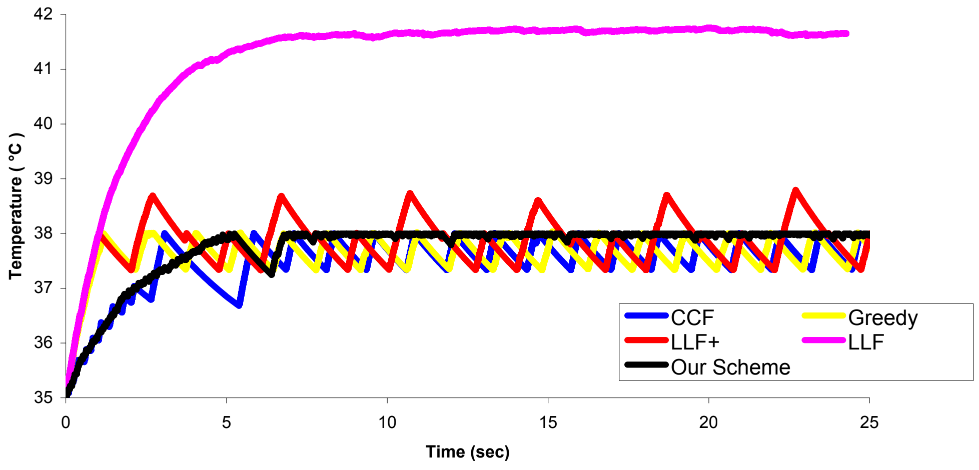

Figure 6.11: Temperature behavior of MD5 with five schedulers.

38 degree. From this figure, we observe that LLF suffers from the highest temperature at all times without thermal management. Its temperature varies around 41.7 degree, which is very close to T_{ss} and violates the given temperature constraint. By adding temperature control, LLF+ brings down the temperature but we see a zig-zag pattern across 38 degree. Although better than LLF, its drawbacks are obvious: 1) significant number of violations to temperature constraint occur, because temperature sensor is unable to respond to thermal emergency soon enough; and 2) a long cool-down interval (at least 1 second) that deteriorates throughput. Compared with LLF+, thermal-aware scheduling schemes Greedy and CCF achieve lower core temperature and they can quickly detect thermal emergency, substantially reducing the thermal constraint violations. However, once they apply Stop&Go when task migration is not available (e.g., system is heavily-loaded and there is no idle core), they still suffer from unnecessarily long cooling time (1 second). Our scheme, nevertheless, outperforms all others in the following three aspects: 1) it never violates $T_{critical}$ because of the predictive thermal model; 2) its temperature rising is similar to CCF, which is slower

than the other three since it is more thermally-balanced among cores; and 3) it avoids wide temperature fluctuation due to minimal cool-down interval (L_i) in Stop&Go. We can see that the temperature stays close to $T_{critical}$, which contributes to higher CPU utilization and better throughput. Results from other applications show similar temperature behavior.

Thermal Violation

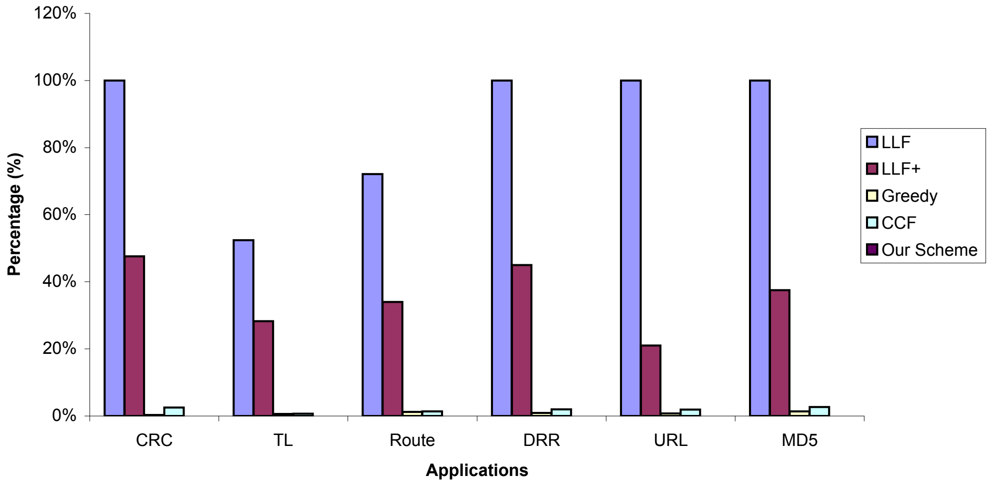

Figure 6.12: Thermal violation percentage for six applications.

We define the term *thermal violation percentage* as the time duration when the core temperature is over $T_{critical}$ divided by the total execution time on that core, representing how frequently the given temperature constraint is violated. Figure 6.12 shows the performance comparison for six applications in terms of *thermal violation percentage* on a randomly-selected core. From Figure 6.12, we observe that our scheme never violates the constraint at any time because, based on our predictive thermal model, it can proactively apply task migration or Stop&Go before thermal emergency occurs. In addition, the other two thermal-aware schemes, Greedy and CCF, also demonstrate negligible violations. Since both of them rely on our thermal model as opposed to thermal sensor for temperature monitoring, they can respond to thermal emergency fast enough. In contrast, LLF has the highest *thermal violation percentage* due to the lack of thermal management. Its temperature stays 100% over $T_{critical}$ except for TL (52.4%) and Route (72.1%), as both TL and Route have very fast per-packet execution time and thus naturally benefit from longer cool-down interval between packets. With respect to LLF+, although it applies Stop&Go in case of thermal emergency based on thermal sensor readings, it still exhibits substantial thermal violations ranging between 21.0% for URL and 47.6% for CRC. This is because 1) it can only passively react to thermal emergency after $T_{critical}$ is exceeded, and 2) its slow sampling rate fails to capture the thermal emergency immediately.

Thermal Balancing

We keep track of the largest real-time temperature difference between the highest and the lowest after the thermal behavior stabilizes during execution. Figure 6.13 shows the temperature variance of MD5 with different schedulers. From this figure, we observe that LLF, LLF+ and Greedy suffer the most as none of them consider thermal balancing in their schemes. Chances are that 1) some cores are heated up more frequently, and 2) some cores are sitting idle most of the time. We see at least one core with the ambient temperature (35 degree) in all three schemes. More specifically, LLF performs the worst with temperature ranging between 35 degree and 41.7 degree at certain time point. With sensor-based thermal management, LLF+ can reduce the top temperature to 38.9 degree, which mitigates the thermal imbalance problem to some degree compared LLF. Lastly, Greedy further improves thermal balancing among cores by limiting the highest temperature to 38 degree. Compared to these three schemes, CCF and our scheme have much better thermal balancing performance. CCF's temperature spans a short range between 36.7 degree and 38 degree, as it always schedules the next packet to the coldest core. However, it still suffers from the fixed cool-down interval (1 second), which unnecessarily increases the temperature variance among cores. Our scheme, nevertheless, is able to minimize the worst case temperature difference to only 0.4 degree (between 37.6 degree and 38 degree) among all 8 cores. This is because 1) we always schedule the next packet to the coldest idle core and 2) we use the minimal cool-down interval derived from our thermal model in Stop&Go. It is worth noting that similar observations also apply to other applications.

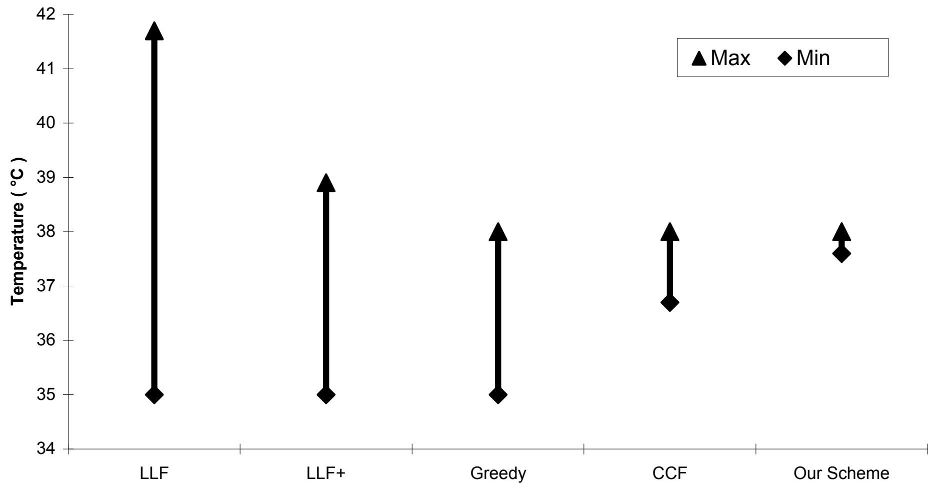

Figure 6.13: Temperature variance of MD5 with five schedulers.

Throughput Performance

We define *throughput percentage* as the normalized throughput in comparison to baseline LLF (100%). Except LLF, all other schemes suffer from throughput degradation to some extent due to enforced thermal constraint. However, as shown in Figure 6.14, we still see that our scheme outperforms the other three schedulers, namely, LLF+, Greedy and CCF for all six applications with an average of 28.1% throughput improvement and a maximum of 65.4% for MD5. As LLF+, Greedy and CCF achieve similar load balancing and have the same cool-down interval while applying Stop&Go, their throughput are more or less the same. Compared to them, the superior performance of our scheme results from the appropriate combination of task migration and Stop&Go in a cost-effective and cache-aware fashion due to our predictive thermal model. In case of task migration, we prefer to migrate current task to a nearby core with less communication overhead. In case of Stop&Go, we guarantee the minimal cool-down interval to maximize overall throughput. Our scheme has higher CPU utilization with runtime temperature maintained at near the threshold level (Figure 6.11), which also leads to higher throughput. In addition, we observe that our scheme does not cause substantial throughput degradation in four out of six applications compared to LLF. The average performance loss for Route, DRR, URL and MD5 is only 7.8% in our scheme, as opposed to an average loss of 36.5% for the other three schemes. Our scheduler is more suitable for applications that require longer execution time (IP level or Application level as classified in Table 6.1), where thermal management overhead only takes a small portion of the total processing time.

Scheduling Overhead

We first compare the absolute time to obtain one temperature value from the model and sensor. Experiments show that our model takes only $0.1\mu s$ to compute one

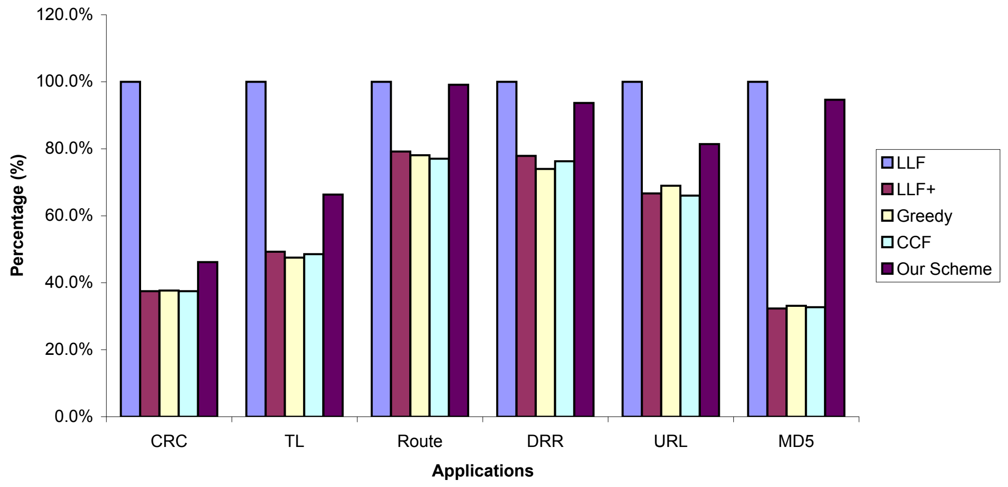

Figure 6.14: Throughput percentage for six applications.

temperature, whereas it takes $8300\mu s$ for one reading from on-chip thermal sensor. This is because our scheme is completely integrated into a user-level program. It takes the timing information and one history temperature to calculate the current temperature from an exponential equation that does not need many CPU cycles. On the contrary, LLF+ relies on a monitoring tool (i.e., lm-sensors) to call Intel's Coretemp driver to access on-chip thermal sensors, which involves expensive system calls, interrupts, and mode switches, thus incurring significant time overhead.

We next compare the scheduling overhead of our scheme with respect to packet execution time for six applications and show the minimal, average, and maximal overhead in Figure 6.15. In our experiment, the model will be invoked at least once for each incoming packet (line 6 in Algorithm 2). If thermal emergency occurs while executing this packet, extra time will be spent on calculating and comparing the overhead for Stop&Go and task migration (line 8-15 in Algorithm 2). In such a case, the task migration overhead or the cool-down interval for Stop&Go take much longer time. Therefore, we only consider one

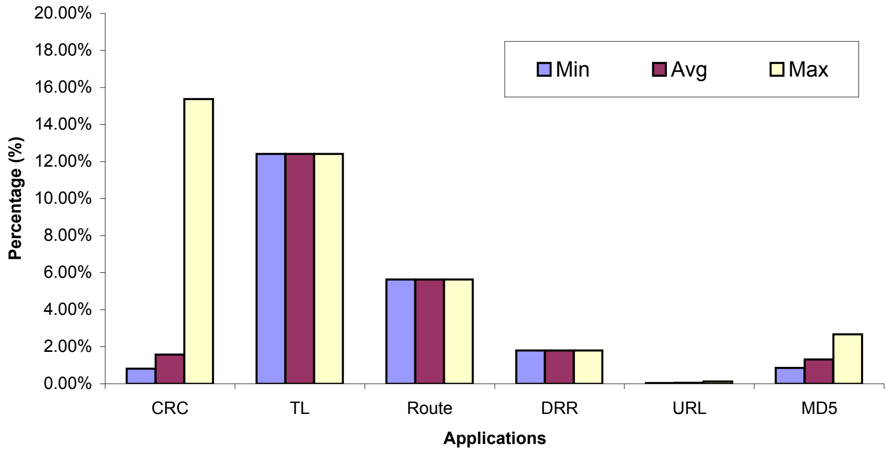

Figure 6.15: Scheduling overhead per-packet for six applications.

model invocation per packet as scheduling overhead. From Figure 6.15 we observe that the average overhead ranges from 0.06% for URL to 12.42% for TL, and the worst case happens when CRC is executing a 1500-byte packet (15.38%). We also see that TL, Route and DRR have constant overhead (12.42%, 5.64% and 1.80%) since their packet execution time is independent of packet size, and CRC, URL and MD5 exhibit varying overhead from the lowest 0.04% (URL for 40-byte packet) to the highest 15.38% (CRC for 1500-byte packet). In summary, our thermal-aware scheduler does not cause significant overhead and is suitable for network applications.

Chapter 7

Conclusion

The multicore architecture has prevailed in every aspect of computing platforms including desktops, servers, and embedded systems. The superiority of multicore systems comes from high performance, low cost, and good programmability. Accordingly, in the domain of network applications, we see a shift from single-core system, to multicore servers to accommodate high traffic volume with computationally intensive applications. Along with increased throughput, however, comes significantly increased power consumption. Higher power consumption also increases core temperature, which exponentially increases the cost of cooling and packaging, as well incurs indirect and life-cycle costs due to reduced system performance, circuit reliability, and chip lifetime. Therefore, it is critical to run such applications in a power efficient manner, which naturally gives rise to research on "intelligent" scheduling.

In this thesis, we focused on power-efficient scheduling for network applications on multicore architecture. Our goal is to improve the performance of network applications not only in throughput, but also in latency, power, energy, and temperature.

In Chapter 3, we designed, implemented and evaluated LATA, a latency and throughput-aware packet processing system. By adopting hybrid parallelism with parallel pipeline core topology in fine-grained task level, LATA is able to achieve both low latency and high throughput. LATA consists of a list-based pipeline scheduling algorithm, a deterministic search-based refinement process, and a cache-aware resource mapping scheme. Compared with other approaches (Parallel, Greedy, Random and Bipar) for six real network applications, LATA exhibits an average of 36.5% reduction in latency across all applications and a maximum of 62.2% reduction in latency for URL application over Random with comparable throughput performance.

Secondly, we proposed a novel algorithm to optimize both throughput and latency given a power budget for network packet processing on multicore architectures in Chapter 4. This algorithm addresses the power-aware parallel-pipeline scheduling problem by applying per-core DVFS to optimally adjust frequency on each core. We implemented our algorithm in addition to five other conventional algorithms on an AMD machine with two Quad-Core Opteron 2350 processors. Compared to existing algorithms and given the same power budget for six real packet processing applications, our algorithm exhibits substantially better throughput and latency by an average of 64.6% and 25.2%, respectively.

Thirdly, in Chapter 5, we designed a traffic-aware and power-efficient multicore server system that appropriately changes the system operating level according to varying traffic rate and dynamically adjusts the per-core frequency configuration using a combination of per-core DVFS, power gating, and power migration techniques to minimize power consumption. The system optimally configures the number of active cores and per-core frequency in real-time to handle traffic variation based on our power model that considers both dynamic and static power consumption of all cores. Our experimental results show that, on an average, our system saves 41.0% power compared to a native system. It also consumes less power than three other approaches, C-DVFS [81], PG [65], and C-Hybrid [60], by 35.2%, 24.3%, and 10.5% respectively.

Lastly, we explored temperature related issues by proposing a predictive model-based thermal-aware scheduling scheme in Chapter 6. This scheduling scheme is based on 1) a novel predictive thermal model for generic periodic tasks, which can dynamically derive the core temperature at any time, and 2) an online model update, which uses on-chip thermal sensors to effectively correct incidental errors by adjusting model parameters dynamically. Our scheduler is capable of maximizing throughput and achieving thermal balancing without violating thermal constraint by appropriately combining task migration and Stop&Go techniques in a cost-effective and cache-aware fashion. To verify the model and evaluate the scheduler, we used both the HotSpot simulator and a real Linux multicore server to run six network applications chosen from NetBench. Extensive results showed that our model can predict real temperature quickly and accurately, while achieving higher throughput, lower temperature and better thermal balancing with negligible scheduling overhead and no thermal constraint violation compared with other schemes.

Some potential future research directions are listed below. First, this thesis only focused on single application, but concurrent execution of multiple applications should be studied in future. It will be challenging to extend the single application techniques to multiple applications with minimal changes. Second, while we are concerned with metrics such as throughput, power and temperature, we have not yet considered quality of service (QoS) for different packet streams. QoS should be considered when developing scheduling schemes for network applications. Third, it would be interesting to make task-level scheduling schemes presented in Chapter 3 and Chapter 4 dynamic, as the task execution time may vary significantly with different packet sizes. The existing static approaches can only handle packets with fixed or bounded execution time.

Bibliography

[1] Amd opteron machine. http://www.amd.com/opteron.

[2] Cavium octeon processor family. http://www.caviumnetworks.com/OCTEON_MIPS64.html.

[3] Cisco AON Technology. http://www.cisco.com/en/US/products/ps6692/Products_Sub_Category_Home.html.

[4] Equinix-sanjose Internet Monitor. http://www.caida.org/data/monitors/passive-equinix-sanjose.xml.

[5] Extech power analyzer. http://extech.com/instruments/.

[6] IBM BladeCenter System. http://www-03.ibm.com/systems/bladecenter/.

[7] IBM POWER7 Systems. http://www-03.ibm.com/systems/power/.

[8] Intel ixp2xxx product line of network processors. http://intel.com/design/network/products/npfamily/index.htm.

[9] Intel xeon machine. http://www.Intel.com/Xeon.

[10] International Technology Roadmap for Semiconductors. http://public.itrs.net.

[11] Lm-Sensors. http://lm-sensors.org/.

[12] Lmbench. http://www.bitmover.com/lmbench/index.html.

[13] Machine-suif, harvard university. http://eecs.harvard.edu/hube/software/nci/overview.html.

[14] Netfpga. http://www.netfpga.org/.

[15] Papi. http://icl.cs.utk.edu/papi/.

[16] Suif compiler system. http://suif.stanford.edu/.

[17] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 1974.

[18] Min Bao, Alexandru Andrei, Petru Eles, and Zebo Peng. Temperature-aware idle time distribution for energy optimization with dynamic voltage scaling. In *Proc. of DATE '10*, 2010.

[19] Frank Bellosa, Simon Kellner, Martin Waitz, and Andreas Weissel. Event-driven energy accounting for dynamic thermal management. In *Proc. of COLP '03*, 2003.

[20] Mats Bjorkman and Per Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proc. of SIGCOMM '93*, 1993.

[21] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of HPCA '01*, 2001.

[22] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of ISCA '00*, 2000.

[23] Louis Burmeister. *Convective Heat Transfer.* John Wiley & Sons, New York, NY, 1993.

[24] Pedro Chaparro, Jose Gonzalez, Grigorios Magklis, Qiong Cai, and Antonio Gonzalez. Understanding the thermal implications of multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[25] Michael Chen, Xiao Feng Li, Ruiqi Lian, Jason Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of PLDI '05*, 2005.

[26] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Hendrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *Proc. of ISLPED '07*, 2007.

[27] M. Chol, N. Sathe, M. Gupta, S. Kumar, S. Yalamanchilli, and S. Mukhopadhyay. Proactive power migration to reduce maximum value and spatiotemporal non-uniformity of on-chip temperature distribution in homogeneous many-core processors. In *Proc. of SEMI-THERM '10*, 2010.

[28] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Kenny C. Gross. Temperature management in multiprocessor socs using online learning. In *Proc. of DAC '08*, 2008.

[29] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proc. of ICS '06*, 2006.

[30] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proc. of PLDI '05*, 2005.

[31] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proc. of ISCA '06*, 2006.

[32] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *Proc. of ISSCC '07*, 2007.

[33] Hesham El-Rewini, Hesham Ali, and Ted Lewis. Task scheduling in multi-processing systems. *IEEE Transactions on Computers*, 1995.

[34] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[35] N. Fisher, J.-J Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *Proc. of RTAS '09*, 2009.

[36] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L Rawson, J. C. Rubio, and M. S. Ware. System power management support in the ibm power6 microprocessor. *IBM Journal of Research and Development*, 2007.

[37] Johan De Gelas. Dynamic power management: A quantitative approach. *AnandTech IT Computing*, 2010.

[38] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*.

[39] Kate Greene. Data centers' growing power demands. *MIT Technology Review*, 2007.

[40] Flavius Gruian. System-level design methods for low-energy architectures containing variable voltage processors. In *Proc. of PACS '00*, 2000.

[41] Flavius Gruian and Krzysztof Kuchcinski. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *Proc. of ASP-DAC '01*, 2001.

[42] Danhua Guo, Guangdeng Liao, Laxmi Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proc. of ANCS '08*, 2008.

[43] Vinay Hanumaiah, Ravishankar Rao, Sarma Vrudhula, and Karam Chatha. Throughput optimal task allocation under thermal constrains for multi-core processors. In *Proc. of DAC '09*, 2009.

[44] Shaoxiong Hua and Gang Qu. Power minimization techniques on distributed real-time systems by global and local slack management. In *Proc. of ASP-DAC '05*, 2005.

[45] Christopher Hughes, Jayanth Srinivasan, and Sarita Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proc. of Micro '01*, 2001.

[46] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. of Micro '06*, 2006.

[47] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. of Micro '06*, 2006.

[48] Ramkumar Jayaseelan and Tulika Mitra. Temperature aware task sequencing and voltage scaling. In *Proc. of ICCAD '08*, 2008.

[49] Vida Kianzad, Shuvra S. Bhattacharyya, and Gang Qu. Casper: An integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *Proc. of ASAP '05*, 2005.

[50] W. Kim, M. Gupta, G. Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proc. of HPCA '08*, 2008.

[51] Ravi Kokku, Upendra B. Shevade, Nishit S. Shah, Mike Dahlin, and Harrick M. Vin. Energy-efficient packet processing. *University of Texas at Austin Technical Report TR04-04*, 2004.

[52] Frank Kreith, Raj Manglik, and Mark Bohn. *Principles of Heat Transfer*. Cengage Learning, Stamford, CT, 2003.

[53] Jilong Kuang and Laxmi Bhuyan. Lata: A latency and throughput-aware packet processing system. In *Proc. of DAC '10*, 2010.

[54] Jilong Kuang and Laxmi Bhuyan. Optimizing throughput and latency under given power budget for network packet processing. In *Proc. of INFOCOM '10*, 2010.

[55] Jilong Kuang, Laxmi Bhuyan, Haiyong Xie, and Danhua Guo. E-ahrw: An energy-efficient adaptive hash scheduler for stream processing on multi-core servers. In *Proc. of ANCS '11*, 2011.

[56] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj Jha. Hybdtm: A coordinated hardware-software approach for dynamic thermal management. In *Proc. of DAC '06*, 2006.

[57] R. Kumar and G. Hinton. A family of 45nm ia processors. In *Proc. of ISSCC '09*, 2009.

[58] Eren Kursun, Chen-Yong Cher, Alper Buyuktosunoglu, and Pradip Bose. Investigating the effects of task scheduling on thermal behavior. In *Proc. of TACS '06*, 2006.

[59] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 1999.

[60] Jungseob Lee and Nam Sung Kim. Optimizing throughput of power- and thermal-constrained multicore processors using dvfs and per-core power-gating. In *Proc. of DAC '09*, 2009.

[61] Stefan Leue and Philippe A. Oechslin. On parallelizing and optimizing the implementation of communication protocols. *IEEE Transactions on Networking*, 1996.

[62] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Partha Ranganathan, and Christos Kozyrakis. Power management of datacenter workloads using per-core power gating. *HP Labs Technical Report HPL-2009-326*, 2009.

[63] Jian Li and Jose F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proc. of HPCA '06*, 2006.

[64] Yan Luo, Jia Yu, Jun Yang, and Laxmi Bhuyan. Low power network processor design using clock gating. In *Proc. of DAC '05*, 2005.

[65] Yan Luo, Jia Yu, Jun Yang, and Laxmi Bhuyan. Conserving network processor power consumption by exploiting traffic variability. *ACM Transactions on Architecture and Code Optimization*, 2007.

[66] Arindam Mallik, Yu Zhang, and Gokhan Memik. Automated task distribution in multicore network processors using statistical analysis. In *Proc. of ANCS '07*, 2007.

[67] Rich McGowen, Christopher A. Poirier, Chris Bostak, Jim Ignowski, Mark Millican, Warren H. Parks, and Samuel Naffziger. Power and temperature control on a 90-nm itanium family processor. *IEEE Journal of Solid-State Circuits*, 2006.

[68] Gokhan Memik, William Mangione-Smith, and Wendong Hu. Netbench: A benchmarking suite for network processors. In *Proc. of ICCAD '01*, 2001.

[69] Pierre Michaud, Andre Seznec, Damien Fetis, Yiannakis Sazeides, and Theofanis Constantinou. A study of thread migration in temperature-constrained multicores. *ACM Transactions on Architecture and Code Optimization*, 2007.

[70] Ramesh Mishra, Namrata Rastogi, and Dakai Zhu. Energy aware scheduling for distributed real-time systems. In *Proc. of IPDPS '03*, 2003.

[71] Fabrizio Mulas, David Atienza, Andrea Acquaviva, Salvatore Carta, Luca Benini, and Giovanni De Micheli. Thermal balancing policy for multiprocessor stream computing platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[72] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, Luca Benini, and Giovanni De Micheli. Temperature control of high-performance multi-core platforms using convex optimization. In *Proc. of DATE '08*, 2008.

[73] Terry Nelms and Mustaque Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *Proc. of ANCS '10*, 2010.

[74] nVIDIA. Firstpacket technology improved system performance. *nVIDIA Technical Brief TB-02434-001_v01*, 2006.

[75] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of SOSP '01*, 2001.

[76] Michael Powell, Mohamed Gomaa, and T. Vijaykumar. Heat-and-run: Leveraging smt and cmp to manage power density through the operating system. In *Proc. of ASPLOS '04*, 2004.

[77] John W. Pratt. F. y. edgeworth and r. a. fisher on the efficiency of maximum likelihood estimation. *The Annals of Statistics*, 1976.

[78] Yaxuan Qi, Bo Xu, Fei He, Baohua Yang, Jianming Yu, and Jun Li. Towards high-performance flow-level packet processing on multi-core network processors. In *Proc. of ANCS '07*, 2007.

[79] Ramaswamy Ramaswamy and Tilman Wolf. Packetbench: A tool for workload characterization of network processing. In *Proc. of WWC '03*, 2003.

[80] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. Iterative schedule optimization for voltage scalable distributed embedded systems. *ACM Transactions on Embedded Computing Systems*, 2004.

[81] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling for portable systems. In *Proc. of DAC '01*, 2001.

[82] Oliver Sinnen. *Task Scheduling For Parallel Systems.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[83] Kevin Skadron, Tarek Abdelzaher, and Mircea Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Proc. of HPCA '02*, 2002.

[84] Kevin Skadron, Mircea Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proc. of ISCA '03*, 2003.

[85] Kevin Skadron, Mircea Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 2004.

[86] Jayanth Srinivasan and Sarita Adve. Predictive dynamic thermal management for multimedia applications. In *Proc. of ICS '03*, 2003.

[87] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of ISCA '08*, 2008.

[88] Shengquan Wang and Riccard Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *Proc. of RTSS '06*, 2006.

[89] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proc. of ISCA '09*, 2009.

[90] Zhe Wang and Sanjay Randa. A simple thermal model for multi-core processors and its application to slack allocation. In *Proc. of IPDPS '10*, 2010.

[91] Ning Weng and Tilman Wolf. Pipelining vs multiprocessors-choosing the right network processor system topology. In *Proc. of ANCHOR'04*, 2004.

[92] Guowei Wu and Zichuan Xu. Temperature-aware task scheduling algorithm for soft real-time multi-core systems. *Journal of Systems and Software*, 2010.

[93] Jun Yang, Xiuyi Zhou, Marek Chrobak, Youtao Zhang, and Lingling Jin. Dynamic thermal management through task scheduling. In *Proc. of ISPASS '08*, 2008.

[94] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[95] Jingnan Yao, Yan Luo, Laxmi Bhuyan, and Ravishankar Iyer. Optimal network processor topologies for efficient packet processing. In *Proc. of Globecom '05*, 2005.

[96] Inchoon Yeo and Eun Jung Kim. Temperature-aware scheduler based on thermal behavior grouping in multicore systems. In *Proc. of DATE '09*, 2009.

[97] Inchoon Yeo, Chih Chun Liu, and Eun Jung Kim. Predictive dynamic thermal management for multicore systems. In *Proc. of DAC '08*, 2008.

[98] Jia Yu, Jingnan Yao, Laxmi Bhuyan, and Jun Yang. Program mapping onto network processors by recursive bipartitioning and refining. In *Proc. of DAC '07*, 2007.

[99] Sushu Zhang and Karam Chatha. System-level thermal aware design of applications with uncertain execution time. In *Proc. of ICCAD '08*, 2008.

[100] Yifan Zhu and Frank Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *Proc. of RTAS '04*, 2004.

[101] Martina Zitterbart. A multiprocessor architecture for high speed network interconnections. In *Proc. of INFOCOM '89*, 1989.